THE DESIGN OF SYSTEMS FOR TELECOMMUNICATIONS BETWEEN SMALL AND LARGE COMPUTERS

Robin B. John

Thesis presented for the Degree of Doctor of Philosophy

Faculty of Science, University of Edinburgh

November, 1973.

ABSTRACT

This thesis describes the development of a data communication system for small computers to enable them to link to large computers. The particular advantages and additional facilities made available to computer users through the use of such a link are described. A detailed description is given of the hardware and software components needed to achieve this link, together with the reasons for choosing the particular techniques employed. The discussions given highlight the problems involved in this type of operation. Some of these problems, such as lack of standardization, are short-term and will be overcome with the natural evolution of computer systems, while others are of a more fundamental nature related to the use of data transmission over long distances.

The system was designed to be applicable to a number of different small computers. This has resulted in a system which is easily transferable between machines, through the careful choice of interfaces to other components. This is seen as a step towards a more flexible and more modular method of system construction whereby complete software systems for arbitrary configurations can be put together using 'off-the-shelf' components already well-developed and tested. This contrasts with the present situation in which whole new systems are developed for a new computer, frequently duplicating systems already developed on other hardware. A detailed description of the factors involved in producing machine-independent, easily-transferable system components is given as a guide to other developments in this direction. It is felt that there is need for a better-engineered approach to the construction of software systems and it is hoped that the work described makes some contribution towards this end.

CONTENTS

Chapter 1 1

TELECOMMUNICATIONS BETWEEN SMALL AND LARGE COMPUTERS

1.1 Introduction 1

1.2 Small computer capabilities 1

1.3 Central computers 3

1.4 Communications requirement 3

1.5 Summary of developments 4

1.6 The General System approach 6

Chapter 2 8

USER APPLICATIONS FOR SMALL COMPUTER COMMUNICATION LINKS

2.1 ERCC PDP-8/L 8

2.2 Physics PDP-8 10

2.3 Social Medicine PDP-8 13

2.4 Computer Science CAD Project PDP-7 16

2.5 IBM 1130 in ABRO and Dept. of Statistics 19

2.6 Conclusions 21

Chapter 3 24

GENERAL COMMUNICATIONS HARDWARE CONSIDERATIONS

3.1 Main features of data communication hardware . . . 24

3.2 Asynchronous communications adapter 26

3.3 Synchronous communications adapter 27

3.4 Telegraphic signalling 29

3.5 Modems 30

3.6 Electrical circuit facilites 33

3.7 Conclusions 35

Chapter 4 37

SYNCHRONOUS V. ASYNCHRONOUS FOR COMPUTER-COMPUTER COMMUNICATIONS

4.1 Introduction 37

4.2 Handling of transmission errors 38

4.3 Error detection 39

4.4 Block-oriented error detection 41

4.5 Comparisons 42

4.6 Conclusions 43

Chapter 5 44

LARGE COMPUTER COMMUNICATIONS HARDWARE

5.1 Introduction 44

5.2 Communications controllers 44

5.3 IBM synchronous controllers 46

5.4 ICL synchronous controller 47

5.5 Problem of compatibility 48

5.6 Conclusions 49

Chapter 6 50

INTERFACING TO THE SMALL COMPUTER

6.1 Introduction 50

6.2 The 'uniform interface' requirement 50

6.3 Small computer input/output interfaces 52

6.4 PDP-8 53

6.5 Modular One 56

6.6 ICL 4100 58

6.7 IBM 1130. 61

Chapter 6 50

INTERFACING TO THE SMALL COMPUTER (cont.)

6.8 Common facilities of the input/output interfaces. . 64

6.9 Generalized interface description 66

6.10 General applicability 67

Chapter 7 69

SMALL COMPUTER COMMUNICATIONS HARDWARE

7.1 Introduction 69

7.2 Flexibility requirement 69

7.3 Basic functional requirements 70

7.4 Extra functions 72

7.5 Preliminary functional specification 75

7.6 Non-interrupt mode of operation 77

7.7 Program interface 77

7.8 Communications interface specification. 82

7.9 Experience of first implementation. 85

7.10 Improved functional specification 89

7.11 Interface test facility 91

7.12 Conclusions on small computer communications hardware . . . 92

Chapter 8 94

SPECIFICATION FOR SYNCHRONOUS COMMUNICATIONS INTERFACE FOR SMALL COMPUTERS

8.1 Basic requirement 94

8.2 Modem interface consierations 94

8.3 Program control 95

Chapter 8 94

SPECIFICATION FOR SYNCHRONOUS COMMUNICATIONS INTERFACE FOR SMALL COMPUTERS (cont.)

8.4 General 95

8.5 Receive (RX) channel 95

8.6 Modem interface considerations for RX channel . . . 98

8.7 Transmit (TX) channel 98

8.8 Modem interface considerations for TX channel . . . 100

8.9 Timeout control 101

8.10 Interrupt control 101

Chapter 9 102

COMMUNICATIONS SOFTWARE FOR THE SMALL COMPUTER

9.1 Overview 102

9.2 General requirements. 103

9.3 Package implementation-general details. 104

9.4 The user interface 106

9.5 The communications hardware interface 112

9.6 The executive interface 119

9.7 Remote host computer interface 120

9.8 Conclusions 123

Chapter 10 124

IMPLEMENTATION DETAILS IN ACTUAL SYSTEMS

10.1 Summary 124

10.2 PDP-8 with ERCC communications controller 126

10.3 PDP-8 with Data Dynamics 6310 controller 128

10.4 ICL 4100 with ERCC communications controller . . 132

10.5 Modular One with 1.61 communications multiplexor. 133

10.6 PDP-11 with DP11 communications controller . . . 140

10.7 Conclusions 144

Chapter 11 146

IMPLEMENTING THE COMMUNICATIONS SYSTEM ON A NEW SMALL COMPUTER

11.1 Introduction 146

11.2 General 147

11.3 Difficulty of 'live' testing 147

11.4 User program. 148

11.5 Communications hardware 150

11.6 Communications software 153

11.7 Hardware interfacing routines 173

11.8 User interface routines and executive interface . 177

11.9 Conclusions about transferability 179

Chapter 12 180

COMMUNICATIONS PROTOCOLS FOR INTER-COMPUTER WORKING

12.1 Introduction 180

12.2 General characteristics of protocols 181

12.3 Examination of existing protocols 182

Chapter 12 180

COMMUNICATIONS PROTOCOLS FOR INTER-COMPUTER WORKING

12.4 Applicability of protocols 183

12.5 Efficiency of protocols 183

12.6 Implementation complexity 187

12.7 Half-duplex protocol with one-way data traffic . 187

12.8 Half-duplex with two-way transmission interleaved 196

12.9 Full-duplex with two-way simultaneous traffic . . . 211

12.10 Conclusions on communications protocols 217

Chapter 13 220

SYSTEM PROGRAMMING ON SMALL COMPUTERS WITH HIGH-LEVEL LANGUAGES

13.1 Introduction 220

13.2 Compilation on a different computer 222

13.3 Language facilities - general consideration . . . 224

13.4 Data elements 228

13.5 Data structures 229

13.6 Machine address manipulation 230

13.7 Program structure and statements 232

13.8 Expressions 233

13.9 Conditional statements 234

13.10 Programmed loops 236

13.11 Programmed transfers 236

13.12 Machine-code 238

13.13 Routines and functions 239

13.14 Block structure 241

13.15 Run-time environment 243

13.16 Conclusions 244

Chapter 14 247

CONCLUSIONS

14.1 Introduction 247

14.2 Uses of computer telecommunications 247

14.3 Technical aspects of data communications 250

14.4 System construction techniques 253

14.5 Transferable system components 254

14.6 Interfaces 256

14.7 Real-time components 257

14.8 Use of high-level languages 258

14.9 Transferable hardware 258

14.10 System construction in the future 259

14.11 Further developments 262

ACKNOWLEDGEMENTS 264

REFERENCES 265

Appendix A

Chronology of significant developments, indicating items of work involving other people.

Appendix B

Other available documents on particular implementations.

Chapter 1

TELECOMMUNICATIONS BETWEEN SMALL AND LARGE COMPUTERS

1.1 Introduction

The development of computing to support research and teaching work in Edinburgh University has involved the use of both a large, central computer run as a general university service and a number of small computers local to one department or research unit. A similar pattern of development has taken place in other universities, as well as many other large institutions using computers in diverse applications. The central computer provided the capability for handling large-scale jobs in respect of core store, processing requirements and backing storage. The small computers were applied to local small-scale computing problems, such as on-line control of experiments, data collection and reduction, interactive graphics, etc. There was a wide variety of such computers in use in Edinburgh and a small list is given in Table 1.1 to illustrate some of the applications.

1.2 Small computer capabilities

These small computers were performing tasks which would be difficult to implement in an efficient way on large general - purpose computers, because of the requirement for fast response to real - time events, 'hands-on' access, attachment of special peripherals, etc. For these reasons, it was not feasible to use the central computer directly for these applications.

Computer	Location	Applications
PDP-8	Physics	Experiment control; data collection
PDP-8	Social Medicine	Analysis of survey data
PDP-7	Computer Science	CAD work
PDP-8	Computer Science	Teaching and research
IBM1130	Dept. of Statistics	Statistical analysis problems
IBM1130	ABRO	Analysis of experimental data
ICL4130	School of Mach. Int.	Machine Intelligence Research
ICL903	Royal Infirmary	Path. Lab. work; medical data analysis

Table 1.1

However, while the small computer was best suited to the special - purpose system, it did not generally have the computing power, store size or backing store capacity to handle the larger non - real - time problems that followed, such as the processing of experimental data by applications programs, or even the compilation of programs written in a comprehensive high-level language.

These difficulties could be overcome by providing the small computer with easy access to a powerful backup computing facility by means of a direct communications link. In other words, the small computer could have the same degree of access to large - scale computing power as the on-line teletype user. However, a much higher data rate would be necessary for the link to be useful to the small computer since the volume of data involved would be much greater than could be sensibly handled at a teletype speeds.

Furthermore, the small computer would not be limited in its speed of operation by human operator action times, and therefore the data rate would be limited only by technical and economic considerations.

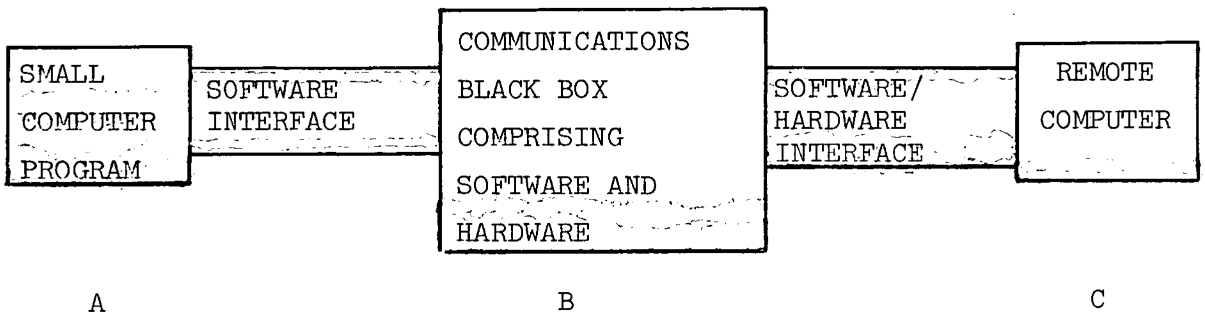
1.3 Central computers

There were two relevant central computers in Edinburgh in 1969. The first was the ICL 4-75 which was installed in early 1969 and for which a sophisticated multi-access system was being developed jointly by ICL and the Edinburgh University Computer Science Department. The second was the IBM 360/50 which was installed in mid 1969 to take over the batch processing workload from the ICL KDF9 which was the first central computer installed in the University. The central facility was run in all cases by the Edinburgh Regional Computing Centre (ERCC). The use of other large computers was also considered as the source of backup computing power in order to allow for any possible future developments. All large computers currently available seemed to provide the capability for communication at speeds significantly higher than teletype speeds.

1.4 Communications requirement

The general requirement, then, was to provide a facility for all the small computers currently in use or likely to be used by individual departments which would enable them to communicate at high speed with any possible large mainframe computer, the initial target machines being the ICL 4-75 and the IBM 360/50.

This requirement can be shown diagrammatically as follows:


SMALL COMPUTER PROGRAM
SOFTWARE INTERFACE
COMMUNICATIONS BLACK BOX COMPRISING SOFTWARE AND HARDWARE
SOFTWARE/ HARDWARE INTERFACE
REMOTE COMPUTER
A
B
C

A suitable implementation of item B had to be devised which would match the requirements of item A against the constraints imposed by accessing item C via a communications link. This implementation of item B should provide facilities whereby the user program could send and receive any type of data that he might wish to process on the remote computer. Furthermore, item B should be constructed in such a way that it would require a minimum of effort to transfer it to any other small computer once the first implementation had been completed successfully.

1.5 Summary of developments

In order to achieve this objective, detailed studies were made of a number of areas as follows:-

a) user applications implemented by item A in order to determine the facilities to be provided.

b) the communications facilities supported by the possible large remote computers, particularly the ICL 4-75 and the IBM 360/50.

c) the communications facilities provided by the GPO.

d) the communications facilities provided by the various small computers.

e) the development of communications hardware suitable for attaching to any small computer in the event that there was no suitable communications peripheral available.

f) ways of incorporating new peripheral - handling software into the small computer software system, involving direct physical control of the peripheral.

g) the use of a high-level language for the small computer software to facilitate the transfer to other small computers.

h) the development of software construction techniques to aid the debugging of the real-time, interrupt - driven software needed to handle the communications link.

i) communications techniques for achieving maximum utilization of the communications link while guaranteeing error - free transmission.

These studies are described in later sections of this report.

As a result of these studies, a communications system has been developed which has been successfully implemented on a number of small computers. This system comprises both software and hardware, although in some cases where the computer had suitable communications hardware, only the software component was needed. The software is constructed in such a way that it will work with any suitable communications hardware, irrespective of the way the hardware is programmed. Because of the ease with which the system has been mounted on different computers, even one which did not exist when the original studies were made, there is now a high degree of confidence that the system can be applied to any small computer.

A detailed description of the implementation of this system, together with a description of the routine procedures for applying it to a new computer, is given in subsequent sections of this report.

Two other sections of this report give a general discussion of computer-to-computer communications protocols and a general discussion on the use of high-level languages on small computers. The last section attempts to summarize the particular aspects of the work described which are considered to be of importance, both in the specialized area of communications and in the more general area of system construction and development.

1.6 The General System Approach

Although other work has been described in the literature on the subject of communication links for small computers, notably (1) and (2), no attempt has been made to produce a system generally applicable in a wide range of environments. This report describes an attempt to produce a properly engineered communication system which can be easily 'plugged-in' to an existing system to provide communication facilities.

As such, the ideas developed apply to the comparatively new area of computer system development in transportable software techniques. Other relevant work has been described in (3) and (4). Hopefully, we are moving away from the situation where identical systems are programmed anew for each new computer and there is no possibility of transferring well-developed systems from existing computers. In the future, it should be possible to transfer complete systems onto new hardware with the minimum amount of new programming and it is hoped that the system described in this report makes some contribution to the techniques required to achieve this end.

Chapter 2

USER APPLICATIONS FOR SMALL COMPUTER COMMUNICATIONS LINKS

2.1 Introduction

When the development of suitable communications facilities for small computers was first being investigated during 1969, there existed a small number of specific applications which had well-defined objectives for the use of a communications link. There also existed a much larger number of potential applications because of the rapidly-growing number of small computers being used for dedicated purposes within individual departments. The intention was that as a result of studying carefully the requirements of the well-defined applications, a general communications support system could be developed which would also be suitable for the potential future users with their as yet undefined applications.

The The specific applications which existed at the time are described below.

2.2 ERCC PDP-8/L

The proposed use for this computer was to support the Calcomp graph plotter, which was at the time directly attached to the KDF9. The graph plotter service had to be transferred initially to the 360/50 and eventually to the 4-75. Although it was possible to connect the Calcomp plotter directly to the 360/50 and also to the 4-75, this method of connection was expensive, and the controller involved could obviously only be used to support the graph plotter.

Furthermore, the standard software available for the 360/50 did not support the use of a graph plotter as a standard output peripheral, so that special software would have been needed, the expertise for which did not exist within ERCC at that time. It was decided therefore to use a small computer to control the plotter directly, and connect the small computer via a communication link. This solution had several advantages. Firstly, the hardware needed on the 360/50 to control the communications link could also be used for other remote computer links, since the plotter would not be in use all the time. Secondly, since the support for remote computer links was also planned for the 4-75, there would be no special problems involved with the planned transfer of the graph plotter to the 4-75. Thirdly, the standard software for the 360/50 supported the use of remote computer links, so there would be no requirement for special software on the 360/50. The requirement for the PDP-8/L communications system was to support the transmission of the graph plotter files from the 360/50. These files were effectively binary data files specifying XY vectors and pen control commands.

A second service planned for the PDP-8/L was the support of paper-tape input to the 360/50. There was no paper-tape equipment on the 360/50 as there had been on the KDF9, yet there were still many users who produced paper-tape from experiments using data-loggers or small computers. These users were initially catered for by transferring the paper-tape to magnetic tape on the 4-75 and then reading this magnetic tape on the 360/50, where the user could process the data. Since it was a comparatively simple job to interface a paper-tape reader to the PDP-8/L, the paper-tapes could be read directly into the 360/50 if the communication system provided the appropriate facilities. There was no standard format for paper-tape data - some of it was in Edinburgh ISO code, some was in other less common character codes, some was genuine binary data and there was also some 5-channel paper-tape. In order to cater for all these diverse requirements, two facilities would be sufficient. Firstly, support for the locally recognised ISO code, which would be handled as character data, and converted to a form which would be recognised as characters by the 360/50. Secondly, all other data formats would be treated as 8-bit binary data, whether it be binary or not, and the user could then interpret the data according to his own conventions.

The requirements for communications support for the PDP-8/L were therefore character and binary data outwards, and binary data inwards.

2.2 Physics PDP-8

The Physics department had been using a sophisticated PDP-8 installation for some time for the direct control of on-line experiments, and the gathering of data from a variety of analogue equipment. A small time-sharing system had been developed for the computer which supported the use of programs written in a subset of IMP for the control of the experiments(5). The compiler for this IMP subset was itself written in IMP, but was far too big a program to be run on the 8K PDP-8 system. Compilations were therefore carried out on the 360/50 or 4-75, and the resulting binary object programs punched out on paper-tape at ERCC. This process therefore involved long delays for the PDP-8 users at Physics, since their turnaround was limited by the van schedules.

If a communication link could provide a fast and convenient way of accessing this remote compiler, and getting the program listing and object program back quickly into the PDP-8 filing system, program development for the PDP-8 user would be much more convenient.

A second application for the communication link at Physics concerned the data that was being gathered from the experiments and the analogue equipment. The IMP pprograms in the PDP-8 could perform a certain amount of first level data analysis in order to vet the raw data. Any major computational work on the data, however, had to be performed on the 4-75 or 360/50 because of the very limited size and speed of the PDP-8. The data, which was stored on DEC-tape during the experiment, had to be punched out on paper-tape and sent to ERCC for transfer to a magnetic tape before any major analysis could take place. This obviously entailed considerable delays. If the data could be sent directly from the DEC-tape to the 360/50, and the analysis program called up at the same time, these delays would practically disappear since the only holdups then would be the actual transmission time for the data and the natural turnaround delay for executing the program on the 360/50.

Experimental data consisted either of numbers in character format, or numbers in 12-bit PDP-8 binary format, so the communication link should be able to handle both types.

The communication link requirements for this system were therefore character data and binary data inwards and outwards.

A further requirement became apparent in this application on considering the quantities of data accumulated by the experiments. The data transmission speed provided by the communication link would have to be considerably in excess of the speed available for, say, teletype communications. Otherwise, it would take several hours to transmit the data from an average experiment, and the advantage gained from the direct communication link would be nullified. There were several cases where a time in excess of 24 hours would have been required to transmit the data at teletype speed. The changes that all elements of the communications link, including both computers, would continue operating for that length of time were quite low, so it is unlikely that such cases would be handled at all.

It was an essential requirement for this application, therefore, that the data transmission speed should be at least an order of magnitude faster than teletype speed.

It might be argued in connection with this application that it would be even more satisfactory if the data could be transmitted directly as it was being gathered, rather than going through an intermediate stage of storing it on DEC-tape. The feasibility of doing this depends very much on the rate at which data is being gathered, and the reliability of the main computer. A dedicated small computer usually has a much higher probability of staying up through the whole of an experiment than a general-purpose computer which is performing a number of other un-related functions at the same time. Also, there may be upredictable delays involved in servicing the communication link on the main computer because of peak load situations caused by the other activities on the system. Such delays would be unacceptable in a real-time experiment and valuable data might be lost. Because the small computer is working in a dedicated environment, and the data is being output to the relatively fast medium of DEC-tape, an adequate response time can be guaranteed, even if it means running only the one experiment. Although there may be some situations where the data gathering is not time-critical, and the experiment can be repeated if the communication link or remote computer fails in any way, in general it is more convenient to gather the data locally, thereby minimizing the number of different components involved in the real-time situtation.

2.3 Social Medicine PDP-8

The Department of Social Medicine had been using a small PDP-8 system for statistical analysis of survey data. The installation comprised a PDP-8 with 4k of core and teletype, paper-tape reader and punch and a card reader which could read column binary cards. The paper-tape equipment was just used for ease of program development, the main peripheral being the card reader. All the survey data to be analysed was punched onto cards and extensive use was made of so-called column binary cards, in which any combination of the 12 holes in a particular column may be punched, rather than the restricted combinations allowed by certain standard card codes, such as EBCDIC or BCD. The use of column binary can produce a considerable increase in the packing density of data on cards when purely numeric data is being recorded. This feature is obviously useful therefore where a large volume of data is being processed. The use of column binary was also attractive to the PDP-8 users at Social Medicine since the PDP-8 is a 12-bit machine, and it is very easy to store and manipulate each individual card column.

Although the processing of the column binary cards was easily handled on the 12-bit PDP-8, problems arose when larger and more complicated survey programs necessitated the use of the main ERCC machine. Both the 4-75 and the 360/50 are organized around the 8-bit byte unit of storage, and the peripheral equipment is intended to handle 8-bit data characters, implying the use of the restricted EBCDIC card codes for card data. It is possible to have special hardware options fitted to the 4-75 and 360/50 card readers which enable column binary cards to be read by splitting each card column into two 8-bit bytes. The mode of reading, EBCDIC or binary, is selected by software command. However, it is operationally very inconvenient to read jobs in which program and control cards are punched in normal EBCDIC code, and the data is punched in column binary format. In fact, the standard spooling software on both the 4-75 and the 360/50 did not cater for this situation, and the only way to handle column binary was to run a special utility which bypassed the normal spooling software and read the cards directly into core from where they were stored on magnetic tape. The user program could then access the data in a subsequent run. Such special procedures obviously produced considerable delays for any user with columnebinary card data.

In one sense, the problem of handling column binary cards was a historical one, since, if users could be discouraged from producing any new binary cards, then all the binary cards currently in use could be copiëd onto magnetic tape with the special utility and that would be the end of the problem. However, a strong requirement was developing within the University for the use of mark sense cards and documents. The use of these facilities could eliminate the laborious data preparation work involved in transcribing the information from manually prepared forms and documents into a machine-readable form. There was a demand for such facilities in the areas of gathering data for surveys and for examinations involving the use of multiple-choice questions.

Mark sense cards and documents produce data of a very similar form to binary cards, the difference being that there may be less than 80 columns per card. The demand for the use of such facilities, therefore, reinforced the requirement for a convenient way of reading column binary cards as normal program data.

Since there is no sophisticated spooling software being used on a dedicated small machine, the program is reading the cards directly, one at a time, and it is a simple matter to change reading modes by a suitable operator command. The normal program and control cards can be read and translated according to the card code being used and transmitted as character data. The 12-bit column cards can be converted to a suitable 2-byte representation such that each binary card is sent as 160 binary data characters. This format can be easily reconstituted by the program to the original binary card representation if required, using the 16-bit half-word data type in IMP or FORTRAN to store each card column. This facility then means that binary card data can be processed as easily as normal card data on the 360/50 or 4-75.

The requirement for the communication system was simply to be able to transmit character data and binary data and be able to switch between them in the same transmission.

2.4 Computer Science CAD Project PDP-7.

The CAD group in the Department of Computer Science were using a PDP-7 computer to support a number of projects in interactive graphics. They had long recognised the difficulty of writing large, complicated graphics applications programs to run in the PDP-7 itself and were wedded to the philosophy of using a larger, general purpose computer as a backup machine. The PDP-7 would be responsible for managing the low-level control of the display, and handling those things that required a fast real-time response to maintain the interaction, such as pen-tracking. The graphics application program, which actually generated the display file, would run in the powerful backup machine. This program, which frequently involved a considerable amount of numerical work and manipulation of complex data structures could be written almost entirely in a high-level language. This meant that it was much easier to write and test than if it were written in the PDP-7 assembler language, and also that the size of application that could be handled was not limited by the small amount of core storage available on the PDP-7.

This type of two-computer system obviously requires the use of some sort of communication link between the two machines. The speed of this link must be sufficient to avoid long delays in making changes to the picture displayed which require the re-generation and re-transmission of the display file.

The CAD group were already using a two-computer system of this nature, the back-up computer being the Elliott 4130 in the Department of Machine Intelligence. Since the PDP-7 was situated in the next room to the 4130, a high-speed local connection had been established which gave a data rate equivalent to a serial data transmission speed of about 40 kilobauds. This speed was sufficiently high that there were never any delays involved in changing the picture being displayed.

Although this arrangement gave a very satisfactory performance for the display system, there was considerable inconvenience involved in using it. This was caused by the fact that the other main activity for the 4130 was running the multi-access POP-2 system for the Machine Intelligence Department. This was a special-purpose system and was not suitable for running the large graphics application programs, for which the favoured language was FORTRAN which was not compatible with the Multi-POP system. Therefore, pressure for time on the 4130 meant that the dedicated sessions for graphics work were only available during the evening and night.

Because of this, the CAD group were anxious to establish a communication link to a remote large computer which would allow their graphics programs to be time-shared with other programs, thereby avoiding the requirement for the dedicated and inefficient use of an expensive general-purpose computer. This was obviously the sort of application that could eventually be supported by the multi-access operating system being developed for the 4-75, but as this project was still in its early stages, other alternatives had to be considered. The 360/50 was not particularly suitable since the software was oriented to high throughput of batch work, and although time-sharing was supported, the dynamic swapping in and out of programs performing interactive work was not, so the graphics program would have been permanently resident for long periods and making inefficient use of that portion of core store.

The other possibility was to use the 360/67 at Newcastle University which was running the interactive Michigan Terminal System (MTS) for most of the day. This system was certainly suitable for running interactive graphics programs, since it had been used in this way at the University of Michigan(6). Furthermore, a communication link already existed between Newcastle and Edinburgh to support a Remote Job Entry service to ERCC. If appropriate communications support could be provided for both ends, the existing link could be made available to the PDP-7 for part of the day.

Although the existing link between the PDP-7 and the 4130 was equivalent to a data rate of 40 kilobaud, this was generally far in excess of requirements. It was found that a number of the graphics applications would still work satisfactorily with the link speed artificially slowed down to about 2 kilobauds, particularly if certain minor changes were made to the graphics software to minimise the traffic across the link. It would have been prohibitively expensive to provide a link speed of 40 kilobaud between Edinburgh and Newcastle, but 2 kilobaud was perfectly feasible economically, and was the speed that had been used for remote interactive graphics at the University of Michigan.

The communication system requirements for this application were again quite clearly defined. The transmission of text messages was required in both directions to enable the operator of the graphics satellite to send commands to the graphics applications program and receive teletype messages back. Also, it was necessary to transmit the display file generated by the applications program to the small machine. This required the transmission of binary data between the two machines. Information about operator interaction with the display, such as the position of the light-pen at particular times, was also most conveniently transmitted as binary data.

2.5 IBM 1130 in ABRO and Department of Statistics

Both these departments were using the standard IBM 1130 configuration comprising processor, store, card reader, line printer and cartridge disk. The computers were used for small-scale statistical and numerical analysis applications, using the extensive subroutine library and FORTRAN II compiler available for the 1130. They were restricted in the size of problems they could handle because of the small file store (1 megabyte), the slow speed of the 1130 as a computer (6μs core store), and the very slow (100 lpm) line printer. In order to handle larger problems, they needed access to a larger computer and this could be conveniently provided by means of a communications link. By writing the FORTRAN programs carefully, they could be run on both the 1130 and the 360/50 without changes, so that program development could be carried out most conveniently on the 1130 before submitting the program for large-scale processing to the remote machine.

This application was obviously a standard Remote Job Entry system where the communication link merely provided a faster turnaround of jobs. The data transmission requirements were for character data in both directions for job input and output.

A more sophisticated application was planned, however, which involved performing successive phases of a calculation on alternate machines. In other words, a preliminary phase would be conducted on the 1130, possibly being steered by interactive work on the 1130 operators console, at the end of which intermediate results would be stored on the disk. These intermediate results would then be transmitted as data for a program to be run on the 360/50, where the extra power of the larger machine was required. The output from this program might then contain information to be stored back on the 1130 disk for further local processing, and so on.

Although this procedure could be carried out by punching out intermediate results on punched cards for submission in a normal job to the other computer, this would be a very tedious business and would detract considerably from the convenience of carrying out certain phases of the calculation in the controlled, hands-on environment of the 1130. This was really an application which needed a communications link between the two machines.

Since the most convenient way of storing data on the 1130 disk was in the internal number representation format of the 1130, rather than in character format, the transmission of this information between the machines required a binary data transmission capability in addition to the character data transmission required for the normal job input and output.

2.6 Conclusions.

The conclusions that can be drawn from the above descriptions of specific applications about the requirements of a communication system are as follows:-

a) the applications involve a number of completely different small computers, so the communication system should not use the special features of any one small computer, but rather should use only a minimum subset of features which are common to all small computers, and this applies to hardware and software.

b) all the applications involve a completely different environment in respect of usage, supporting peripherals and executive, even when the same computer is used, so the communication system should use the executive to the absolute minimum, if at all, and should provide an interface to the rest of the software which makes no assumptions about the local source or destination of transmitted data.

c) the transmission speed required is at least an order of magnitude greater than teletype speed, and since in some applications, speed is critical, it should be possible to buy more speed without affecting the communication system; it is also desirable to maximize effective throughput, since in some cases large volumes of data are involved.

d) suitable communications hardware is available for some of the computers but not for others, so that although hardware will need to be developed as part of the communication system, the software component of the system should not be oriented to that particular hardware implementation, but should be capable of using any implementation available for the type of communication chosen.

e) the system should be capable of being used to communicate with different main computers in order to accommodate future changes in the central computer system available.

f) the system should be capable of handling two basic data types - text data, where the bit patterns are interpreted according to some universally agreed code such as ISO or EBCDIC and where conversion between different representations may be necessary in order to preserve the textual meaning of the data; binary data, where the bit patterns have no universal interpretation and have meaning only to the user program processing the data. In this latter case, the data must be transmitted by the system

Chapter 3

GENERAL COMMUNICATIONS HARDWARE CONSIDERATIONS

3.1 Main features of data communication hardware

Before going on to discuss the various communications facilities supported by the large computers, a brief description of the main features of serial data communication links is given here in order to support some of the technical arguments used later.

Attachment of local peripherals

Information transfer between a computer and its local peripheral devices normally makes use of high-speed parallel data interfaces. These parallel interfaces provide lines for at least one character together with lines for error checking and control of the data transfer. A simple example is the British Standard Interface (BSI) for parallel data transfer. Electrical signals are propagated along these interfaces by fairly low DC voltages, e.g. 5 or 6 volts, that correspond with the low voltages used in the computer electronics. Also, signal duration is very short in order not to slow down the execution of input/output instructions in the computer.

The influence of electrical interference, resistance losses and propagation delays on these low-magnitude, short-duration signals means that expensive high-quality cables have to be used in order to guarantee reliable data transfer. These factors mean that this type of interface - high-speed parallel - is only feasible for peripherals which are close to the computer. In fact, a maximum cable length of up to 100 feet is common.

Attachment of Remote Peripherals

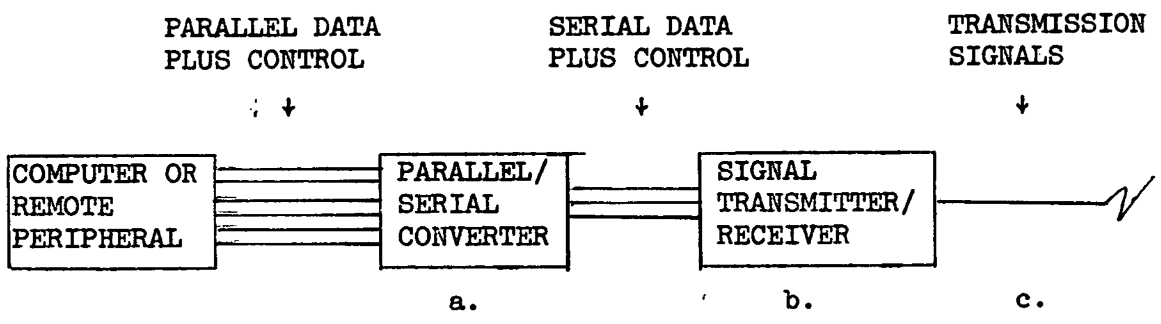
Remote peripherals may be situated several miles from the computer, so that even if the electrical transmission problems were solved by using special hardware, the cost of multi-way parallel cables would be very high. In fact, because of the costs involved, the simplest possible electrical interface is used for the connection of remote peripherals. This consists of one circuit for data into the computer and one circuit for data out of the computer. One circuit is sometimes used for both functions. The signals which are transmitted simultaneously in a parallel interface to a local peripheral are sent in serial form, one after the other at a fixed rate, along a single transmission line to a remote peripheral.

The electrical connection between the computer and the remote peripheral thus consists of the following components:-

a) hardware to convert the parallel information from the computer into serial form for output and vice versa for input.

b) hardware to convert the serial information at the low DC voltage levels into a form suitable for reliable transmission over several miles of cable.

c) single or double circuit electrical cable for one way at a time (half-duplex) or both directions simultaneously (full-duplex) data transmission.


PARALLEL DATA PLUS CONTROL
SERIAL DATA PLUS CONTROL
TRANSMISSION SIGNALS
COMPUTER OR REMOTE PERIPHERAL
PARALLEL/ SERIAL CONVERTER
SIGNAL TRANSMITTER/ RECEIVER
a.
b.
c.

It is the variations in the methods of implementing these three components that provide the different systems of serial data communications in use today.

There are basically two different implementations of item a. which are referred to as synchronous and asynchronous communications adapters. There are a number of implementations of items b. and c. which differ in the speed of reliable data transmission which they give, and these implementations are closely related to the choice of a synchronous or asynchronous transmission adapter. Economic considerations play a large part in deciding which type of system to use. The requirement for higher speed and/or higher reliability usually increases the cost of the system.

3.2 Asynchronous communications adapter

In asynchronous communications (Fig 3.1), each character of a message is identified by one start bit and one or more stop bits framing the character. Each character is therefore self-identifying and there is no fixed time relation between successive characters in a message. The start bit is always of opposite polarity to the quiescent state of the data link since its purpose is to signal the arrival of a new character. On recognising the start bit the

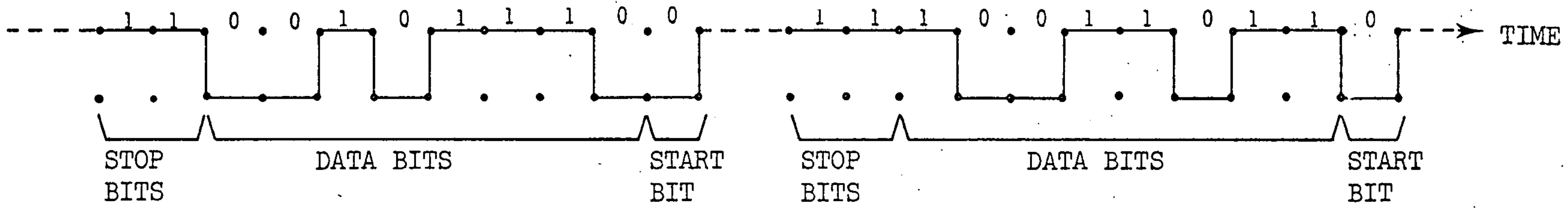


FIG 3.1 TWO CHARACTERS TRANSMITTED ASYNCHRONOUSLY

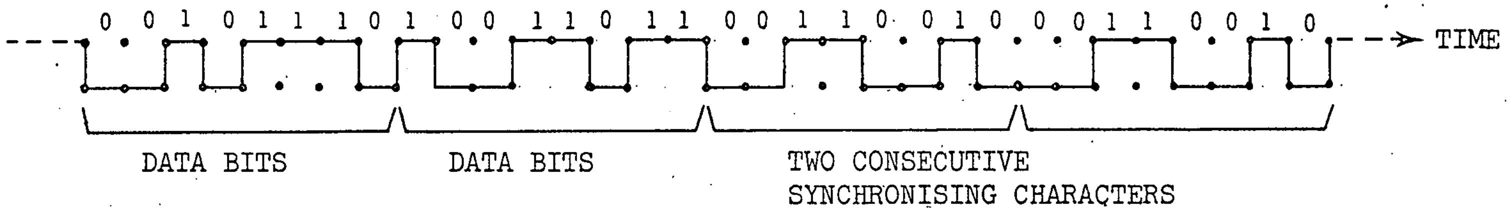


FIG 3.2 SAME TWO CHARACTERS TRANSMITTED SYNCHRONOUSLY

receiving hardware has to start its bit clock, which runs at a fixed rate, and count in the requisite number of bits following to form the character, which can then be transferred in parallel form to the computer. The purpose of the stop bits, which are always of opposite polarity to the start bit, is to return the line to the quiescent state so that the start bit of the next character can be correctly recognised. Asynchronous communication is therefore essentially one-character-at-a- time message transmission, and is usually used to support the simplest kinds of data transmission system, such as typewriter terminals.

3.3 Synchronous communications adapter

In synchronous communications (Fig 3.2), a message is sent as a continuous stream of characters with no interval between the last bit of one character and the first bit of the following character. A message is therefore a contiguous series of bits, and in order to correctly identify the individual characters within this bit stream, the message is preceded by a number of characters of a particular non-repeating bit pattern. The receiving hardware has to recognise this bit pattern in order to lock in to the correct character frame. Once this character phase has been established, the receiving hardware has to count off the required number of bits to form each character using an appropriate clock. Because of the requirement to maintain accurate bit timing over a long message and the inherent difficulty of maintaining two independent clocks in synchronism with each other, the clocks used to provide bit timing in synchronous systems are more sophisticated than those used in asynchronous systems, where the bit clock only needs to be accurate enough to maintain bit timing for one character. The synchronous receive clock has to be self-regulating in order to follow any variations in the transmit clock at the other end of the transmission link. Allowance also has to be made for apparent timing shifts caused by distortion of the bit stream by the dynamic electrical characteristics of the transmission link.

Synchronous communications is therefore essentially a block-oriented transmission system and all the characters in a message have to be present before message transmission can be started. Consequently, complete message buffering has to be provided in a terminal using synchronous communications, and, generally, sufficient levels of character buffering have to be provided to ensure that the input and output shift registers can be emptied or refreshed within the crisis time, which is always one bit time.

Comparison between asynchronous and synchronous communications adapters

The hardware necessary to implement synchronous communications is more complicated than that for asynchronous communications in the following ways:

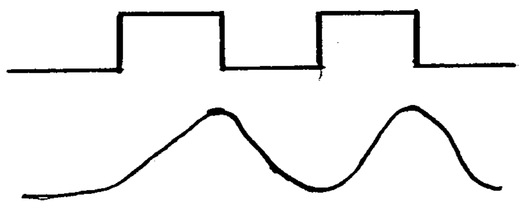
a) recognition of character synchronization pattern
b) provision of self-regulating receive bit clock
c) extra character and message buffering

Synchronous communications has the advantage that a given message can usually be transmitted more efficiently than with asynchronous communications. This is a consequence of the high redundancy involved in asynchronous communications with a minimum of 2 bits extra per character compared with the fixed overhead of the leading synchronization characters. All but the shortest messages will be transmitted more efficiently in synchronous mode.

3.4 Telegraphic signalling

A major factor affecting the economics is the equipment used for item b. This is the equipment which accepts the low voltage (about 6 volts) DC serial data and turns it into a form suitable for transmission over long distances.

The simplest implementation of this merely converts the low voltage into a much higher DC voltage (about 80 volts) which is then capable of being recognised correctly after a few miles of cable. This works on the simple principle of telegraphy and is only capable of supporting fairly low speeds (up to about 200 bauds) over a few miles of cable. The effect of the capacitance and resistance of a long length of cable is to convert the original well-shaped square waves into something like a saw-tooth shape as follows:-



If the original DC voltage is not maintained for a long enough period, the signal level at the receiving end does not reach a large enough value because of the long rise time of the pulse. Therefore, in order to ensure that the serial data can be properly reconstructed at the receiving end, the rate of change of the data being fed into the transmitted end is limited accordingly. This is the factor which limits the speed of operation with this type of signal transmission. The maximum speed is governed by the rise time of the circuit, which is dependent on its length. For circuits up to a few miles, this limits the speed to about 200 bauds. For longer circuits, signal repeaters have to be included in the circuit which re-shape the waveform before transmitting it further. To achieve higher speeds, the signal repeaters would have to be placed at shorter distances along the circuit, which would nullify the economic advantages of the very simple equipment (basically just electrical relays) needed to transmit data in this way. This type of equipment then is suitable only for the lowest data rates.

3.5 Modems

All other equipments in use for signal transmission use some kind of AC transmission which uses the DC voltage levels to modulate a carrier signal of a particular frequency. AC signals are more easily transmitted over long distances because of the electrical impedance characteristics of long circuits. The increasing sophistication of the modulation techniques used is giving higher data rates from the same circuits. The equipment at the receiving end has to remove the basic carrier signal (demodulation) in order to reconstruct the original serial data streams. The equipment used at both ends to transmit and receive is therefore known as a modulator-demodulator or modem.

One of the techniques used to obtain higher speeds is to combine two or more data bits into a new multiple-value signal with which to to modulate the carrier (7) . Thus, in a typical system (GPO Modem 7), one of four phase shifts is applied to the carrier in order to transmit two bits of data. The increased sophistication of the electronics required to accurately encode and decode these multiple value signals, allowing for the unavoidable distortion caused by the transmission line, means that higher data rates will cost more.

Distinction between synchronous and asynchronous modems

There is an important relation between modem technology and the use of synchronous or asynchronous transmission adapters.

As mentioned previously, asynchronous transmission adapters usually provide their own bit timing and can use a simple fixed rate oscillator which only needs to be accurate enough for a single character. The modem used for asynchronous transmission does not need to know the rate at which data is being transmitted or received. It is transparent to the actual bit stream and its function is merely to transmit or receive a two-valued DC signal which can change value at an unspecified rate, up to some maximum which is the limit for reliable transmission. The terminal equipment must be aware of the maximum reliable rate, but otherwise the speed can be varied by setting clocks in the customer equipment inboard of the modem at both ends. No timing information is exchanged between the equipment and the modem.

The bit timing used in synchronous transmission is more sophisticated since it must maintain correct bit synchronism over a whole message, which may be many thousand bits long. Although the bit timing circuitry may be in either the terminal equipment or the modem, it is normally included in the modem since the modem designer is better able to assess the effects of particular types of line distortion on the bit stream and therefore compensate for them in the bit timing circuitry. When the bit timing is incorporated in the modem, timing information is passed from the modem to the communications adapter to tell it at what rate to transmit or receive data.

The distinction between asynchronous and synchronous communications, which is normally made at the character level, can then also be made at the bit level and modem level. Modems can be divided into asynchronous and synchronous classes, the latter class providing bit timing information to the user equipment and the former class not, in which case the user equipment provides its own timing. Because of this, asynchronous modems should be cheaper than synchronous ones.

The consequence of this distinction is that asynchronous modems are restricted to two-level modulation techniques and cannot take advantage of the multi-level modulation techniques which are being used to give increased transmission speeds. The multi-level coding can only be used where the modem controls the bit timing. Therefore, only synchronous modems can be used where higher transmission speeds are required. The maximum available speed for asynchronous modems in this country is 1200 baud (up to 1800 baud in the States), whereas speeds of 9600 baud can be obtained over equivalent circuits using synchronous modems.

It is, of course, possible to use asynchronous start/stop character framing with a synchronous modem since the modem is not aware of the character structure. This, however, would require modified or completely new asynchronous communications adapters, able to accept an external timing signal from the modem, which is a feature not normally available with present asynchronous equipment.

Modems available

Currently, modems offered by the GPO (who have a monopoly over modems using the dial-up transmission facilities) will support dial-up operation at up to 1200 bauds (Modems 1 and 2) with bit timing provided by the customer equipment and private line operations at 2400 bauds (Modem 7) with bit timing provided by the modem. They have also recently announced a facility for dial-up operations at 2400 bauds (Moden 7C) provided the circuit conditions are favourable, i.e. they do not guarantee that the speed can be obtained from all exchanges and lines currently in service. Using the same private line that the GPO uses at 2400 bauds, it is possible to use more expensive proprietary modems which will operate at up to 9600 bauds, but this becomes very expensive. These speeds are adequate for all the applications considered in this report.

The cost of these modems are such that there is a steady increase in price from the Modem 1 to the Modem 7C, with the high-speed proprietary modems disproportionately more expensive.

3.6 Electrical circuit facilities

All variations of item c. in the diagram are provided by the GPO since they have a monopoly of public telecommunications facilities in this country. The situation of in-house communications which can make use of non-GPO facilities will not be considered in this report since this can take advantage of specially-laid high-quality circuits, such as co-axial cables, which are not generally applicable to the telecommunications problems considered in this report.

The line facilities provided by the GPO fall into two categories - dial-up facilities available through the normal voice network and private lines which use telephone circuits but which are permanently allocated and are effectively hard-wired between the two ends of the link. The former facility provides a data link of variable quality, since the circuit is routed through mechanical switching equipment and is likely to use different pairs of wires each time a call is made. The variability of the voice telephone network in terms of reliability and quality of connection, especially over long distances, is well known. A private line, because it always uses the same wires and is not routed through switching equipment will give a constant quality and the reliability is obviously very high, leaving aside accidental interference by GPO maintenance personnel. Because the circuit used for a private line is always the same, the GPO can guarantee the quality of the line in terms of its electrical transmission characteristics. The GPO offer a number of Tariffs, giving different guaranteed electrical characteristics to cover a range of possible transmission speeds. The simplest is Tariff J which is intended for use with slow-speed Telex-type terminals. The best is Tariff T which will support speeds from 2400 up to 9600 bauds using special transmission techniques. The GPO will provide special facilities for operation at 48k bauds but this is not a service normally available and will not be considered further since the speed range covered by the normal Tariffs is sufficient for all the applications considered in this report.

The economic considerations are as follows.

For dial-up connections, the normal STD charges apply which means that the cost depends on the distance, the time of the connection and the period of connection. For private lines, a fixed rental applies which is determined by the Tariff and the point-to-point distance and is independent of the amount of use made of the circuit.

The choice between the two facilities is determined by the speed required and the amount of connection time required. The latest GPO facilities provide for up to 2400 bauds using dial-up connections.

3.7 Conclusions

The basic elements of data communications hardware have been described in this chapter and some general observations can be made.

The cheapest and simplest communication system for lower speed applications (up to 1200 baud) would use an asynchronous communications adapter and an asynchronous modem with a dial-up or private line of a suitable Tariff depending on the speed and the amount of connection time required.

This system, however, could not be used in an application requiring speeds higher than 1200 baud unless the asynchronous adapter was equipped with an external timing option, enabling use of the higher-speed synchronous modems. Note that this also requires that the other end of the link has the same capability, since compatible modems must be used at both ends of the link.

For these higher speed applications, the use of a synchronous communications adapter should be considered. This involves more complicated hardware (although the extra cost becomes less significant when compared with the increased cost of the higher-speed modems) but gives a higher useful data rate than asynchronous over the same speed line.

Chapter 4

SYNCHRONOUS V. ASYNCHRONOUS FOR COMPUTER-COMPUTER COMMUNICATIONS

4.1 Introduction

All the major large computers support the use of serial data communication links for the attachment of remote peripherals. All of them support the use of both asynchronous and synchronous transmission so the first choice to be made in deciding how to communicate with them involves which of these two systems to use.

As mentioned in the previous chapter, asynchronous communications is suitable for applications involving the transmission of messages where there is no time relation between successive characters in the message. This makes it suitable for the connection of typewriter terminals, where the rate of data input by the human operator can be extremely variable.

Synchronous communications is inherently a block-oriented transmission system, where the successive characters in a message must be transmitted in a fixed time sequence. This necessitates the use of terminals with suitably-sized message buffers.

When the overall system implications of handling terminals are considered, the requirement for reliable and error-free transmission means that communication between computers must operate in a block-by-block mode. This distinguishes it from applications where human operators are the source and receptor of all transmitted messages.

This distinction is demonstrated by the argument below.

4.2 Handling of transmission errors

In computer-to-computer communications, the responsibility for the correctness of the data transmission no longer lies with a human operator. If a human operator is an inherent part of the data transmission system, as is the case with typewriter terminals and alphanumeric video display terminals, this operator can monitor the correctness of the data being transmitted and can initiate intelligent action to recover from any transmission errors. The computer end of the link need perform very little recovery from any errors it detects, other than ignoring the last message and sending a message to the operator to inform him of the error. It is then up to the operator to send the message again if necessary. This aspect of human-operated terminals makes them particularly easy to handle from the point of view of the computer at the other end of the link.

When the data transmission is from one computer to another, then it is essential for the computers to monitor the correctness of the data transfer for themselves. Otherwise it would be necessary for the data to be displayed for vetting and approval by an operator, which would seriously reduce the effective data transmission rate. Furthermore, since some of the data transferred may be binary data rather than character data, it is not always possible to display it in a form which can be intelligently checked by an operator.

It is also desirable if the computers can themselves initiate recovery action after detecting an error, since this gives much faster recovery response from transient error conditions, which are the usual source of errors on data transmission lines.

This again points out a difference between peripherals connected locally and those connected remotely. Any errors which occur during data transfer over a local high-speed parallel interface are considered to be fairly serious, and recovery usually requires manual intervention. Errors during data transfer to a remote peripheral are to be expected and only merit investigation if their frequency noticiably degrades the link performance.

Automatic recovery from transmission errors is therefore an important part of computer-to-computer communications. If automatic recovery is to be effective, then it is important to have very effective error detection. Errors are detected by adding redundant information, such as a parity bit, to a character or to a whole message, which is then checked on reception.

4.3 Error detection

The exact nature of data transmission link reliability in terms of susceptibility to corruption of the data is obviously highly dependent on the particular facilities used, e.g. dial-up or private line, transmission speed, the precise locality of the wires used and its routing, etc. However, a great deal of investigation has been done on this subject by a number of common carriers in different countries and it is possible to draw certain general conclusions.

The most important conclusion is that while constant factors such as thermal noise cause signal distortion, suitable modem design can help to compensate for this and most data errors are caused by 'impulse noise' which results from the nature of the switching equipment used in most exchanges. Even if a private line is used, which does not involve any switching equipment, such lines are normally ordinary telephone circuits which have been specially wired point-to-point. As such they share the same cable runs as dialled circuits, and are liable to pick up impulse noise generated in adjacent circuits by dialling, etc.

Impulse noise produces a short burst of electrical interference sufficiently large to completely swamp the transmitted data signal and replace the effected bits by a random pattern. The duration of a noise pulse can be anything up to 1/100th of a second, so it is clear that several bits in a message at 2000 bauds would be affected. Experiments carried out on a long private line in Europe at 2000 bauds (9) showed that over 60% of messages in error had 2 adjacent bits corrupted and 30% had up to 8 adjacent bits in error.

The simple conclusion to be drawn from this is that simple character parity does not give very good protection against burst errors, since two or more adjacent bits in error have an even chance of producing a bit pattern with the same parity as the corrupted bits. Therefore, message transmission which relies on simple character parity is not well protected against localized errors, since the redundancy is also localized and the effect of an error is not propagated throughout the message.

4.4 Block-oriented error detection

To deal with burst errors effectively, a form of non-localized redundancy is required which is accumulated over a whole message and which will propagate the effect of a local error through to the end of the message. This type of redundancy is then a redundancy check on the message as a whole rather than on individual characters, and examples of this are lougitudinal parity and cyclic redundancy check(10).

This consideration forces computer-to-computer communication to become message-oriented or block-oriented rather than character-oriented. Long messages have to be broken down into smaller blocks for transmission purposes, and the optimum block length is determined by the mean error rate and practical considerations of buffer size within the computers. Because of this requirement for block-oriented transmission between computers, with error detection and recovery being done on a block basis, computer-to-computer communications cannot take advantage of the inherent simplicity of using asynchronous mode as in character-oriented transmission.

For reliable and error-free transmission between computers, a rigorous scheme must be implemented to achieve automatic control of the transmission link, and the system complexity of an asynchronous link handler becomes equivalent to that for a synchronous link.

The choice between the two systems is then based on the economic and hardware complexity considerations described in the previous chapter.

4.5 Comparisons

The previous chapter indicated that the cheapest communication link was provided by a totally asynchronous system, i.e. asynchronous transmission adapter and asynchronous modem. This however had limitations if it was required to operate the link at speeds greater than 1200 baud.

This gives a maximum character rate of only 120 characters per second. This would not be adequate for a number of the applications considered in this report. To take some concrete examples, peripherals commonly used on the small computers have the following speeds:-

card reader - 300 cards per minute

line printer - 300 lines per minute

paper-tape reader - 300 characters per second.

Assuming that trailing blanks are not transmitted and that only about 40 leading characters per card or line are actually used for information, these peripherals generate a data rate of 200-300 characters per second. Other peripherals, such as DEC-tape or a disk, could generate an even higher rate.

Such applications are therefore not well matched to the cheapest facilities and it becomes necessary to use the higher speed synchronous modems. Assuming that the problem of interfacing an asynchronous transmission adapter to a synchronous modem at both ends of the link is amenable to a solution, which is not always so, the use of asynchronous transmission would produce a lower line utilisation than synchronous transmission by at least 25% with the minimum of 2 framing bits per character.

In these circumstances, it seems more appropriate to opt for a synchronous transmission scheme, which will make more efficient use of the higher speed modems.

Once the required data rates justify the extra cost of the higher speed modems, which can be considerable, it is important to take full advantage of the higher speeds provided. The extra cost and complexity of the synchronous communications hardware becomes less significant as the modem becomes the most expensive component in the system.

4.6 Conclusions

From the above arguments, the advantages of asynchronous communications are its low cost and relative simplicity when used with human-operated terminals. Since such terminals are used in large numbers, it is important to use the cheapest possible facilities, provided these are adequate. For higher-speed communications however, suitable for inter-computer working, the synchronous system has the advantage of making best use of the more expensive facilities needed.

Chapter 5

LARGE COMPUTER COMMUNICATIONS HARDWARE

5.1 Introduction

As has been stated previously, all the large computers considered supported the use of synchronous communications. Those considered were IBM 360, ICL System 4, Univac 1108, CDC 6600 and Burroughs B5500. The way in which these computers supported synchronous operation was then investigated in more detail to see if there was a compatible way of communicating with them all. The list above is in order of the amount of information available on detailed operation of synchronous hardware.

The first thing to become apparent was that the communications codes and protocols were implemented in very inflexible ways on most of the computers. The communications facilities were controlled very largely by hardware with very little software control. Details of control characters and their interpretation, message formats, error checking, etc., were implemented in such a way that they could not be changed by a user program.

5.2 Communications controllers

The implementations of the communications controllers varied from pure hardware on the IBM 360, through hard micro-program on the System 4 to small, special-purpose programmable processors on the CDC 6600 and Burroughs B5500. However, even on those systems which were genuinely programmable, it was not intended that the control programs should be accessible to the user for him to implement his own communications protocols. This meant that all communications controllers were effectively hard-wired to the manufacturer's defined codes and protocols.

The only large computer to control the communication line directly by software in the main processor was the Univac 1108. This had a very simple communications controller which generated an interrupt on a per character basis. However, for reasons of efficiency and crisis time limitations, in order to reduce the overhead associated with handling these interrupts, they were handled at a very intimate level within the Executive rather than being routed through to a user program for analysis. Thus, it was again very difficult for the user to implement other codes and protocols.

This inflexibility in the large computer communications facilities would have to be accommodated for in the facilities provided for the small computers which will be described in later chapters.

It is perhaps worth making the general point that the peripherals on a large computer have a much higher degree of hardware control than on a small computer. This is evidenced by the fact that, on small computers, many peripherals operate by transferring one character at a time under interrupt control. Because of the simpler software in use on a small computer, the overheads involved with handling interrupts, such as register-saving and status-switching, are much less than on a large computer, where a considerable amount of CPU time would be consumed by handling peripherals in this way. Peripheral transfer speeds for equivalent peripherals are also usually much higher on a large computer, so the interrupt rate would be considerable. Peripherals on a large computer are normally handled on an autonomous block basis, with just one interrupt at the end of the block. It is therefore in line with the general philosophy of peripheral control on large computers to handle communication lines on a block-transfer basis, which requires that most of the protocol be defined by the communications controller.

The sensible solution to this problem of inflexibility is to use a programmable communications controller and make it easy for the user to program it himself. Both IBM, with the 3705 (11), and ICL, with the 7905 (12), are now adopting this approach and are providing a proper user programming system for the communications processor. This will, in the future, give the user the freedom to implement different protocols if he wishes.

However, for the present, these inflexibilities exist which result in certain incompatibilities between the different large computers in the synchronous communications environment. The particulars of the IBM 360 2701 Synchronous Data Adapters and the ICL 4-75 MCCCU synchronous buffers will be given here by way of example.

5.3 IBM synchronous controllers

IBM 2701 SDA I supports a communications protocol known as STR (Synchronous Transmit/Receive). This is based on a special-purpose code known as 4-out-of-8 code, in which only those characters which have 4 bits set out of 8 are valid. This gives a code set of 72 characters of which 8 are used as control characters, leaving a usable character set of 64. This is a completely non-standard code used only for data transmission purposes and, as such, bears no similarity to any of the character codes in common use on other peripherals, or on other computers.

IBM 2701 SDA II supports a protocol known as BSC (Binary Synchronous Communications) which can use one of three transmission codes. These are EBCDIC (8 bits per character), ISO (8 bits per character) and Six-Bit-Transcode, or SBT, (6 bits per character). In EBCDIC, all 8 bits are available for data, giving 256 code values of which a small number are reserved for transmission control characters which are recognized by the hardware. The EBCDIC code also provides a special mode of operation known as transparent mode, in which, by using a particular 'escape' sequence, it is possible to transmit any 8-bit code including the transmission control codes. The ISO code uses 7 bits plus a parity bit, and SBT uses all 6 bits as data. Again in these two codes, a small number of codes are reserved for transmission control purposes. It is not possible to switch between these codes by software control. The options have to be wired in to the 2701. There is a special additional 2701 feature which allows any two of the codes to be wired in with a software-controlled selection between the two. There is otherwise very little software control over the transmission facilities.

5.4 ICL synchronous controller

ICL MCCCU is a general-purpose communications multiplexor for both synchronous and asynchronous communications. The synchronous buffer facility allows a software-controlled choice between EBCDIC (8 bits plus parity) and ISO (7 bits plus parity). The EBCDIC code, requiring 9 bits per character, is obviously incompatible with the IBM EBCDIC code, and also does not support the transparency feature. The ISO code is also not compatible with the IBM code as different values are used for the transmission control characters, such as SYN.

5.5 Problem of compatibility

Thus, there would appear to be no compatible communications facility between the IBM 360/50 and the ICL 4-75, which were the computers of most relevance in Edinburgh. However, there was a certain amount of compatibility between the two systems at one level in that they both supported the same general type of point-to-point communications protocol.

This type of protocol, which will be explained in more detail later, allows communication in one direction at a time. Once one end has acquired control of the line, it can continue to transmit data blocks until it relinquishes control of the line by sending an end-of-transmission control sequence. The receiving end has to transmit acknowledgement sequences to the data blocks it receives.

At this functional level of the IBM and ICL transmission systems, there was compatibility. However, the two systems were different at the detailed level of message format, acknowledgement format, etc. This general type of protocol was also acceptable to the UNIVAC 1108, but information on the CDC and Burroughs protocols was not available.

5.6 Conclusions

Therefore, any attempt to communicate with the two computers in a compatible way must be based on this general type of protocol. The differences in details of implementation must be accommodated by the small computer software, and the small computer hardware must be sufficiently flexible to allow the software the necessary level of detailed control. Both these aspects will be described in more detail in later chapters.

Chapter 6

INTERFACING TO THE SMALL COMPUTER

6.1 Introduction

Previous chapters have considered the general aspects of communications hardware and particular implementations on present-day large computers. It is now necessary to consider the provision of suitable communications hardware for the small computer end of the link.

In some cases, the small computer already has suitable communications hardware and this will not be dealt with until the chapters on communications software, where it will be shown how any suitable hardware can be used with the standard software. In other cases, notably the PDP-8 which was the most common small computer in use at the time, there was no suitable hardware and it was necessary to develop it.

The description of this development is conveniently split between two chapters. The present chapter is concerned with the way the communications hardware should be interfaced to the small computer. The next chapter describes the development of a functional specification which is considered to give an optimum division of responsibility between communication hardware and software.

6.2 The 'uniform interface' requirement

The functional specification merely defines the programming characteristics of the communications interface. In order to achieve the desired objective of developing communications hardware that would be suitable for use on any of the small computers considered in this report, it is necessary to study the characteristics of the small computer input/output interfaces to see if the functional specification envisaged can be supported in a compatible way on the different computers.

The specification can obviously be supported on each small computer by implementing it in the way most suited to the particular input/output interface, but this would necessitate a significant amount of re-design effort for each different computer as there are considerable differences in the way input/output devices are supported, for example, in the way peripherals are addressed.

The following discussion compares the different input/output interfaces and arrives at a minimum subset of the facilities provided which have equivalents on all the computers. The precise electrical characteristics of the interface are obviously going to be different between the various small computers in terms of signal levels, pulse lengths, voltage or current driven signals, etc. Such differences can be accommodated by a first level of signal buffering between the communications interface and the computer in which the signal levels from the computer are converted into the signal levels acceptable to the communications interface logic. Such a signal buffering in a simpler form is a standard feature of some input/output devices where logic signal levels are not normally suitable for directly driving long lengths of interface cable because of the current drain involved.

6.3 Small computer input/output interfaces

The facilities provided by small computers for handling input/output devices can be grouped under the following headings:-

C1. program selection of a particular device, i.e. addressing capability

C2. program commands issued to the device so addressed

C3. capability for device to input and output the data

C4. capability for device to demand servicing by program, i.e. interrupts.

All the small computers considered in this report support these capabilities, but the ways in which they are implemented differ considerably. The input/output interfaces of the different small computers were investigated in some detail and a summary of the interfaces of four of the computers is given below, in terms of the capabilities listed above. These four computers are described because they represent four completely different methods of input/output control, but they are all capable of supporting in a common way a minimum subset of input/output facilities which is sufficient to support the communications interface envisaged. These facilities are sufficient to support any simple input/output device which has a data rate that can conveniently be handled one character at a time under program control. Faster data rate devices which require some means of autonomous data transfer are not considered, since the requirement for flexibility in the protocol used means that the software must control data transfers one character at a time.

The four computers considered are the DEC PDP-8, the CTL Modular One, the Elliott 4100 and the IBM 1130.

6.4 PDP-8

C1, C2. An input/output instruction on a PDP-8 has the following format:

6	Device Address	Command

The number '6' is the PDP-8 instruction code for an input/output instruction. The device address field is 6 bits and the command field is 3 bits. When this instruction is executed, the device address field is gated in parallel onto an I/O bus and the 3 bits of the command field are gated out on separate lines one after another at fixed intervals while the device address is held steady. All devices on the system are connected to the I/O bus and it is the responsibility of a particular device to recognize its own address and interpret the signals on the command lines appropriately. This sequence lasts for a fixed time and there are no return signals to indicate whether or not any peripheral device responded to the command.

Because there are no strobe signals accompanying the pulses on the command lines, it is difficult to detect the absence of a pulse on a particular line. It is only sensible to use the 'on' condition of a command pulse to initiate any action, and since the pulses are not gated in parallel, it is also not sensible to try to interpret combinations of 'on' pulses to initiate distinct actions.

It is only sensible to use multiple command bit settings to initiate the same actions in one instruction that would be initiated separately if the command bits were issued singly in separate instructions. This means that only three distinct commands can be associated with any particular device address.

If a device needs more than three commands, and most of them do, then the technique used is to allocate multiple addresses to the device. Since all device addresses broadcast on the I/O bus are available to all devices on the bus, it is perfectly feasible for one device to recognize more than one address and interpret the command lines differently depending on the address recognized. Since there are 64 possible device addresses, the allocation of multiple addresses to one device does not normally restrict the number of separate devices that can be attached to the I/O bus.

C3. The method of achieving single character input/output on the PDP-8 is a standard part of the basic input/output cycle.

When the input/output instruction is being executed, the contents of the main accumulator (AC) are gated onto 12 parallel lines of the I/O bus and are held there for the duration of the cycle. Then, if the particular device addressed wishes to interpret one of the commands as an output command, it can gate the AC lines into an internal register. Similarly for input purposes, there are 12 AC IN lines in the I/O bus, which can be set by the device, and suitable control signals to tell the PDP-8 CPU to gate these lines into the AC.

All these functions are controlled by the device, and the CPU does not place a particular significance on any of the commands. The device can use any of the command pulses with any address to signal input or output of a 12-bit wide data word.

C4. A further signal line present on the I/O bus for use by the devices is the INTERRUPT line. Any device can request an interrupt by holding this line to the appropriate 'ON' condition. Since there is only one interrupt signal for all devices, the program has to interrogate each device on the bus to determine which device requested the interrupt.

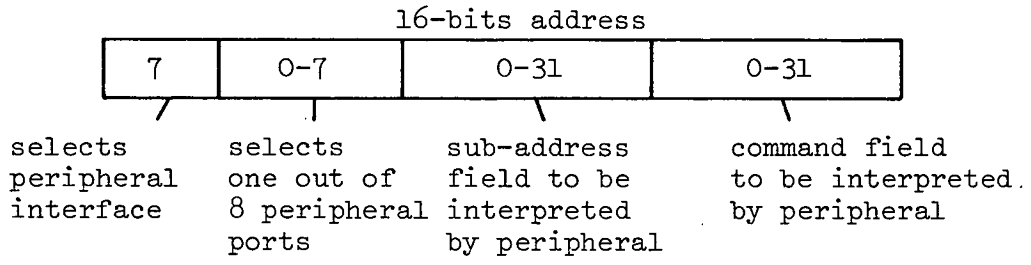
The I/O bus provides a SKIP line for this purpose. If a particular device has requested an INTERRUPT then it should set the SKIP line to the 'ON' condition when it recognizes a particular command on its address. The SKIP line being set causes the CPU to step the program counter by one, thereby providing the facility for a conditional branch following the input/output instruction.

Again if a device wishes to use more than one distinct interrupt, it can use different combinations of address and command to select the correct interrupt. There is obviously no practical limitation on the number of distinct interrupts that can be used.

The proposed functional specification could clearly be implemented on a PDP-8 by using multiple device addresses to accommodate all the commands and error flags.

6.5 Modular One

C1,C2. The Modular One uses an unusual method of addressing peripherals, in that there is no actual input/output instruction in the CPU. The addressing range of the Modular One is from 0 to 64k, and the convention adopted is that any attempt to reference a store address above 56K is routed to the peripheral interface, and the address so generated is then interpreted as follows:-


16-bits address
7
0-7
0-31
0-31
selects peripheral interface
selects one out of 8 peripheral ports
sub-address field to be interpreted by peripheral
command field to be interpreted by peripheral

Therefore the CPU uses 6 of the 16-bits to select a particular peripheral channel and the remaining 10 bits can be interpreted by the peripheral device on the channel. It is common practice to use 5 of these bits for further addressing if necessary and 5 bits to specify the command, although there is no requirement to do so. Since these fields are strobed out to the peripheral in parallel, any combination of the 5 bits can be used to give a total of 32 possible commands, of which the first 5 are used for standard functions common to all peripherals, such as testing operability. This command and addressing structure obviously gives plenty of scope for a peripheral to handle a wide range of different commands.

C3. All references to a peripheral are conducted using a standardized 3-word handshaking exchange format. The three words involved are referred to as Control Word, Slave Word and Master Word respectively.

When the CPU executes an instruction which generates a store reference in the peripheral address range, the address so generated constitutes the Control Word, and this is sent out to the selected peripheral channel. The peripheral must examine the least significant 10 bits of the Control Word and interpret them according to its particular command repertoire. In response to the Control Word, the peripheral must return a signal, saying whether it accepts or rejects the command, together with a 16-bit Slave Word, which has no pre-defined format. If the peripheral indicates acceptance of the command, then the CPU will send out a 16-bit Master Word, again of no pre-defined format, for interpretation by the peripheral. This completes the 3-word handshake sequence.

Depending on the actual instruction executed by the program, the Slave Word can be loaded into any one of the programmable CPU registers and the Master Word can be taken from any one of these registers. This is then the method of performing single word data input and output to the peripheral under program control.

C4. A general-purpose INTERRUPT capability is a standard feature of the Modular One peripheral interface. An interrupt is accomplished by the peripheral initiating a three-word exchange sequence similar to that described above.

In this case, the Control Word specifies the store address of a 4-word area used to exchange the current working register context of the processor. This has the effect of simultaneously calling up a specific interrupt routine and saving the working registers of the interrupted program. The Slave Word of the exchange is extracted from the A register position in the 4-word area and the Master Word is loaded into the A register of the interrupt routine. The Master Word can therefore be used to give further information to the program about the interrupt, e.g. the peripheral status. The Slave Word can also be used if necessary to send program information to the peripheral.

Since the store address used for the register exchange is specified completely by the peripheral (within the limits of the first 8k of store), it is obviously possible for one peripheral to call up more than one interrupt routine for different functions by specifying different store addresses, which must be pre-loaded with the relevant interrupt routine register context.

This peripheral interface architecture allows implementation of the proposed functional specification in a variety of different ways. All the commands could be accommodated on a single address, and all the error flags could be combined into a single interrupt by indicating the flag bit settings in the interrupt Master Word. Separate interrupts could be used to request data input and output.

6.6 Elliott 4100

C1,C2. The 4100 processor has a small number of instructions specifically allocated for input/output purposes which confine the peripheral device to quite a restricted set of facilities in terms of program interaction.

The 4100 makes use of a peripheral port system rather than I/O bus and the selection of a particular port is done by the CPU on the basis of the address specified by the input/output instruction. Only the peripheral thus selected receives the I/O signals generated by the instruction. There are four instructions used by the processor which are communicated to the peripheral by means of two control lines in the I/O interface.

There are 8 Data In lines and 8 Data Out lines which are routed to the 4100 main accumulator, M. Two of the four instructions cause the Data Out lines to be set and the other two expect the Data In lines to be set. Of the two output instructions, one is intended for actual data output and the other for the output of a control command which can be interpreted in any way by the peripheral. Of the two input instructions, one is intended for actual data input and the other for inputting the peripheral status.

This type of input/output interface requires that the full command repertoire of the peripheral device be accommodated within the 8-bit control command, which allows 256 different commands, since the Data Out lines are strobed out in parallel. The output control command is abbreviated as OCUM, meaning 'output control unpacked from M'.

C3. The input and output of data one character at a time under program control is achieved as explained above by the two instructions IDUM and ODUM, whose meaning is obvious. Both instructions handle a single 8-bit wide character between the peripheral and the main 4100 accumulator.

C4. The interrupt facilities are also limited, since each peripheral can only request two distinct interrupts, known as INTERRUPT and ATTENTION, for which two separate lines are provided in the I/O interface. The processor is able to determine which peripheral channel originated the request by examining the bit settings in INTERRUPT and ATTENTION registers, which have one bit allocated for each channel on the system.

It is assumed that an INTERRUPT is used to signal a request for more data input or output, and the INTERRUPT condition will be cleared in the device when the corresponding IDUM or ODUM instruction is executed. The ATTENTION is assumed to signal some error condition which must be handled by the program. In order to discover the particular nature of the error condition, the program should execute an ISUM (input status) instruction which should give the setting of the various error flags associated with the peripheral. The execution of the ISUM instruction also clears the ATTENTION condition in the peripheral.

The 4100 I/O interface then tends to put the peripheral in a straight-jacket and expects it to conform to a well-defined command structure. The formalization of the procedures connecting program and peripheral means that all peripherals will be programmed in a standard way. There is no freedom for the programmer to define a wide range of special input/output instructions to be executed by the peripheral.

However, although the peripheral interface is restricted and formalized, it does have sufficient capabilities to support the communications interface specification. The commands specified for the device can be implemented by means of different control words for the OCUM instruction. The input and output of single 8-bit characters using interrupts and under program control is supported, and the error flags specified can be accommodated through the ATTENTION feature and the ISUM instruction.

6.7 IBM 1130

C1,C2. The 1130 processor includes just one instruction for controlling peripheral operations. This is the Execute Input/Output, or XIO instruction. When this instruction is executed, its address field points to a 2-word store area containing an Input/Output Control Command (IOCC). The format of the IOCC is as follows:-

ADDRESS (16 bits)	DEVICE (5 bits)	FUNCTION (3 bits)	MODIFIER (8 bits)

The first cycle of the execution of an I/O instruction results in the second half of the IOCC being gated onto the Data Out lines of the I/O interface, and a control line is pulsed to indicate that an XIO is being executed. The 1130 works on an I/O bus system and so this information is available to all the devices on the bus.

The devices must examine the DEVICE field in order to recognize their particular address. If the device code is recognized then the FUNCTION and possibly the MODIFIER field must be examined by the device. The different FUNCTION codes are interpreted by the CPU to produce different variants of the I/O instruction cycle, so the device must conform to the particular FUNCTION code settings used by the CPU. The interpretation of the MODIFIER field is entirely device-dependent.

The first cycle of the XIO execution is the same for all FUNCTION codes, but the rest of the execution is dependent on the particular FUNCTION code and may involve one or two extra cycles. The device must respond accordingly.

The seven valid FUNCTION code settings are defined as follows:-

001 - Write
The contents of the store location pointed to by the ADDRESS field of the IOCC is gated onto the Data Out lines

010 - Read
The device is expected to set the Data In lines, whose value is written into the storage location pointed to by ADDRESS

011 - Sense Interrupt
This code is used to sense the state of the INTERRUPT lines for all devices on the system and is explained in more detail below.

100 - Control

With this code, the device should interpret the MODIFIER field as device-specific control information. The ADDRESS field is also made available on the DATA OUT Lines and may be used to specify further information to the device.

101 & - Initiate Read and Initiate Write
110

These codes are used to control autonomous transfer operations only.

111 - Sense Device

This code requests the device to place the contents of its device Status Register on the Data In lines from which the CPU loads the ACC.

C3. The capability for single character input and output is available as illustrated by the Read and Write functions described above. In this case, actual storage locations are used directly as the source and destination of the data rather than the programmable registers, but the device itself is not aware of this. The CPU organizes the necessary store accesses by interpreting the ADDRESS field of the IOCC.

C4. The 1130 provides a comprehensive interrupt facility, which allows a device to request up to four distinct interrupts. There are four interrupt priority levels available, and one device can request an interrupt on each level if required. When an interrupt at a particular priority is being serviced, the program issues an XIO instruction with a SENSE INTERRUPT function code, and a device must set a unique bit on the Data In lines if it is requesting an interrupt at that priority level, which is indicated on four interface lines.

It is normal for a device to request an interrupt at only one priority level, since many conditions requiring interrupt service can be accommodated with one interrupt by setting specific indicators in the Device Status Register before requesting the interrupt. Then, once the interrupt has been identified, the program can issue a SENSE DEVICE command to read the Device Status Register and further identify the cause of the interrupt.

Thus, the 1130 I/O interface has sufficient capability on a single device address to support the proposed functional specification. The basic data input and output commands are implemented directly. The mode setting commands can be implemented using the MODIFIER field with the CONTROL function. Interrupts are available for both data and error flags. If required, data interrupts can be combined with the error flags into a common interrupt, or they could be given separate priority levels. In either case, the Device Status Register can be used to give further detailed information about the cause of the interrupt.

6.8 Common facilities of the input/output interfaces

The important conclusion that can be drawn from the foregoing discussion of small computer input/output interfaces is that it is possible to formalize all aspects of the program control of simple peripherals. As shown in the discussion of the 4100 peripheral interface all the requirements for program control of a character peripheral can be met by the following short list of facilities:-

$$\left.\begin{array}{ll}\text{1.} & \text{Output control command}\\ \text{2.} & \text{Input status flags}\\ \text{3.} & \text{Output data character}\\ \text{4.} & \text{Input data character}\end{array}\right\}\ \text{issued by program}$$

$$\left.\begin{array}{ll}\text{5.} & \text{Interrupt program to service data request}\\ \text{6.} & \text{Interrupt program to handle error condition}\end{array}\right\}\ \begin{array}{l}\text{issued by}\\ \text{peripheral}\end{array}$$

This list is actually formalized into the hardware of the 4100 peripheral interface, but the facilities can be easily mapped onto any computer interface that allows single character input/output by program and that allows interrupts.

If the small computer allows two separate 'output character' instructions, two separate 'input character' instructions and two separate interrupts for the one actual peripheral, e.g. on the PDP-8 by using two separate device addresses, then the formal scheme of the 4100 can be reproduced on that computer.

If our communications controller is designed to interface in this way to the small computer, then it can be interfaced in a compatible way to any small computer that can support the 'dual-device' facility described above. The controller will also be programmed in a compatible way on the different computers since it obeys the same four basic instructions and generates the same two interrupts.

6.9 Generalized interface description

The appearance of such a peripheral interface to the program is of four registers, each up to 8 bits wide with associated interrupt characteristics. Two of these registers are for input only and two for output only. The two input registers contain the last input character and the device status respectively. The two output registers contain the next output character and the last control command respectively. The interrupt characteristics are that two separate interrupt requests are made for the following three cases:-

a. when a transfer is made from the input shift register to the input character register - Interrupt 1
b. when a transfer is made from the output character register to the output shift register - Interrupt 1
c. when any status bit is set in the device status register - Interrupt 2

The interrupt requests are cleared when the particular register that caused the interrupt is serviced by the program, or when a Reset control command is given.

From the actual peripheral device logic side of the interface, the same four registers are seen, but with different characteristics. Two of the registers are set by the device logic, these being the input character register and the device status register, and two of the registers are read out by the device logic, these being the output character register and the control command register. Control pulses are associated with the loading and unloading of certain of the registers. The device logic sends a separate control pulse at each of the following three times:-

a. when it sets the input character register

b. when it sets any bit in the device status register from 0 to 1

c. when it unloads the output character register

The device logic expects a separate control pulse at each of the following three times:-

a. when the program reads the input character register

b. when the program loads the output character register

c. when the program loads the control command register

Providing that this 4-register structure with the associated control lines can be implemented on the input/output interface of a particular small computer, then it is possible to use that computer to control the communications interface envisaged.

6.10 General applicability

All four small computer input/output systems considered in detail in the previous discussion can be used in this way. A number of other small computers have also been studied, such as the PDP-7 and PDP-9, SPC12, Interdata 70 and Honeywell DDP516. They are all found to be capable of handling the 4-register structure specified above. In particular, the Ferranti Argus 600, which s is one of the simplest digital computers ever produced, with a very simple input/output interface, also has the necessary input/output capabilities.

It is also worth noting that a new small computer which was introduced after this study was first made has an input/output system ideally suited to the type of interface proposed. This is the DEC PDP-11, whose UNIBUS system enables a number of registers to be associated with a peripheral device. These registers are programmed as if they were normal storage locations for holding data. This obviously maps very easily onto the 4-register structure proposed for the communications controller. A comprehensive multiple interrupt facility is also available via the UNIBUS.

The use of this generalized interface in implementing the detailed functional specification of the communications interface is described in the following chapter.

Chapter 7

SMALL COMPUTER COMMUNICATIONS HARDWARE

7.1 Introduction

A previous chapter compared the use of asynchronous and synchronous transmission for computer-computer communication and concluded that synchronous mode was more suitable. This chapter describes the development of a functional specification for a synchronous communications interface for a small computer. The object of the exercise is to achieve an optimum balance between the work done by hardware and the work done by software, while retaining the degree of flexibility necessary to communicate with different large computers.

7.2 Flexibility requirement

It was observed in a previous chapter that although all the mainframe computers supported synchronous communication, they did not all implement it in the same way. Transmission codes, character sets, transmission control characters, methods of error checking, message formats and communications protocol were not generally compatible between the different mainframes. In many cases, the fact that most of the communications control was implemented by hardware meant that it was physically impossible to communicate from one large mainframe to another and that remote terminals intended to be used on one mainframe could not generally be used on a different one.

An important consideration then was that the communications hardware for the small computer should not incorporate the same inflexibility since it should be capable of communicating with all the different main computers. All aspects of communications which differed between the various main computers should be implemented by software or by program-selected hardware options.

As a general rule, there is a far higher degree of software control over peripherals on small computers than on large computers. It is quite common for peripherals to generate an interrupt for each character on a small computer. The peripheral speeds are such that this does not generate an excessive interrupt rate, and because of the simpler software structure on a small machine the CPU time involved in handling an interrupt can be quite small.

It is quite acceptable then for the communications hardware to operate by generating an interrupt per character and giving the software the responsibility for checking control characters, analysing message format and handling error checks. This approach gives the flexibility necessary for communicating with different main computers.

7.3 Basic Functional Requirements

The very minimum requirement for the communications hardware is that it should clock in the input data from the modem and convert it into N parallel bits for presentation to the computer, where N is some convenient number, preferably equal to the character size, and do the converse for output. This presents the software with the raw bit stream from the line, which has to be scanned for the synchronising pattern in order to establish the correct character frame. It is easier for the software if the hardware itself performs this first level and establishes the correct character frame so that it then presents the software with complete characters at each interrupt.

The incompatibility between different main computers usually extends to the bit pattern used to establish synchronization, i.e. the SYN character. However, one feature that is common between them is that they all support an 8-bit character size as one option, so this would seem a sensible standard to adopt.

A minimum sensible hardware requirement then is that the communications hardware should scan the incoming data stream for a program-selected SYN character and then generate an interrupt for all following characters so that the software can build up the message. On output, the requirement is simply to accept 8-bit parallel characters from the program and generate an interrupt when the next character is required.

A minimum set of commands for this controller would then be:-

1. Set SYN character, either by loading a hardware register, or by selecting one of a set of pre-wired alternatives.

2. Enter receive mode, scan for SYN character.

3. Input a character (after an interrupt).

4. Cancel receive mode.

5. Enter transmit mode.

6. Output a character (after an interrupt).

7. Cancel transmit mode.

7.4 Extra Functions

This set of commands would enable software to be written to communicate with any of the large computers. However, there are some other functions which are easily handled by hardware and which make the software simpler.

The most important of these is the timer function. The use of timeouts to recover from error situations such as failure to achieve synchronization is an essential part of the communications procedures. A hardware timer is not usually a standard feature on small computers and the use of software loop counts is cumbersome and also prone to inaccuracy in an interruptible environment. Therefore, since the timer is needed to handle the communications effectively, it seems reasonable to incorporate a hardware timer as part of the communications interface. The timer would be started by software command and would generate an interrupt when a pre-determined fixed time interval had elapsed. A command to stop the timer should also be included so that the time interval could be cancelled if it was no longer needed. Since different timeout periods are used by different large computers and also for different functions within a particular protocol, the timer interval should be small enough so that all timeout periods actually needed could be constructed by counting a suitable number of hardware timer intervals. Obviously, the timer interval should not be so short that it generated a substantial interrupt load compared with the actual communications data transfers. A timer interval of 100 ms would be suitable for all systems currently in use on the various large computers.

Another function which is more easily implemented in hardware than software is the checking and generation of character parity, where this is used. Two commands should be included to enable and disable the parity feature, since some transmission codes use all 8 bits for data and use only a block check for error checking. If the feature is enabled, the parity of input characters should be checked and an error flag set or an interrupt generated if the check fails. On output, if the feature is enabled, the parity bit position should be set to the correct value.

The use of a block check sum can impose a considerable software overhead, particularly if a cyclic redundancy check is used. For example, an implementation of a cyclic check on the PDP-8 takes an average of about 450μs per character, although for other computers with 16 or more bits per word and an 'Exclusive OR' instruction, the time can be much less(13). It would be useful if the hardware could generate the block check sum itself, especially since the PDP-8 was one of the target machines. However, the determination of which characters have to be included in the check sum is a non-trivial problem, particularly when transparent mode is being used. The hardware would have to be considerably more complicated to do this automatically and so this requirement is not included in the functional specification.

However, a separate piece of hardware, independent of the interface proper, can be used to accumulate the check sum once the software has determined which characters to include. Such a unit was developed by the ERCC Engineering Support Group and is described in (14). This unit will not be described further here since it operates independently of the basic communications interface.

A functional hardware specification for a small computer communications interface was then drawn up which incorporated the basic facilities described above. A number of other features were also specified which it was thought might relieve the software of certain tasks without restricting the flexibility of the device. These features were:-

a. automatic stripping of all leading SYN characters from an input message so that the first interrupt presented to the software was for the first significant data character of the message.

b. automatic removal of embedded SYN characters from a message since, in a text message, SYN characters are not significant and are used only for timing purposes.

c. selection of a binary, or non-text mode, in which embedded SYN characters were not removed since they may have been part of the message data.

d. automatic generation of the requisite number of SYN characters at the start of an output message so that the first output interrupt was a request for the first significant data character of the message.

e. additions to the hardware timer feature to re-initialise the timer interval whenever a SYN character was detected in the input data.

f. additions to the hardware timer feature to insert a SYN - timing sequence automatically into the transmitted data stream at a fixed interval (this capability was required by the IBM transmission adaptors).

g. automatic detection of failure to service an interrupt within the crisis time, which is one character time at any speed; on input, this condition caused an error flag to be set, on output a SYN-idle sequence was automatically inserted.

h. an error flag was incorporated to detect the loss of the data carrier signal while inputting data.

i. an error flag was incorporated to indicate that the modem had become inoperable.

j. extra commands to disable and enable interrupts from the device to protect certain critical sections of the control software.

7.5 Preliminary Functional Specification

The full functional specification proposed for hardware implementation then incorporated the following list of software commands:-

1. select SYN character from a pre-wired list of alternatives
2. enter receive mode; scan input data for two consecutive SYN characters; remove all leading SYN characters; generate interrupt on all following characters except SYN characters in text mode, which should be removed.
3. input a character from the receive buffer (should only be executed after an input interrupt).
4. cancel receive mode; cancel synchronization flag.
5. enter transmit mode; generate leading SYN characters; request interrupt for all other output characters.

6. output a character into the transmit buffer (should only be executed after an output interrupt).

7. cancel transmit mode.

8. start timer interval; enable special timer features.

9. stop timer interval; disable special timer features.

10. enable character parity checking.

11. disable character parity checking.

12. select binary input mode; do not delete SYN characters from input data.

13. cancel binary input mode; delete SYN characters from input data and do not generate an interrupt for these characters.

14. read condition of error flags.

15. enable interrupts.

16. disable interrupts.

In addition, the communications interface should generate interrupts for the following conditions:-

17. next input character available.

18. next output character required.

19. hardware timer interval has elapsed.

20. one or more of the error flags has been set.

7.6 Non-interrupt Mode of Operation

An additional consideration from the point of view of testing the communications interface and the communications link in general is the capability to write very simple test programs that operate the communications interface without using interrupts. These programs are coded as simple sequential programs. The state of the peripheral that would normally initiate an interrupt must be sensed by the program in a way not involving interrupts. This is usually accomplished by the program executing a tight loop waiting for a hardware flag to be set, or a particular bit in a status register to be set.

If possible, the communications interface should be capable of being operated in this way on all the small computers, so that the same simple test programs can be applied in all cases. One consequence of this mode of operation is that it must be possible to clear an interrupt condition without actually accepting the interrupt. This can be achieved by clearing the data interrupt flags (17 and 18 above) as part of the Read Input Buffer (command 3 above) and Load Output Buffer (command 6) instructions, and the error status flags as part of the Read Status Register (command 14) instruction.

7.7 Program Interface

The above section (7.5) describes the functional requirements of the communications interface. This functional specification must now be related to the actual program input/output instructions that are used on a particular computer. The previous chapter demonstrated that a single set of program instructions could be used on all small computers if the device was implemented in a particular way (see section 6.8).

The facilities described in section 7.5 can be associated with the six basic program interface facilities described in section 6.8 as follows:-

6.8(1) - commands 1,2,4,5,7,8,9,10,11,12,13,15,16
6.8(2) - command 14
6.8(3) - command 6
6.8(4) - command 3
6.8(5) - interrupts 17,18
6.8(6) - interrupts 19,20

Details of the actual input/output instructions that would be used in practice are given below for the four computers considered in the previous chapter. The letters a-f correspond to the facilities 6.8(1) - 6.8(6). Similar instruction lists can easily be devised for any other small computer.

<u>PDP-8</u> - assuming addresses AA and BB are assigned to the communications interface.

a. 6AA4 - Load control register from ACC
b. 6AA2 - Read contents of status register into ACC
c. 6BB4 - Load output buffer register from ACC
d. 6BB2 - Read input buffer register into ACC
e. 6BB1 - Skip if input/output data interrupt (after interrupt)
f. 6AA1 - Skip if status register interrupt (after interrupt).

These instructions can be used equally easily to program the PDP-8 in both interrupt and non-interrupt mode. The state of the hardware in non-interrupt mode can be sensed directly by the 'skip if flag set' instructions. This then satisfies the requirement stated previously that it should be possible to write very simple sequential programs to test the hardware without using interrupts.

Modular One - assuming channel address A, and store address B the start address of 8 store locations assigned to the communications interface.

Commands from program -

a. Load Control Register

Control word	- 7A,05
Slave word	- NULL
Master word	- control command value

b. Read Status Register (rejected if status flag not set)

Control word	- 7A,06
Slave word	- contents of status register
Master word	- NULL

c. Load output buffer register (rejected if output flag not set)

Control word	- 7A,07
Slave word	- NULL
Master word	- next output character

d. Read input buffer register (rejected if input flag not set)

Control word	- 7A,10
Slave word	- last input character
Master word	- NULL

Commands from communications interface:-

e. Input/output data interrupt

Control word - 70,B

Slave word - next output character

Master word - last input character

f. Status Flag interrupt

Control word - 70,B+4

Slave word - NULL

Master word - contents of status register

If programmed using interrupts, only the instruction Load Control Register needs to be used, since the other instructions are automatically included as part of the Interrupt cycle. However, the full set of instructions is necessary to program the device without using interrupts.

<u>Elliott 4100</u> - assuming address A assigned to communications interface.

a. Load control register

OCUM, A - control command value in ACC

b. Read status register; rejected if status flag not set

ISUM, A - contents of status register input to ACC

c. Load output data buffer; rejected if data output flag not set

ODUM, A - next output character in ACC

d. Read input data buffer; rejected if data input flag not set

IDUM, A - last input character loaded into ACC

e. INTERRUPT set if input or output flag set

f. ATTENTION set if any bit set in status register

This version of the interface can be programmed in non-interrupt mode by using a facility of the 4100 peripheral interface whereby a command issued to a peripheral can be rejected by the peripheral by setting a particular control line. The effect of this rejection is to cause the 4100 processor to execute the next instruction in sequence. If the command is not rejected, the 4100 program counter is incremented by one in order to skip the next instruction in sequence. Thus the program can effect a conditional branch which reflects the state of the peripheral. The peripheral should reject a command to read from the input buffer register or the device status register if their respective flags are not set. Similarly, any attempt to load the output buffer register should be rejected if the output buffer flag is not set.

<u>IBM 1130</u> - assuming peripheral address A and ILSW bits N and N+1 assigned to communications interface.

a. Load control register

IOCC - NULL ADDRESS,A,100,MODIFIER set to control command value

b. Read status register

IOCC - NULL ADDRESS,A,111,NULL MODIFIER

contents of status register loaded into ACC

c. Load output buffer register

IOCC -- STORE ADDRESS OF NEXT OUTPUT CHARACTER,A,001,NULL MODIFIER

d. Read input buffer register

IOCC - STORE ADDRESS FOR NEXT INPUT CHARACTER,A,010,NULL MODIFIER

e. Interrupt Level 2 bit N set if any status bit set

f. Interrupt Level 2 bit N+1 set if input or output data flag set.

The communications interface can be run without using interrupts by running the CPU on priority level 2 when all interrupts from the interface will not be serviced by the CPU. The ILSW instruction can be used to sense the condition of the interrupt lines under these circumstances.

7.8 Communications interface specification

The functional requirements of the communications interface discussed previously can be combined with the program interface described above to produce the following list of instructions for implementation.

The communications interface should obey the four basic instructions:-

1. Output control function from accumulator
2. Input device status to accumulator
3. Output 8-bit character from accumulator
4. Input 8-bit character to accumulator

The interface should generate the interrupts:-

1. next input or output character required
2. device status flag set

The control functions to be interpreted are:-

1. General reset to quiescent state
2. Enter receive mode
3. Enter transmit mode
4. Select SYN character
5. Enable parity checking
6. Disable parity checking
7. Set binary mode
8. Set text mode
9. Start timer interval
10. Stop timer interval
11. Enable interrupts
12. Disable interrupts

The error conditions indicated by device status are:-

1. Timer interval elapsed
2. Data carrier lost while inputting data
3. Data overrun - input character buffer not serviced in time
4. Modem inoperable condition
5. Parity error detected on input

A more detailed explanation of these functions has been given earlier in this chapter and will not be repeated here.

A block schematic for the proposed communications interface is given in Figure 7.1.

The detailed implementation of the specification given above was undertaken by the Engineering Support Group of ERCC during 1969(15).

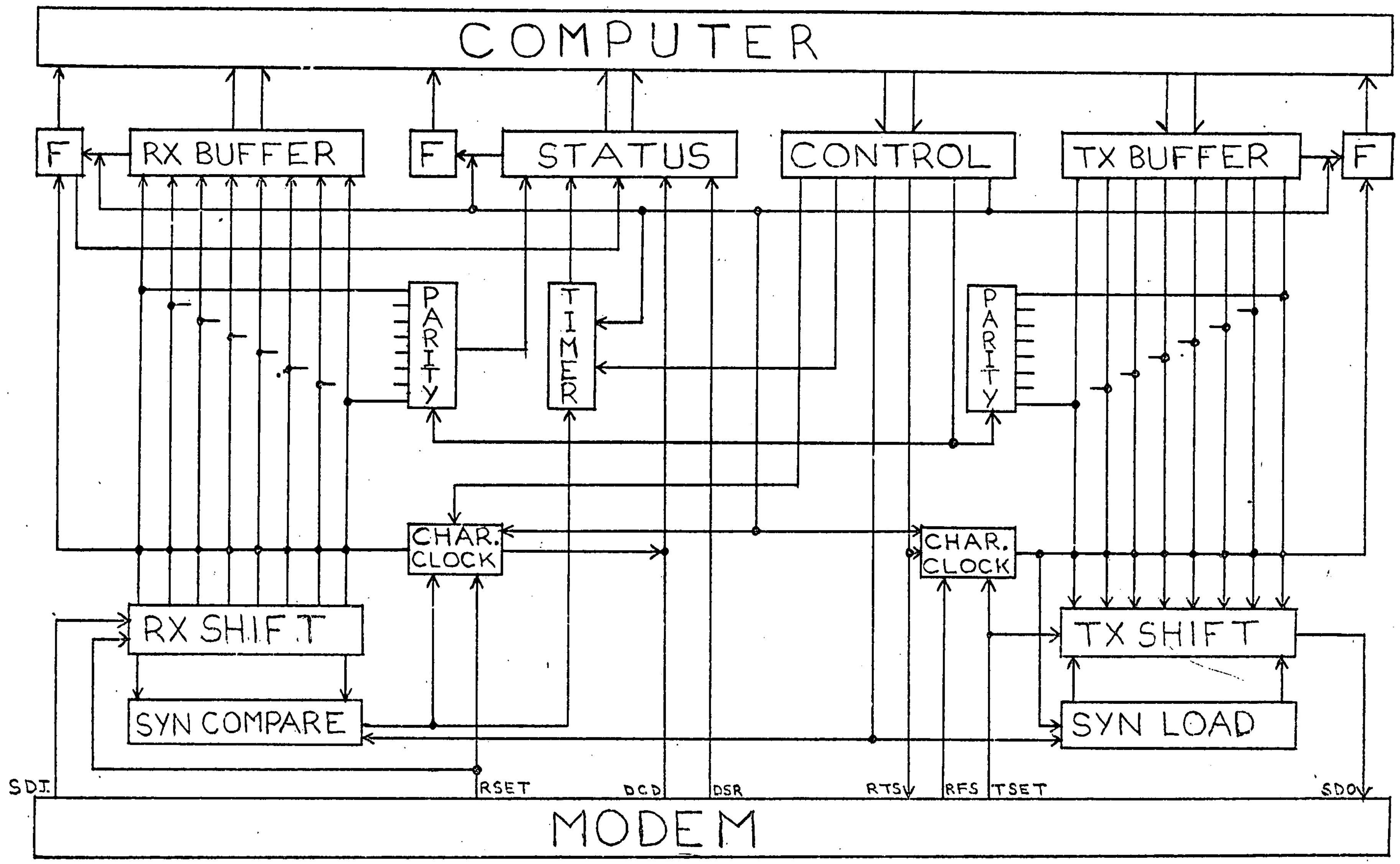


FIGURE 7.1 BLOCK SCHEMATIC OF COMMUNICATIONS INTERFACE

This hardware specification was then used as the basis for the communications software which is described later in this report.

7.9 Experience of first implementation

Software/Hardware Problems

The experience gained once the software had been written and the whole system, hardware and software, had been operational led to the conclusion that some of the extra functions specified for the hardware in the discussion on functional requirements could safely be dispensed with without imposing significant extra burden on the software.

In fact, one of the extra functions specified imposed an extra burden as onerous as the one it was intended to remove. This feature was the capability for distinguishing between binary and text mode. When text mode was set, all SYN characters found in the data stream were automatically deleted, and the program was not informed. This was intended to prevent generating interrupts for characters which would be discarded by the software, since embedded SYN characters were used only for hardware timing purposes. In binary mode, all characters were passed through to the program since the SYN pattern could occur as binary data.

However, although text messages could be guaranteed not to contain a SYN character as real data, no such assurance could be given about the block check characters, which were transmitted after the message ending character. Since the hardware did not recognise the message ending character, if one of the block check characters happen to coincide with the SYN character, then this would be deleted from the data and the block check sum as seen by the program would be incorrect. This would lead to an unrecoverable error situation, since every attempt to re-transmit the same message would produce the same effect.

Although this would seem to be an unlikely occurrence, the effect was observed in normal operating conditions. The problem was worse for IBM oommunications, which specified the use of an intermediate block check sum after every record in a transmission block in addition to the one at the end of the block.

The problem could be overcome by the program selecting binary mode whenever a block check was expected, and re-selecting text mode, if the whole block was in text mode, after the block check had been received. A simpler solution would be to read the whole block in binary mode and delete SYN characters in a text block by program. The binary/text mode feature of the hardware could then be removed.

Hardware Test Problems

A further problem encountered with this first implementation concerned the difficulties of testing the communications hardware. Since all the main computer communications hardware being considered operated in half-duplex mode i.e. only allowing data to be transmitted in one direction at a time, the small computer communications hardware was designed to operate in the same way and the operations of transmitting data and receiving data were mutually exclusive at the hardware level.

To test the communications hardware, therefore, required that there be compatible, working communications equipment at the other end of the link. This was difficult to arrange during prototype development when the only such equipment was the 360/50 with its communications adaptor. Available testing time was limited and it was exceedingly cumbersome to write the simple kind of test program for the 360 necessary for elementary engineering tests while still running in a general user multi-programming environment.

This level of engineering checkout would have been much simplified if the communications hardware could operate in full duplex mode, i.e. transmit and receive data simultaneously. The communications hardware could then be used in a looped back-to-back fashion, and data from the serial data output stage would be fed back into the serial data input stage. A simple test box used in place of the modem could provide the necessary timing and control signals.

Additional Functions Specified

Other extra facilities defined in the functional specifications, namely - stripping of leading SYN characters on input, generation of leading SYN characters on output, additional timing features associated with input and output of data, were found to produce only marginal simplification of the software. Although the extra hardware involved was not very complicated, the extra cost, size and wiring necessary was not justified by the software savings.

Additional Error Flags

Similarly, the extra error indicators used to signal loss of data carrier and modem inoperability were reckoned to be superfluous. The data carrier lost signal was intended as a fall-back method of detecting end of input data if a proper message-ending character was not recognised by the software because of data corruption. In fact, it was a common practice of main computer communications equipment to maintain data carrier on a 4-wire circuit after all data had been transmitted. This enabled faster line turnaround between receive and transmit since it was not necessary to establish and stabilise the carrier signal before transmitting data. This could save up to 140 ms on each handshake exchange over the link, which represents about 40 extra characters of data at 2400 bauds. Since the software could no longer rely on a lost carrier signal to indicate end of data, this aspect of error recovery had to be implemented using timeout controls and buffer overflow conditions. The lost carrier signal was then no longer needed.

Since there was no way in which the software could recover automatically from a modem inoperable condition, it was decided to display the state of the modem on operator lights and rely on operator action to investigate the condition of permanent failure to establish communication which would result from a faulty modem. The use of the modem inoperable error flag therefore was of marginal benefit, since it only enabled the software to inform the operator of a condition he would discover for himself. If however there were no facility for providing operator lights, then obviously the modem status must be made available to the software.

7.10 Improved Functional Specification

On the basis of the considerations outlined above, a new functional specification was drawn up which is considered to represent an optimal division of function between communications hardware and communications software on a small computer. The details of this functional specification are given below.

This functional specification was implemented by Data Dynamics Ltd. in 1971(16). Because of a commercial decision to produce a communications interface for a PDP-8 only, the interface was not implemented in accordance with the generalised 4- instruction format indicated previously. Rather, the multiple-address capability of the PDP-8 was used and several separate instructions were defined which are shown in the list below.

The communications interface should be able to operate in full-duplex mode, so the transmit and receive channels should function independently. The functions of each channel, and the commands used are described below. The interface should also incorporate a hardware timer facility, the details of which are given below.

Receive Channel

Command 1.	Load SYN recognition register from AC
Command 2.	Enter receive mode, scan incoming data for SYN character; when found transfer all subsequent 8 bit sequences to input buffer register and set input buffer flag to request an interrupt.

Command 3. Reset receive mode; cancel character synchronisation; reset any interrupt requests; ignore incoming data; reset all flags.

Command 4. Read input buffer register into AC; reset interrupt request.

Command 5. Skip if input buffer flag set.

Command 6. Skip if input buffer overrun flag set; this flag is set if the input buffer flag is still set when a second character is transferred to the input buffer; this flag does not interrupt.

Command 7. Skip if parity flag set; this flag is set if the parity of the input character is odd; this flag does not interrupt.

<u>Transmit Channel</u>

Command 1. Enter transmit mode; set all - 1's pattern in output shift register and shift out in accordance with transmit clock from modem; set output buffer flag to request interrupt when modem sets 'Ready For Sending' signal.

Command 2. Reset transmit mode; stop shifting bits from shift register; reset 'Request To Send' modem signal if in two-wire mode; set 'Transmitted Data' modem signal to 1 condition; reset output buffer flag, output buffer overrun flag and parity flag.

Command 3. Load output buffer register from AC, reset output buffer flag.

Command 4. Set parity generation flag; when set, this flag causes the most significant bit of the character to produce odd parity when the character is transferred from the output buffer register to the output shift register.

Command 5. Skip if output buffer flag is set.

Command 6. Skip if output buffer overrun flag is set; this flag is set if the output buffer flag is still set when the next output character is requested; this flag does not interrupt.

Hardware Timer

The communications interface incorporates a clock which generates a pulse every 100 ms. This pulse sets a flag to request an interrupt if a programmable gate is open.

Command 1. Enable timeout flag; this command allows the clock pulse to set the flag.

Command 2. Disable timeout flag and clear flag; this command prevents the clock pulse from setting the flag.

Command 3. Skip if timeout flag set and clear flag.

7.11 Interface Test Facility

To ease the problem of testing the communications hardware, a 'test-box' was specified which fitted in place of the modem and provided facilities for testing both transmit and receive hardware. This test-box is capable of testing any full-duplex 8-bit synchronous communications interface fitted with a modem interface.

It consisted basically of an 8-bit register and a timing generator, capable of running at various speeds. The contents of this 8-bit register could be set either from 8 hand switches or from the transmitted serial data from the communications interface. The contents of the register were shifted out to produce the serial data input for the communications interface. The shifting was controlled either from the timing generator or from a single-step switch, which produced a one-bit shift each time.

Therefore, the receive and transmit hardware could be tested alone, or in conjunction with each other at manually controlled speed or at normal operating speed. This test facility proved indispensable since the prototype communications interface was developed in an environment where no compatible communications equipment was available to attach to the other end of a data link.

7.12 Conclusions on small computer communications hardware

The arguments and conclusions presented in this chapter can be summarised by saying that the ideal implementation for a communications interface suitable for attaching to any small computer would support the functional specification defined for the Data Dynamics interface implemented with the generalised 4-instruction system used in the ERCC interface.

One important simplification of this generalised interface can be made in relation to interrupts. The ERCC specification defined two interrupts, one indicating that input or output of a character was complete, the other indicating a status report. Since an important part of the Data Dynamics specification was that it should be capable of running in full-duplex mode, the same interrupt could not be used for both input and output, since it would be impossible to tell whether the input or output register needed servicing, as they could both operate simultaneously. If an extra interrupt were defined, to enable separate interrupts for input and output, this would mean that three interrupts were needed, which could not be supported on certain of the small computers.

It is proposed then that only one interrupt would be used for all conditions, and the particular condition causing the interrupt would be indicated in the status register. Therefore two extra bits are assigned to the status register to indicate input character and output character respectively. It would be the responsibility of the software to handle all conditions indicated in the status register whenever it was read, since more than one condition could occur simultaneously. This reduction to one interrupt for all conditions also has important implications for the software which will be described in a later chapter.

A formal specification for this idealized communications interface is given in the following chapter as a self-contained summary of the foregoing deliberations on small computer synchronous communications hardware.

Chapter 8
SPECIFICATION FOR SYNCHRONOUS COMMUNICATIONS INTERFACE FOR SMALL COMPUTERS

8.1 Basic Requirement

The synchronous communications interface (SCI) should be capable of operating in an 8-bit character synchronous communications environment with a clocked modem (Modem 7 or proprietary equivalent) at speeds of 600, 1200, 2400, 4800 or 9600 bauds. Operation at the higher speeds will be over a private 4-wire circuit, so that it will be possible to maintain continuous carrier in both directions at all times thus saving line-turnaround time. In case continuous carrier operation is not compatible with other computers' communications equipment then a switch should be provided that enables/disables continuous carrier. Operation at 600 or 1200 bauds will be over the switched public network using a two-wire line and the appropriate facility of Modem 7. This is known as standby mode and should be selectable by means of a hardware switch on an operator control panel. Continuous carrier operation is not compatible with standby mode and selection of standby should override any other method of selecting continuous carrier operation.

8.2 Modem Interface Considerations

The modem interface should conform to the CCITT V24 specification(17). This is adequately described in the appropriate reference document, and frequent references will be made in this specification to the signals defined there.

8.3 Program control

The interface should contain four programmable 8-bit registers, two read-only and two write-only. The read-only registers comprise one for holding the latest input character and one for holding the device status. Whenever a bit is set in the status register, an interrupt request is generated. The write-only registers comprise one for holding the next output character and one for holding the latest program control command. All commands to the interface are effected by setting the command into the control command register.

8.4 General

The SCI consists logically of two independent parts - the Receive (Rx) channel and the Transmit (Tx) channel. In order that the SCI can operate in full-duplex mode, these two parts should be able to operate simultaneously and independently and should not interact in any way. A detailed description of the function of the two channels is given below.

8.5 Receive (Rx) Channel

The Rx channel performs the function of obtaining and maintaining character synchronization with the incoming serial data. The characters are then transferred in parallel form to an input buffer register which is accessible to the program. The Rx channel should achieve character synchronization as a result of recognising two consecutive SYN characters. The SYN character used for the comparison is loaded into an 8-bit register by the controlling software at the start of the session. This register will only be loaded once by the software, so the register contents must not be altered in any way by hardware effects, excluding those that also cause the software to be corrupted, e.g. power failure.

The Rx channel contains 3 registers:-

a) SYN comparison register (8-bits). This register is loaded by software with the appropriate SYN-character for the main-frame machine. The register is loaded once at the start of a session and it should remain unaltered throughout the session as no attempt will be made to reload it by the software during a session.

b) Input shift register (16 bits). This accepts the serial data from the modem and it is in this register that SYN recognition is performed when the Rx channel is attempting to achieve synchronization. A double comparison should be made with the 8-bit SYN register every time a new bit is shifted in from the modem. If the 16-bit register contains a double-SYN pattern, the character clock is started and <u>any further SYN recognition is inhibited</u>. The character clock is a division by 8 of the bit clock. Characters are transferred from the shift register to the Rx buffer register at every character clock time. The character clock is stopped by the 'Reset Rx channel' instruction. SYN recognition is re-enabled by the 'Enter receive mode' command.

c) Rx buffer register. This register holds a character ready to be transferred to the computer. There is a flag associated with this register which is set when a character is loaded into it from the shift register. This flag is contained in the device status register and is cleared when the Rx buffer register is read into the computer, or when the 'Reset Rx channel' command is issued. If the flag is still set when the following character is ready to be loaded into the Rx buffer, then an error flag indicating Rx buffer overrun should be set in the status register. The flag should be cleared by the 'Read Rx buffer' instruction and the 'Reset Rx channel' command. There is a further flag in the status register associated with the Rx buffer which indicates the parity of the character. If the parity of the 8-bit character is odd, then this flag is set to a 1, otherwise it is zero. The flag is set at the same time as the character is loaded into the Rx buffer. The flag should be cleared by the 'Read Rx buffer' instruction, and by the 'Reset Rx Channel' instruction.

The following instructions are used to control the receive channel:-

1. Read input buffer register and clear flags
2. Read and clear status register; the following status bits are defined for the Rx channel:-
    a. input character ready
    b. input buffer overrun
    c. parity of input character is odd
3. Load control register; the following commands are defined for the Rx channel

a. Enter Rx mode

b. Reset Rx mode and clear flags

c. Load upper half of SYN register from top 4 bits of command

d. Load lower half of SYN register from top 4 bits of command.

8.6 Modem interface considerations for Rx channel

The modem interface circuits affecting the Rx channel are Data Carrier Detected (DCD), Receiver Element Timing (RET) and Serial Data In (SDI). For continuous carrier operation, DCD will always be present, as will RET, although the SDI will only have meaningful information on it when the far end is transmitting real data. Thus, no action should be taken by the Rx channel unless DCD is present.

8.7 Transmit (TX) Channel

The transmit channel organises the output of data characters in serial form to the modem. The transmit channel is activated by means of the 'Enter transmit mode' command. This command causes the Tx channel to set an all-ones pattern in the output shift register, to set Request-to-send (if it is not permanently set) and then wait for the response Ready-for-sending (RFS), at which time the transmit timing signal will be available from the modem. When RFS is received, the Tx channel should begin shifting the data from its output shift register into the modem, and should also set a flag in the status register for the computer to fill the Tx buffer register. The two registers in the Tx channel are as follows:-

a) Tx buffer register (8-bits). This register is loaded with characters from the computer when the Tx channel generates an output interrupt request. This is done by setting the Tx buffer flag, which is used to set a bit in the status register. The flag is cleared by the 'Load Tx buffer' instruction, and by the 'Reset Tx channel' command. The Tx channel makes the first output interrupt request when RFS is received from the modem. Subsequent interrupt requests are made at every character clock time, i.e. at 8-bit intervals. If, on attempting to make an interrupt request, the Tx channel discovers that the Tx buffer flag has not been cleared, then the 'Tx buffer overrun' bit should be set in the status register. This flag is cleared by the 'Load Tx buffer' instruction, and by the 'Reset Tx channel' command. There is one further flag associated with the Tx channel which determines whether or not character parity is generated for the output character. This flag is set by software and is reset by the 'Reset Tx channel' instruction. The parity bit is the high-order bit of the character and odd parity should be generated. If the parity flag is set, then the actual setting of the high-order bit from the computer should be ignored.

b) Tx shift register (8-bits). This register accepts an 8-bit parallel transfer from the Tx buffer register for shifting out to the modem. The Tx shift register is initially set to an all-1's pattern when the command 'Enter Tx mode' is given. The shifting begins when RFS is received from the modem, and the shift pulses are derived from the 'Tx element timing' signal from the modem. When 8 bits have been shifted out, the next character is loaded from the Tx buffer register and the Tx buffer flag is set to request the software to load another character into the Tx buffer register. Shifting is stopped by the 'Reset Tx channel' command. If appropriate, this command also resets the 'Request to-send' signal to the modem. After the command 'Reset Tx channel' is issued, the signal Serial Data Out should be set to a binary 1 state, as this is the desired quiescent line state.

The following instructions are used to control the transmit channel:-

1. Load output buffer register and clear flags
2. Read and clear status register; the following status bits are defined for the Tx channel:-
    a. next output character requested
    b. output buffer overrun
3. Load control register; the following control commands are defined for the Tx channel
    a. enter Tx mode
    b. reset Tx mode and clear flags
    c. set parity generation flag

8.8 Modem interface considerations for the Tx channel

The modem interface circuits relevant to the Tx channel are Request-to-send (RTS), Ready-for-sending (RFS), Transmit-Element-Timing (TET) and Serial-Data-Out (SDO). In continuous carrier operation, RTS, RFS and TET are always present and SDO should be in binary '1' state. Real data is put on SDO whenever the SCI is in transmit mode. In other than continuous carrier operation RTS is only set when the command 'Enter transmit mode' is given to the SCI. After a short interval, the modem responds with RFS at which time TET is valid and the Tx shift register can be started. In this mode, RTS is reset when the command 'Reset Tx channel' is given to the SCI.

8.9 Time-out control

The SCI should include a clock which sets a flag in the status register every 100 ms if enabled by program. It is the responsibility of the software to time longer intervals by counting timer interrupts.

The following instructions are used to control the hardware timer:-

1. Read and clear status register; the following flag is defined:-

   a. 100 ms clock pulse has occurred

2. Load control register; the following commands are defined

   a. enable timer flag

   b. disable timer flag and clear flag

8.10 Interrupt Control

Two extra control commands are defined for the interface as a whole to enable the program to control interrupts properly.

a. enable interrupts from interface

b. disable interrupts from interface.

Chapter 9

COMMUNICATIONS SOFTWARE FOR THE SMALL COMPUTER

9.1 Overview

Previous chapters have discussed the hardware aspects of the communications link between the small and large computers. The net result of these considerations has been the specification of a synchronous communications interface suitable for interfacing to any small computer. This hardware operated to a well-defined program interface and generated an interrupt for each character to control the data transfer.

The other important aspect of the overall communications system is the software for the small computer. This software matches the interface provided by the hardware to control the communications procedures necessary for effective communication with the large computer. These communications procedures are usually referred to as 'communications protocol' since they define a set of rules which must be adhered to if effective and reliable data communication is to be achieved. These rules pertain to such things as use of control characters, message format, type of error-checking, acknowledgement format and action to be taken in the event of temporary transmission failures.

It is appropriate if all such procedures are handled completely by a communications software package, which presents a well-defined software interface to a user wishing to transmit data over the link. This combination of hardware and software package then implements the 'communications black-box' envisaged in the introductory chapter.

The software interface mentioned above will be referred to as the 'user interface' to the communications system. The user side of the interface should not be aware of any details of the communications protocol. Rather, he should be concerned only with the types of data he wishes to transmit, and these were described in the chapter on 'User Applications for Communication Links'.

The communications package has the responsibility for issuing the necessary control commands to the hardware and responding to the interrupts generated by the hardware. A general block diagram is given in Figure 9.1 showing the position of the communications package in the overall software structure for a communications-based application on a small computer.

9.2 General requirements

In line with the proposed development of a communications system that could be applied to any small computer, this communications package should be designed so as to be easily incorporated into the software system of any small computer, and should present the same user interface independent of the particular user application. Following the conclusion reached in section 5.5, the package should be designed to implement the general point-to-point contention - type protocol in a way that permits easy adaption to different versions of this basic protocol on different large computers. The user interface should remain as near as possible the same across these

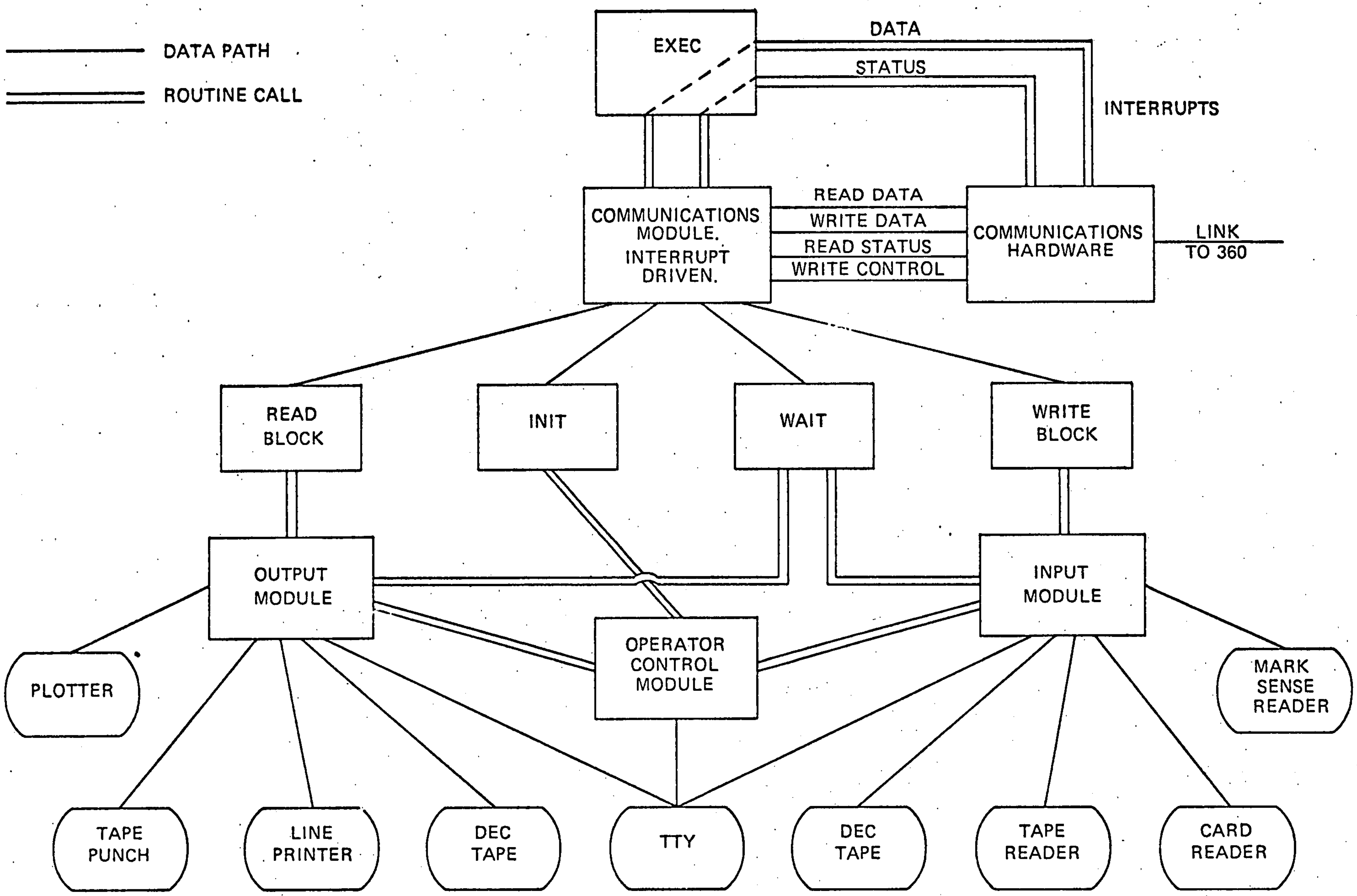


FIGURE 9.1 SMALL COMPUTER COMMUNICATION SYSTEM

different large computers. Finally, although oriented towards working with the small computer communications hardware specified in the previous chapter, the package should be capable of working with any suitable hardware if this already exists on a particular small computer.

The remainder of this chapter describes how a package was developed meeting the general requirements discussed above. The next chapter describes how it has been successfully used in a number of applications involving different small computers. A further chapter describes standardized procedures to be followed when applying the package in a new environment. This includes the case of generating a completely new version for a new small computer, where the basic interrupt-handling software can be tested in a controlled way without the real communications hardware, thereby avoiding the hazards of debugging new software in a real-time environment.

9.3 Package Implementation - General Details

Since one of the stated general objectives was to develop a package applicable to a number of different small computers, the package should be machine-independent. One way of achieving this is to program the package in a high-level language. Any functions which must necessarily be coded in the particular machine assembly language, such as issuing the basic hardware input/output commands, can be confined to a small number of well-defined hand-coded routines. These routines should only need a very few machine instructions so that they can be easily re-coded for a different small computer.

The general problem of using high-level languages for software implementation on a small computer and desirable features for such a language will be discussed in a later chapter.

However, considering the Edinburgh environment, the obvious choice of language was IMP. Although IMP was used mainly as a powerful programming language on the main Edinburgh computers, a number of smaller versions had been developed for small computers which implemented a sub-set of the full specification. In particular an IMP compiler had been developed for the PDP-8, which was the first target machine for the implementation of the communications package.

The use of a high-level language for implementing the package seemed desirable, even if an IMP compiler were not available on a particular small computer, since it would be much easier to produce a hand-coded version from an original source in IMP than from a hand-coded version for a different small computer. Any hand-coded program will take advantage of particular machine features which will make it difficult to transcribe the code to another machine without these features, e.g. hardware stack, auto-indexing. The original IMP source would then serve as a system definition language as well as the actual implementation language on a machine with a suitable compiler.

Another basic design objective is that the package should handle the data transfer on an autonomous basis using interrupts. Then, a user call on the package would merely initiate a data transfer which would be completed under interrupt control, leaving the user free to continue with other processing. This would enable double-buffering to be effected which is highly desirable since the data link speed, being only comparable with normal peripheral operating speeds, needs to be used as efficiently as possible.

The package thus has to conform to four general interfaces:-

a. the user interface, as described briefly above
b. the communications hardware interface, which will not necessarily conform to the ideal specification defined in the previous chapter
c. the executive interface, for routing through the interrupts from the communications hardware
d. the remote host computer interface, which will be based on the standard point-to-point one-way-at-a-time protocol

On the inside of these four interfaces, the package should remain essentially constant, even though everything on the outside is liable to change.

A more detailed description of these interfaces and the consequences of trying to match them in a uniform way follows below.

9.4 The User Interface

The user interface can be defined at two levels, one directly related to the initiation of activity on the communications link and the other one level removed from this. These two levels can be referred to as block and record interfaces respectively, and if the record interface is used, then this will itself make use of the block interface. Probably the most natural interface for the user in most applications is the record interface. This enables the user to perform READ and WRITE operations for single logical records, and would be equivalent to similar operations on local peripherals such as card reader, line printer and teletype.

For some operations, such as the transmission of graphical information in coded form or the transmission of binary information, the concept of single logical records is not always applicable and it is more appropriate to use the direct block interface. The block interface corresponds closely to the actual data transmission procedures used on the communications link. As was mentioned previously as part of the discussion on communications hardware, computer communications is necessarily block-oriented, and the size of block used is related to expected error rates and availability of store for buffers. Typically, a transmission block can contain several logical records if the data is record-oriented, or may just be a suitably-sized part of a binary data sequence. For transmission purposes, the block of data is enclosed in message-framing characters and may contain intermediate block-check characters as well as the block check at the end of the block.

The data exchanged over the user interface at the block interface level consists of the data blocks as they are transmitted but excluding any characters used solely for the purposes of communications protocol. The user therefore is only concerned with the data content of the block. In the case of record-oriented data, the block will contain the data characters with end-of-record characters at appropriate points. For binary data, the block will contain only the data characters.

It is clear from the above that the record-level interface uses the block-level interface, since records are merely combined into blocks before any actual data transmission takes place. The block interface is therefore more fundamental as far as the communications package is concerned, and it is this interface which will be described.

Block-oriented user interface to communications package

The user controls the package by calls on four routines, some of which require input and return output parameters.

Initialization

The first routine to be called at the start of a session is INIT. This routine has no parameters and performs the function of initializing the communications hardware and setting variables to the initial working state. Although it need only be called once, this routine may be used after a permanent failure condition to return the package to an initial state.

Receiving Data

The user calls the routine READBLOCK to initiate the reading of a block from the remote end, but the routine is not directly involved in the transfer of the data. The data is transferred under interrupt control and the user can detect termination of the transfer by calling routine WAIT (described below). Input parameters for READBLOCK are the buffer address and size. Output parameters available to the user after the transfer has terminated are:

a. a flag indicating whether the transfer was completed successfully, or whether the transfer was abandoned after a number of retries.

b. a flag indicating whether end-of-file (EOF) was received.

c. a flag indicating whether the block received was in binary mode or character mode.

d. a count indicating the number of characters actually received.

Unless flag a. is set, the other output parameters should not be examined. If flag b. is set, the other output parameters should not be examined. If flag c. is set (indicating binary mode), all characters in the block are data characters, otherwise the block may contain end-of-record characters.

Successive calls on READBLOCK may be made to receive a whole file providing that routine WAIT is called to achieve proper synchronization with transfer termination.

Transmitting data

The user calls routine WRITEBLOCK to initiate transmission of the next data block. Again, this routine does not handle any actual data transfer, which is effected under interrupt control. Routine

WAIT must be called to check for transfer termination. Input parameters to WRITEBLOCK are:

a. the address of the output buffer

b. the number of characters to be transmitted

c. a flag indicating whether the data is binary data (flag set) or character data

d. a flag indicating whether this is the last block in the file

If flag d. is set, then an EOF will be transmitted with the block to terminate the current file.

The one output parameter available after routine WAIT is called indicates whether the transfer was completed successfully or whether the transfer was abandoned after a number of retries.

Synchronization

Since routines READBLOCK and WRITEBLOCK initiate data transfers which are completed asynchronously with the user program, the routine WAIT must be called in order to synchronize correctly with the termination of the data transfer. This routine returns a parameter of value one if the transfer is still in progress and value zero if the transfer has terminated. When the transfer has terminated, the user program can examine the output parameters relevant to the original call on READBLOCK or WRITEBLOCK.

Compatibility with different mainframes

This general type of block-oriented interface is compatible with all main computers supporting the point-to-point type of communications protocol. Differences between the main computers will occur in respect of:-

a. maximum buffer size allowed

b. character set used

c. end-of-record character used in record-oriented transmissions

d. other non-transmission control characters used, such as 'newpage' specification

e. whether transmission of binary data is supported

Therefore, if the user program uses the block interface directly, it must be aware of the conventions applicable to the particular main computer. At the record-level interface, it is possible to conceal some of the differences, such as buffer size, but it would be difficult to conceal differences in the character set used, since there are no universal one-to-one mappings from one character set to another unless a restricted character set is used. The user program, then, still needs to be aware of the particular main computer it is communicating with, since this determines the type of data that can be sent. However, given this limitation, the record-level and block-level interface is generally applicable to all the large computers considered.

Compatibility with different small computers

Regarding compatibility with different small computers, the user interface defined above can obviously be implemented on any small computer. Since it is likely that different languages and different executive systems will be in use on the different computers, the precise method of invoking the various functions of the communications package is likely to be different in each case, e.g. mechanism for routine calls and parameter passing. Any such differences can be accommodated by the provision of special minimal interfacing routines which convert the routine call and parameters to the form expected by the package. Such routines would normally be written anew for each system and could even allow for different languages to be used for the user program and the communications package. If this method of a double routine call was inefficient in a particular application, then the necessary code could be incorporated directly into the four routines in the package comprising the user interface, although this is obviously a less clean way of handling such differences.

9.5 The communications hardware interface

This interface is the means by which the communications package exercises control over the communications hardware and has two distinct parts. The first is concerned with the actual commands which the software issues to the hardware and the second concerns the interrupts which the hardware can request from the software. Both these aspects have been implemented in the communications package in a generalized way, which may or may not correspond to an actual hardware implementation. However, because the interface has been implemented in terms of basic primitive functions for a character-at-a-time two-way communications channel operating in half-duplex mode, it should be possible to map this software interface onto any type of hardware implementation. In fact, this has already been done for five different hardware implementations, the details of which will be given later.

Software commands to hardware

The first aspect mentioned above is defined completely by four basic routines which are called READDATA,WRITEDATA,READSTATUS and WRITECONTROL. The definitions of these routines are as follows:-

READDATA is called as a function to get the latest input character assembled by the communications hardware; the character is returned as the function value.

WRITEDATA(CHAR) is called to set CHAR as the next character to be output by the communications hardware; character parity should be included if necessary.

READSTATUS is called as a function to fetch the latest error status report generated by the communications hardware; the valid error reports are PARITY ERROR ON INPUT, TIMEOUT, DATA CARRIER LOST, CHARACTER OVERRUN ON INPUT, MODEM FAULT; the error report is returned as the function value.

WRITECONTROL(CONTROL) is called to instruct the communications hardware to perform the function defined by CONTROL; valid control functions are SET SYN CHARACTER,SET/RESET PARITY CHECKING, ENTER RECEIVE MODE, ENTER TRANSMIT MODE, RESET COMMUNICATIONS CHANNEL, ENABLE/DISABLE COMMUNICATIONS INTERRUPTS.

A further function used by the communications package may also result in a hardware command being issued. This is the timer function, which is invoked by WRITECONTROL(STARTTIMER) to start a pre-defined time interval. If the timer is implemented by a real-time clock or in the communications hardware, then the WRITECONTROL routine will issue the necessary hardware command, and set a counter if necessary when the hardware timer interval is significantly less than that required. The implementation of the timer is a function which is machine-dependent, and generally has to be written anew for each machine. It is also a function of the WRITECONTROL command 'RESET COMMUNICATIONS CHANNEL' to cancel any outstanding time interval.

The routines listed above constitute a set of primitive operations for a half-duplex communications channel of the type considered in this report. The code for these routines will normally need to be written in assembler to perform whatever low-level hardware functions are necessary to carry out the defined functions correctly. The routines will be different for each different communications hardware implementation, but it should be possible to implement them in all cases.

The result of calling routine WRITECONTROL with parameter ENTER RECEIVE MODE or ENTER TRANSMIT MODE is that communications channel interrupts will occur at some undefined time later. Similarly, calling the routine with parameter STARTTIMER will result in an interrupt after the specified time interval, unless the timer has been cancelled or another call for STARTTIMER has been made to set a new interval. This leads on to the second aspect of the communications hardware interface, which is that of the interrupts which the hardware can request from the software.

Communications hardware interrupts

As far as the communications package is concerned, there are three interrupt conditions defined:-

a. input data interrupt, generated when the next input character has been assembled
b. output data interrupt, generated when the next output character is required
c. error status interrupt, generated when an error condition is detected by the communications hardware, such as timeout, lost carrier, etc.

Interrupt a. occurs only when the communications channel is in receive mode, as defined by the ENTER RECEIVE MODE function of WRITECONTROL. The first such interrupt signals the first character of real data, i.e. it is assumed that all leading SYN characters are removed by hardware or software.

Interrupt b. occurs only when the communications channel is in transmit mode, as set up by the ENTER TRANSMIT MODE function of WRITECONTROL. It is assumed that the first such interrupt is generated only after the requisite number of leading SYN characters have been transmitted, and so this interrupt requests the first real data character.

Interrupt a. or b. will continue at a rate determined by the line speed until the WRITECONTROL function of RESET COMMUNICATION CHANNEL is issued, which prevents any further interrupts until a new mode is selected. For each occurrence of interrupt a. or b., routine READDATA or WRITEDATA as appropriate must be called as a software response to the hardware that the interrupt has been serviced.

Interrupt c. may occur at any time if interrupts are enabled. Routine READSTATUS must be called to fetch the status word defining the particular type of error and to acknowledge to the hardware that the interrupt has been serviced. The error reports that may occur have been defined above.

To correspond to these three interrupts, there are three interrupt routines defined in the communications package. These are called RECEIVE, TRANSMIT and ANALYZESTATUS respectively. These three routines constitute the hardware interrupt interface to the communications package.

Now this interface is an idealized one conceived in terms of an ideal communications hardware interrupt structure corresponding to the three conditions defined above. The actual communications hardware used in practice might not provide these three basic interrupts, but may have a completely different interrupt structure. However, it is possible to map any type of hardware interrupt structure onto the interface described above since that interface defines the basic primitives of any single-character-transfer two-way communications channel.

A set of minimal routines called first-level routines is needed to convert the actual interrupt structure into that defined above. There will obviously be one first-level interrupt routine for each actual interrupt generated by the hardware and there may be more or less actual interrupts than the three defined above. For example, both input data interrupt and output data interrupt may be the same actual interrupt and it is then the responsibility of the first-level interrupt routine to inspect relevant software flags to determine whether the hardware is receiving or transmitting, and hence call the RECEIVE or TRANSMIT routine appropriately. (It is assumed that only half-duplex communication is being used so that it is always possible to determine unambiguously which is the current mode.) Similarly, all the different error conditions may be signalled by separate actual interrupts or by one actual interrupt. In the former case a code indicating the particular type of error must be set into a state variable by the first-level interrupt routine before invoking the ANALYZESTATUS routine.

In whatever form the interrupt structure is actually implemented, there is one basic hardware function associated with each routine. The RECEIVE routine must call READDATA, once and only once, TRANSMIT must call WRITEDATA once, and ANALYZESTATUS must call READSTATUS once. This may involve actual interactions with the hardware or just the manipulation of variables shared with the first-level interrupt routines.

Although these three interrupt conditions are defined separately it is assumed that the servicing of any one interrupt is an indivisible operation with respect to the other interrupts. This is necessary so that the manipulation of state variables common to all three interrupts can be done in a self-consistent manner.

As described above, in the event that the actual hardware interrupt structure does not correspond to the idealized interrupt structure defined, it is necessary to write a set of mapping routines to convert from one to the other. These routines will obviously need to be written anew for each different hardware implementation and also for different executives with the same hardware. It will almost certainly be necessary to code these first-level routines in assembly language since, by their very nature, they are low-level and strictly machine-dependent. However, their function is strictly defined and the code involved should be minimal. In no case, however, is it necessary to change any of the software defined within the RECEIVE, TRANSMIT, ANALYZESTATUS interface. This software is machine-independent and can safely be coded in a high-level language. Any interactions with the hardware from within this interface are accomplished by the four hardware control routines defined above.

All the software necessary to control the communications protocol is defined in the high-level language coding of the RECEIVE, TRANSMIT and ANALYZESTATUS routines together with any subsidiary routines which they may use. The particular protocol implemented is then easily transferable to different machines without any coding changes at this level.

9.6 The executive interface

The purpose of the executive interface to the communications package is to ensure that the package can have full physical control over the communications hardware. All communications hardware interrupts must be routed through to the appropriate first-level interrupt servicing routines and permission must be granted for the package to issue commands directly to the hardware. The extent to which this involves the executive will vary considerably from one system to another. This interface is therefore not precisely defined and has to be planned anew for each new system.

In almost all systems, the executive is involved to a greater or lesser extent in interrupt - handling to perform the functions which are common to all interrupts. These include such things as saving the status of the interrupted program, checking for spurious interrupts where possible, invoking the appropriate interrupt-servicing routine and providing a common exit path by which all interrupt-servicing routines can return to the interrupted program. In general, it is necessary for a non-executive program such as the communications package to perform an initialization function telling the executive about the interrupts it wishes to handle so that the correct links can be set up within the executive. In the communications package, this is taken to be one of the functions of the INIT routine, which is one of the user interface routines defined previously. The function will be different for different executives and will involve assembly coding. Inclusion in the INIT routine ensures that the interrupt links will be set up correctly before any communication is attempted.

From the above, it is obvious that, on a machine with any kind of permanent executive, it is essential that there be an approved method of setting-up interrupt links for peripherals controlled out-with the resident executive. Most executives provide some facility of this sort, and it is an essential pre-requisite for running any software using the communications package. Where there is no permanent executive in use, a stand-alone software system would be used incorporating its own basic executive. In this case, the appropriate interrupt links would normally be compiled directly into the executive code, so no initialization would be necessary. This area is the only one where the executive is involved with the communications package.

9.7 Remote Host Computer Interface

This interface is concerned with the particular physical protocol used to effect the block transfers over the link. It was decided to implement a simple type of protocol in the hope of achieving a level of compatibility with different mainframes. The protocol chosen was the point-to-point system for one-way-at-a-time data transfer, with line control being obtained by a 'bidding' arrangement. This type of protocol was supported by the three mainframe computers for which detailed protocol information was available. It was also the bottom level of proposed new standard protocols under consideration by ISO and ASA. As such, it seemed the only type of protocol with a reasonably wide measure of acceptability.

The User Interface to the package was intended to be suitable for use with different mainframes by handling all details of the protocol within the communications package. All communication control characters were inserted or removed as appropriate within the package, and only the data meaningful to the user was exchanged at the User Interface.

The general form of the protocol described above is summarized as follows.

Either end can bid for control of the line when the line is idle in order to initiate a data transmission. There is usually a standard method for resolving contention by assigning one end to be master or by using slightly different timeout periods when bidding for the line.

Either end bids for the line by transmitting the ENQ (enquiry) character. If the other end wishes to receive, it responds with a positive acknowledgement, (ACK), and the data is then transmitted in blocks, each of which must be acknowledged before the next is sent. Alternating odd and even positive acknowledgements are used to avoid the possibility of lost or duplicate blocks after error conditions. The negative acknowledgement (NAK), is used to request retransmission of the previous block. The ENQ character is also used during transmission to request a retransmission of a lost or garbled acknowledgement. The end of a data transmission is signalled by a final block-ending character of ETX (end-of-text) instead of the normal ETB (end-of-transmission block), or by the single character EOT (end-of-transmission). The line then returns to the idle state and bidding must be used to initiate the next data transmission.

While all the protocols conform to this general pattern, different implementations may use different character codes, different record structure within the block, different methods of block check and different forms of positive acknowledgement.

The communications package in its present form is able to accommodate such differences without any internal changes except for the use of different forms of positive acknowledgement. The character values used for transmission control characters are defined symbolically and can be easily changed. The package does not however try to provide a common character code at the user interface. The user program needs to be aware of the particular code in use for each mainframe. The communications package can be made insensitive to the internal structure of the block, providing it does not affect the error checking and this only occurs with the IBM protocol, for which code is included. Different block checking can be used by providing new versions of four very short routines that perform all manipulations of the block check. These are CLEAR, ADDTO, FETCH (used for output block) and CHECK (used for input block). If character parity is also used, this can be checked in the READDATA routine and generated in the WRITEDATA routine, both of which have to be re-written for each new system anyway.

However, positive acknowledgements are currently assumed to be of a particular form in the package, which has been acceptable to both IBM and ICL systems. This form is DLE-(odd/even switch) and acknowledgements are checked and generated in this form. Other protocols use a positive acknowledgement of the form <status>-ACK, where <status> contains other flags in addition to the odd/even switch. Because of this, interpretation of the odd/even switch is protocol dependent and cannot be done by simple character comparison. In order to make this aspect generalized, the functions of checking and generating positive acknowledgements would have to be split off into separate routines instead of using in-line code as a present. The CHECKACK routine would return a value indicating the type of acknowledgement received and the action taken is the same for all protocols. The SETUPACK routine would set up the appropriate acknowledgement as indicated by an input parameter and these are also generated in a uniform way for all protocols. These two routines could then be easily changed for a new system if required.

9.8 Conclusions

This chapter has described how the main body of the communications software can be made independent of the particular hardware environment in which it operates. A minimal set of interfacing routines is used between the main software body and the actual hardware and only these need to be changed for each new system. The next chapter gives details of how this has been done on the systems implemented so far.

Chapter 10

IMPLEMENTATION DETAILS IN ACTUAL SYSTEMS

10.1 Summary

The communications package described in the last chapter, communicating in IBM BSC point-to-point protocol, has so far been implemented in five different versions as follows:-

a) PDP-8 with ERCC communications controller

b) PDP-8 with Data Dynamics 6310 controller

c) ICL 4100 with ERCC communications controller

d) Modular One with 1.61 communications multiplexor

e) PDP-11 with DP11 communications controller

These five versions are significantly different in terms of the outside environment as defined by three of the four interfaces described in the last chapter. A large number of variants of these five major versions have also been produced which differ in minor ways from each other, such as different user environment. There are so far about twenty distinct configurations (see Table 10.1) which have used the communications package.

Two of these five versions have used the IMP code directly, these being a) and d) above. The other three versions used assembler hand-translated from the IMP. In addition to the standard assembler version a direct IMP version of the communications package is currently being developed for the PDP-11. This experience confirms the feasibility and also the desirability of writing this type of

Computer	Location	Non-communications use	COMMS HARDWARE	EXEC	Peripherals supported by communication system								
					TTY	PR	PP	CR	LP	MT	GP	DISK	OTHER
PDP-8/L	ERCC	RJE use only to support special peripherals for IBM 360	ERCC Synch. Comms. Interface (SCI)	Stand-alone (SA)	X	X	X				X		IBM Selectric Typewriter
PDP-8/E	ERCC	None; used only for communications testing and development	ERCC SCI	SA	X	X	X						
PDP-8	Physics Dept.	Experiment control and data logging	ERCC SCI	SA	X	X	X			X			
PDP-8	Social Medicine	Survey analysis	ERCC SCI	SA	X	X	X	X					Mark Sense Reader
PDP-8/I	Physiology Dept. Glasgow Univ.	Physiology experiments	ERCC SCI	SA	X	X						X	
PDP-8/E	Rutherford Laboratory	Neutron beam results analysis	Data Dynamics SCI	SA	X	X				X			
PDP-8/F	Animal Diseases Research Assn.	Processing of experiment data	Data Dynamics SCI	SA	X	X	X		X				

TABLE 10.1 SMALL COMPUTERS USING THE COMMUNICATION SYSTEM

Computer	Location	Non-communications use	COMMS HARDWARE	EXEC	Peripherals supported by communication system								
					TTY	PR	PP	CR	LP	MT	GP	DISK	OTHER
PDP-11/20	Medical Faculty	Medical computing; survey analysis	DP11	DOS or IOX	X	X	X	X	X	X		X	Mark sense reader
PDP-11/20	NIAE Bush Estate	Processing of experiment data	DP11	DOS or IOX	X	X	X	X	X			X	
PDP-11/10	Strathclyde University	Non; service RJE use only	DP11	IOX	X			X	X				
PDP-11/20	College of Agriculture	Processing of experiment data	DP11	IOX	X	X	X		X				
PDP-11/20	Social Science Faculty	Survey analysis; general applications	DP11	IOX	X						X		
PDP-11/20	Chemistry Department	Experiment control and data collection	DP11	DOS or IOX	X	X	X			X		X	
PDP-11/40	Science Faculty	Non; service RJE only	DP11	IOX	X			X	X				
PDP-11/45	Physics Dept.	Experiment control and data collection	DP11	S. Hayes EXEC	X	X	X			X		X	

TABLE 10.1 SMALL COMPUTERS USING THE COMMUNICATION SYSTEM (continued)

Computer	Location	Non-communications use	COMMS HARDWARE	EXEC	Peripherals supported by communication system								
					TTY	PR	PP	CR	LP	MT	GP	DISK	OTHER
Modular One	ERCC	Non; communications testing and development	1.61	E2 or MISER	X	X	X						
Modular One	Glasgow University	None; service RJE and communications development	1.61	E2 or MISER	X	X		X	X		X	X	
Modular One	MRC Unit W.Gen.Hospital	Automatic chromosome recognition system	1.61	E2	X	X		X	X				Shared core
Modular One	Culham Laboratory	General laboratory applications	1.61	MISER	X	X		X	X				
ICL 4120	Napier College	Teaching and research	ERCC SCI	NICE	X	X	X						
ICL 4130	Heriot-Watt University	Teaching and research	ERCC SCI	DES1	X	X	X	X	X				

TABLE 10.1 SMALL COMPUTERS USING THE COMMUNICATIONS SYSTEM (continued)

software in a high-level language.

A version of the package was also prepared for the PDP-7, PDP-9 and PDP-15, which have a compatible instruction set. This version was never put into use however, as the original application for it was not carried through.

The five versions listed above all communicate in the IBM version of the point-to-point protocol. A sixth version has also been produced which provides communication with the ICL 4-75. This version is coded in IMP and runs on the Modular One in the same local environment as the IBM version.

The protocol interface has been changed in the ways indicated in the previous chapter (section 9.7) in order to conform to the ICL requirements, which were oriented to ISO transmission codes. The symbolic values of the transmission control characters (STX,ETX,etc) were changed to ISO values. The use of error checking based on intermediate records was suppressed using a control flag. The method of block checking was changed from cyclic checking to simple Longitudinal checking by re-coding the appropriate routines and character parity was checked and generated in READDATA and WRITEDATA respectively. The IBM method of alternating acknowledgements (by DLE - <odd/even switch>) was acceptable to the 4-75 and so it was not necessary to change this apart from re-defining the symbolic values.

This alternative protocol version demonstrates the capability of the communication system to operate with different mainframes, provided they support the general type of protocol implemented. A later chapter on communications protocols in general indicates the difficulty of achieving compatibility if higher-level, more sophisticated protocols are used.

The experience gained from applying this communications package to a wide range of applications and environments confirms the validity of the original approach. By designing the software around a set of carefully-defined interfaces which were not particular to any one machine or any one application, it has been possible to transfer the package around very easily, considering that it works at a very intimate level with the hardware.

In view of the considerable differences in the environments in which the communications package has been successfully applied, it can be claimed with a certain degree of confidence that it could also be easily applied to any other small computer or different environment and provide the same facilities. A systematic method for applying the package to a new small computer is described in a later chapter.

Particular details about how it was applied in the five versions listed above are given in this chapter.

10.2 PDP-8 with ERCC communications controller

Hardware Interface

This communications hardware, which was described in a previous chapter, most closely resembles the conceptual hardware interface for the communications package defined previously. This hardware generates two interrupts, one for next input or output data character and the other to signal that an error status report has been generated. The second interrupt maps directly onto the ANALYZESTATUS routine, after the contents of the error status register have been converted to the form acceptable to the routine. The first interrupt, however, must be handled first by a small section of code which checks the state of a software indicator to decide whether to call the RECEIVE or TRANSMIT routine. Since the package is only intended to work in a half-duplex environment, there is never any danger of ambiguity, providing the hardware control routines which set and reset receive and transmit modes also set the software indicator correctly.

Three of the four routines for software control of the hardware map directly onto the basic instructions of the communications controller. The machine code bodies of these routines contain a minimal amount of code to execute the basic I/O instruction and pass the parameter via the accumulator.

Most of the WRITECONTROL functions also map directly onto the hardware, but some require some extra code to set the input/output interrupt marker.

The actual coding required to implement the hardware interface is given on subsequent pages.

Executive Interface

The executive interface in this implementation did not require any coding in the INIT routine since the system ran as a stand-alone program incorporating a minimal executive and the necessary interrupt links were compiled directly into the executive.

```
/ HARDWARE INTERFACING SOFTWARE FOR PDP-8 WITH ERCC CONTROLLER.
/ ASSEMBLER CODE VERSION.
/
/ INTERRUPT CODE.
/
          6301                          /TEST FOR ERROR INTERRUPT
          JMP   NEXT                    / NO
          JMS I ASLINK                  /OTHERWISE CALL ERROR ROUTINE
          JMP   RETURN                  /GO BACK TO INTERRUPTED PROGRAM
NEXT,     6311                          /TEST FOR DATA INTERRUPT
          JMP   OTHER                   /NO - TEST OTHER FLAGS
          TAD   INOUTSTAT               /TEST FOR INPUT OR OUTPUT MODE
          SNA                           /IF ZERO, THEN NEITHER
          JMP   RESET                   /  SO IGNORE AND RESET CHANNEL.
          SPA CLA                       /IF NEGATIVE .
          JMP   TX                      /  THEN OUTPUT
          JMS I RXLINK                  /OTHERWISE CALL RECEIVE ROUTINE
          JMP   RETURN                  /RETURN TO INTERRUPTED PROGRAM
TX,       JMS I TXLINK                  /CALL TRANSMIT ROUTINE
          JMP   RETURN                  /RETURN
RESET,    6304                          /RESET CHANNEL
          JMP   RETURN                  /  AND RETURN
ASLINK,   ANALYZESTATUS
RXLINK,   RECEIVE
TXLINK,   TRANSMIT
INOUTSTAT,0
/
```

[cont'd]

```
/ HARDWARE CONTROL ROUTINES.
/
READDATA,0
         CLA
         6312                     /READ THE INPUT BUFFER
         JMP I READDATA
WRITEDATA,0
         6314                     /LOAD OUTPUT BUFFER
         CLA
         JMP I WRITEDATA
READSTATUS,0
         CLA
         6302                     /READ THE STATUS REGISTER
         RAL;RTL;RTL              /SHIFT STATUS TO BOTTOM OF AC
         JMP I READSTATUS
WRITECONTROL,0
         DCA   FUNCTION           /SAVE FUNCTION
         TAD   FUNCTION           /ADD FUNCTION
         TAD   TABLEBASE          /  TO TABLE BASE
         DCA   POINTER            /    TO GET
         TAD I POINTER            /      HARDWARE COMMAND.
         DCA   COMMAND            /SAVE COMMAND
         TAD   COMMAND            /TEST IF BOTTOM
         AND   MASK377            /  8 BITS
         SZA CLA                  /    ARE ZERO.
         JMP I COMMAND            /IF NOT, SPECIAL COMMAND - JUMP TO CO
         TAD   COMMAND            /OTHERWISE, LOAD HARDWARE COMMAND
         6304                     /  AND ISSUE IT TO CONTROLLER.
         CLA
         JMP I WRITECONTROL       /RETURN
FUNCTION,0
TABLEBASE,TABLE
POINTER, 0
COMMAND, 0
MASK377, 377
TABLE,   SETSYN                   /=2000 FOR IBM EBCDIC SYN
         ENTERRX                  /NOT SIMPLE HARDWARE COMMAND
         ENTERTX                  /NOT SIMPLE COMMAND
         STARTTIMER               /=3400
         RESETCHANN               /NOT SIMPLE COMMAND
         SETPARITY                /NOT USED FOR EBCDIC
         UNSETPARITY              /NOT USED
         ENABLEINTS               /=7000
         DISABLEINTS              /=7400
ENTERRX, IAC                      /SET INTERRUPT MARKER TO 1
         DCA   INOUTSTAT          /  FOR RECEIVE MODE.
         TAD   RXCOMMAND          /LOAD HARDWARE RX COMMAND
         6304                     /  AND ISSUE IT
         CLA
         TAD   SETTRANSP          /ALSO SET TRANSPARENT MODE
CONT,    6304                     /  FOR WHOLE MESSAGE.
         CLA
         JMP I WRITECONTROL       /RETURN
ENTERTX, CLA CMA                  /SET INTERRUPT MARKER TO -1
         DCA   INOUTSTAT          /  FOR TRANSMIT MODE.
         TAD   TXCOMMAND          /LOAD HARDWARE TX COMMAND
         JMP   CONT               /ISSUE IT AND RETURN
RESETCHANN,DCA INOUTSTAT          /SET INTERRUPT MARKER TO 0.
         6304                     /ISSUE GENERAL RESET COMMAND
         JMP I WRITECONTROL       /RETURN
RXCOMMAND,4400
TXCOMMAND,2400
SETTRANSP,1400
```

User Interface

The user interface had to cater for the user program being written in assembler while the communications package was written in IMP. A parameter area and address pointers to the four user interfacing routines were located in a reserved area in page 0. A JMS INDIRECT instruction was used to enter the routine via its address pointer, and a few in-line machine code instructions in the routines were needed to copy the parameters from the reserved area to the permanent IMP globals where they could be easily accessed by the package. Return parameters from the WAIT routine were set by similar in-line machine code.

10.3 PDP-8 with Data Dynamics 6310 communications controller

Hardware Interface

The 6310 controller is a full-duplex single-line controller with program-specified SYN character recognition and a 100ms interval timer as two relevant features.

Data Input

The receive channel is enabled by software command which causes it to scan the incoming data for the SYN pattern. As soon as this is found the controller will generate input interrupts for each following character so there is no automatic deletion of leading SYN characters. These have to be removed by the first-level input interrupt routine before passing all subsequent message characters on to the RECEIVE routine. At the end of the message, the 'disable receive channel' command is issued which prevents any further interrupts and stops all activity in the receive channel.

Data Output

The WRITECONTROL function 'enter transmit mode' is issued directly to the output channel. If the channel is in 4-wire mode, then continuous carrier is maintained, so the output channel responds immediately with an output interrupt to request the first character. Leading SYNs for the message must be generated by the first-level output interrupt routine. When the required number have been generated, all further output interrupt requests are passed directly on to the TRANSMIT routine to output the message. In 2-wire mode, which is selected by an operator switch rather than by software, the command 'enter transmit mode' causes 'Request to Send' to be sent to the modem, and the controller will wait for 'Ready For Sending' to come back from the modem before generating the first output interrupt request. At the end of message output, the command 'cancel transmit mode' is sent to the controller.

This stops output channel activity and prevents any further interrupts. In 4-wire mode, carrier is maintained by keeping 'Request To Send' set and the modem output data line is held in a 1 condition, which produced the required quiescent MARK condition on the outbound circuit. In 2-wire mode, 'Request To Send' is cleared to allow the other end to turn the line around.

```
/ HARDWARE INTERFACE SOFTWARE FOR PDP-8 WITH DATA DYNAMICS CONTROLLER
/
/ INTERRUPT CODE
/
          6401               /TEST FOR RECEIVE INTERRUPT
          JMP   TRYTX        /NO
          6412               /TEST FOR RECEIVE OVERRUN
          JMP   EXTRACT      /NO
          TAD   OVERRUN      /YES. SET STATUS
          DCA   STATUS       /  TO OVERRUN
          JMS I ASLINK       /     AND CALL ANALYZESTATUS.
EXTRACT,  6402               /READ INPUT CHARACTER
          DCA   RDCHAR       /SAVE IT
          TAD   LEADSYN      /ARE WE SKIPPING LEADING SYNS?
          SNA CLA            /YES
          JMP   CALLRX       /NO - CALL RECEIVE
          TAD   RDCHAR       /TEST FOR
          TAD   MSYN         /  SYN
          SNA CLA            /NO
          JMP   RETURN       /YES - IGNORE IT
          DCA   LEADSYN      /OTHERWISE, CANCEL LEADING SYN FLAG
CALLRX,   JMS I RXLINK       /  AND CALL RECEIVE
          JMP   RETURN       /RETURN TO INTERRUPTED PROGRAM
TRYTX,    6431               /TEST FOR OUTPUT INTERRUPT
          JMP   TESTTIMER    /NO
          TAD   LEADSYN      /ARE WE SENDING LEADING SYNS?
          SNA                /YES
          JMP   CALLTX       /NO - CALL TRANSMIT
          TAD   MINUS1       /OTHERWISE REDUCE LEADING
          DCA   LEADSYN      /  SYN COUNT BY 1
          TAD   SYN          /     AND OUTPUT
          6432               /       A SYN.
          JMP   RETURN       /RETURN
CALLTX,   JMS I TXLINK       /CALL TRANSMIT ROUTINE
          JMP   RETURN       /RETURN
TESTTIMER,6451               /TEST FOR 100MS TIMER INTERRUPT
          JMP   OTHER        /NO
          TAD   TIME         /IS TIMER COUNT ACTIVE
          SNA CLA            /YES
          JMP   STOPTIME     /NO - INHIBIT TIMER INTERRUPTS
          ISZ   TIME         /DECREMENT (NEGATIVE) COUNT BY 1
          JMP   RETURN       /RETURN IF NOT ZERO
          6454               /OTHERWISE INHIBIT TIMER INTERRUPTS
          TAD   TIMEOUT      /SET STATUS REPORT
          DCA   STATUS       /  TO TIMEOUT
          JMS I ASLINK       /CALL ANALYZESTATUS
          JMP   RETURN       /RETURN
STOPTIME,6454                /INHIBIT TIMER INTERRUPTS
          JMP   RETURN       /RETURN
RXLINK,   RECEIVE
TXLINK,   TRANSMIT
ASLINK,   ANALYZESTATUS
RDCHAR,   0
STATUS,   0
OVERRUN,  5
TIMEOUT,  2
LEADSYN,  0
TIME,     0
MINUS1,   -1
SYN,      62
MSYN,     -62
```

[cont.d]

```
/ HARDWARE CONTROL ROUTINES
/
READDATA,0
          CLA
          TAD    RDCHAR          /FETCH INPUT CHARACTER
          JMP I READDATA
WRITEDATA,0
          6432                   /OUTPUT CHARACTER
          CLA
          JMP I WRITEDATA
READSTATUS,0
          CLA
          TAD    STATUS          /FETCH STATUS REPORT
          JMP I READSTATUS
WRITECONTROL,0
          DCA    FUNCTION        /SAVE FUNCTION
          TAD    FUNCTION        /THEN DO SWITCH
          TAD    TABLEBASE       /  ON FUNCTION
          DCA    POINTER         /     USING
          TAD I POINTER          /        JUMP
          DCA    POINTER         /           TABLE.
          JMP I POINTER
FUNCTION,0
TABLEBASE,TABLE
POINTER, 0
TABLE,    SETSYN
          ENTERRX
          ENTERTX
          STARTTIMER
          RESETCHANN
          SETPARITY
          UNSETPARITY
          ENABLEINTS
          DISABLEINTS
SETSYN,   TAD    SYN             /LOAD APPROPRIATE SYN CHARACTER
          6404                   /PUT IT INTO SYNC REGISTER
          CLA
          JMP I WRITECONTROL
ENTERRX,  IAC                    /SET LEADING SYN FLAG
          DCA    LEADSYN         /  TO 1.
          6414                   /ENABLE RECEIVE CHANNEL
          JMP I WRITECONTROL
ENTERTX,  TAD    NUMSYNS         /SET LEADING SYN
          DCA    LEADSYN         /  COUNTER.
          6441                   /ENABLE TRANSMIT CHANNEL
          JMP I WRITECONTROL
STARTTIMER,TAD MINUS20           /SET COUNT FOR
          DCA    TIME            /  2-SECOND TIMEOUT.
          6452                   /ENABLE TIMER INTERRUPTS
          JMP I WRITECONTROL
RESETCHANN,6421                  /RESET RECEIVE CHANNEL
          6442                   /RESET TRANSMIT CHANNEL
          6454                   /INHIBIT TIMER INTERRUPTS
          JMP I WRITECONTROL
SETPARITY,
UNSETPARITY,HLT                  /PARITY NOT USED FOR IBM EBCDIC
ENABLEINTS,ION
          JMP I WRITECONTROL
DISABLEINTS,IOF
          JMP I WRITECONTROL
NUMSYNS, 6
MINUS20, -24
```

Error reports

The important timeout function can be easily implemented using the 100ms interval timer built into the controller. Software commands are provided to enable and disable interrupts from this timer. The STARTTIMER enables the timer interrupt and sets an appropriate count into a store location. Subsequent timer interrupts decrement this count until it reaches zero. Further timer interrupts are then disabled and a store location reserved for the status report is set to indicate timeout before calling the ANALYZESTATUS routine.

A DATA OVERRUN error indication is also provided by the controller although it does not generate an interrupt. A flag is used which can be tested by the first-level interrupt routine for every input-interrupt. If the flag is set then the ANALYZESTATUS routine can be called with the status report set to the appropriate value for OVERRUN. When this routine returns, the RECEIVE routine can be called to service the input character in the normal way.

The controller does not provide any error reports for LOSTCARRIER or MODEMFAULT, but these are not particularly important for error recovery purposes. The first case can be handled by timeout and the second case can be indicated to the operator by lamp displays.

Hardware Control

Of the four hardware control functions, only WRITEDATA maps directly onto the real hardware function. READDATA and READSTATUS access reserved store locations set by the first-level interrupt routines to obtain the current input character and error status report respectively.

WRITECONTROL uses a combination of direct hardware commands such as 'enter transmit mode' and 'enable receive channel' with the setting of software flags inspected by the first-level interrupt routines. These software flags are used to control such things as generation and removal of leading SYN characters.

This controller is seen to match fairly well the conceptual interface defined previously. This is mainly due to the fact that the design was specified after the software had been developed and tested with the ERCC communications controller.

Executive Interface

This controller has so far been used only in small-core stand-alone systems with a minimal executive compiled in with the communication system, obviating the need for any run-time interrupt linking.

User Interface

Although the original PDP-8 communications package was written in IMP, the version used in this system was a hand-translated assembler version, produced to reduce the core requirements in minimum core systems. The user program was also assembler-coded and the user interface routines were accessed through indirect links in page 0. Parameters were also passed through reserved page 0 locations.

10.4 ICL 4100 with ERCC communications controller.

Hardware Interface

Since this hardware is very similar to that used on the first PDP-8 implementation, the implementation of the hardware interface is almost identical. The executive uses a different method of identifying the interrupt, since the 4100 interrupt system is different from the PDP-8, but the same two interrupts are routed through to a data control routine and the ANALYZESTATUS routine respectively, where the action taken is the same as on the PDP-8.

The four hardware control routines are again almost identical, with the few machine code instructions being oriented to the particular I/O instructions and register usage on the 4100.

Executive Interface

Since this version was implemented using a standard resident executive, namely the NICE executive, (18), some initialization code was necessary to establish the interrupt links. This was achieved according to the technique suggested in the NEAT reference manual (19) by setting the appropriate routine addresses into the executive interrupt transfer table in place of the null entries, for the device addresses which the communications package wished to service. The suggested technique was simple and effective, but also required that the links be set back to the null values at the end of a communication session to avoid spurious communications interrupts causing havoc. This was achieved by taking an orderly exit from the terminal program and restoring the original interrupt table entries.

```
          NOTE HARDWARE INTERFACE SOFTWARE FOR ICL 4100
          NOTE    WITH ERCC CONTROLLER
          NOTE INTERRUPT CODE
          BLOCK CONTROL       HANDLES DATA INTERRUPTS
          NOTE FIRST DUMMY CALL IS TO SET UP INTERRUPT LINKS WITHIN NI
          DATA
LINK                          RETURN LINK
          CODE
          JFL   *SETINT       SET UP INTERRUPT LINK ON FIRST CALL
          LD    0             ACTUAL INTERRUPT ENTRY POINT
          ST    LINK          SAVE RETURN LINK LOCALLY
          LD    INOUTSTAT     TEST SOFTWARE INTERRUPT FLAG
          JZ    RESET         IF ZERO, IGNORE INTERRUPT AND RESET
          COMP:L 1            TEST FOR INOUTSTAT=1
          JZ    INPUT         IF YES, RECEIVE INTERRUPT
          JFL   *TRANSMIT     OTHERWISE CALL TRANSMIT
          JI    LINK          RETURN
INPUT     JFL   *RECEIVE      CALL RECIVE INTERRUPT ROUTINE
          JI    LINK          RETURN
RESET     OCUM  10            ISSUE RESET COMMAND TO CONTROLLER
          JB    4               (IF REJECTED)
          JI    LINK          RETURN
          BLOCK SETINT        ROUTINE TO PLUG NICE INTERRUPT TABLE
          CODE
          LDR   ¬ITABLE       SAVE PRESENT
          LD:M  10              ITABLE
          ST    KEEPINT           ENTRY. (KEEPINT IS A GLOBAL)
          LD    0             REPLACE IT WITH START
          ST:M  10             ADDRESS OF CONTROL ROUTINE.
          JFL   *SETATT       NOW SET ATTENTION TABLE ENTRY
          JF    *ANALYZESTATUS GO TO ROUTINE WHEN ATTENTION OCCURS
          BLOCK SETATT
          CODE
          LDR   ¬ATABLE       SAVE CURRENT ENTRY FOR
          LD:M  10              DEVICE
          ST    KEEPATT           NUMBER 10. (KEEPATT IS A GLOBAL)
          LD    0             REPLACE IT WITH START
          ST:M  10              ADDRESS OF JUMP TO ANALYZESTATUS
          JB    *MAINSTART    RETURN TO MAINLINE CODE DIRECTLY
```

[cont'd]

```
          NOTE HARDWARE CONTROL ROUTINES.
          BLOCK READDATA
          CODE
IN        IDUM   10               READ INPUT BUFFER
          JF     ERROR            REJECTION BRANCH
          JI     0                RETURN IF OK WITH CHAR IN M
ERROR     ISUM   10               IF ERROR, READ STATUS REGISTER
          JB     4                LOOP IF REJECTED
          JB     IN               TRY TO READ INPUT BUFFER AGAIN
          BLOCK WRITEDATA
OUT       RTOM                    MOVE OUTPUT CHAR TO MAIN REGISTER
          ODUM   10               SEND IT TO HARDWARE BUFFER
          JF     ERROR            JUMP IF REJECTED
          LD:L   0                ZERO M
          JI     0                  AND RETURN IF SUCCESSFUL.
ERROR     ISUM   10               TRY TO READ STATUS REGISTER
          JB     4                  IN TIGHT LOOP
          JB     OUT              TRY OUTPUT AGAIN
          BLOCK READSTATUS
          CODE
          ISUM   10               READ STATUS REGISTER
          JB     4                LOOP IF REJECTED
          JI     0                EXIT WITH STATUS REPORT IN M
          BLOCK WRITECONTROL
          CODE
          JI:M   TABLE            USE FUNCTION IN R FOR INDEXED JUMP
TABLE     SETSYN
          ENTERRX
          ENTERTX
          STARTTIMER
          RESETCHANN
          SETPARITY
          UNSETPARITY
          ENABLEINTS
          DISABLEINTS
          NOTE LOAD APPROPRIATE HARDWARE COMMAND AND JUMP
SETSYN    LD:L   4                  TO COMMAND ISSUING SEQUENCE.
          JF     COMMAND
ENTERRX   CLS    INOUTSTAT        SET SOFTWARE INTERRUPT
          INCS   INOUTSTAT          FLAG TO 1.
          LD:L   9
          JF     COMMAND
ENTERTX   CLS    INOUTSTAT        SET SOFTWARE INTERRUPT
          DECS   INOUTSTAT          FLAG TO -1.
          LD:L   5
          JF     COMMAND
STARTTIMER LD:L 7
          JF     COMMAND
RESETCHANN CLS INOUTSTAT          CLEAR SOFTWARE INTERRUPT FLAG
          LD:L   0
          JF     COMMAND
SETPARITY J      390              ERROR EXIT TO EXECUTIVE
UNSETPARITY J    390                IF THESE CALLS ARE USED.
ENABLEINTS LD:L 14
          JF     COMMAND
DISABELINTS LD:L 15
COMMAND   OCUM   10               ISSUE COMMAND IN M TO CONTROLLER
          JB     4                TRY AGAIN IF REJECTED
          LD:L   0                ZERO M
          JI     0                RETURN
```

User Interface

Since both the user program and the communications package were coded in NEAT assembler for this implementation, the user interface was simply implemented by a standard assembler routine call, passing the parameters as named global variables.

10.5 Modular One with 1.61 communications multiplexor

Hardware Interface

Synchronous communication on the 1.61 communications multiplexor (20) provides an interface to the software which is completely different from the conceptual interface defined above, and a certain amount of coding was required to map from one to the other.

The 1.61 is different mainly because it is a multiplexor rather than a single line communications controller. As such, it was designed to handle a large number of lines of different speeds, both synchronous and asynchronous, with the minimum amount of hardware and the maximum amount of shared logic. This meant that synchronous and asynchronous channels were handled in a similar way, and anything special was left to the software.

The multiplexor basically assembles groups of bits from the line on input up to 8 at a time, and places them in a circular buffer which must be examined regularly by the software. On output, the multiplexor sends out a sequence of up to 11 bits, placed in a reserved multiplexor register by the software, and makes an entry in the common circular buffer when the output is complete.

The multiplexor also generates a regular clock interrupt, at a 1.3ms interval, which is a signal to the software to examine the circular buffer to see if any input has been assembled or output complete. These clock interrupts occur all the time, independent of any data traffic, and so are unrelated to the arrival or output of characters. It is the responsibility of the software that responds to these clock interrupts to provide a 'character-interrupt' - type interface to subsequent servicing routines, such as RECEIVE and TRANSMIT.

Data Input

Input data can appear on the line at any time and will cause entries to be made to the circular buffer. There is no way of telling the hardware to ignore input data except by disabling the whole channel for input and output. Therefore, a set of software flags have to be used so that the interrupt routine examining the circular buffer can know whether to ignore input entries or process them. When the main software wishes to accept input data, the WRITECONTROL routine is called to execute 'enter receive mode'.

This results in appropriate flags being set to indicate to the interrupt routine that if synchronization is obtained on an input message, then all characters after the leading SYNs should generate a call to the RECEIVE routine to analyze the message.

For the multiplexor synchronous channels, the groups of 8 bits assembled on input are not necessarily on character boundaries since the hardware does not attempt to perform SYN-character recognition. The interrupt routine analyzing the input data has to scan for the SYN pattern by combining two consecutive 8-bit groups into a 16-bit pattern and looking for a SYN character somewhere in the 16 bits. If it is found, the phase shift (number of bits by which the 8-bit group differs from a character boundary) is calculated so that subsequent whole characters can be assembled from the following 8-bit groups. In performing this function, the software is doing the job normally done by hardware on a single-line synchronous controller such as that on the PDP-8.

The interrupt routine further discards all leading SYN characters until it reaches a non-SYN character. This is the point at which a normal synchronous channel would generate the first input interrupt, which would be routed through to the RECEIVE routine. Therefore, the multiplexor interrupt handler at this point places the non-SYN character into a convenient store location and calls the RECEIVE routine. The RECEIVE routine will eventually call the READDATA routine which can access the appropriate store location to obtain the current input character.

When the RECEIVE routine detects the end of the input message, it calls the WRITECONTROL routine to execute 'cancel receive mode', which then sets the relevant software flags such that any further input data is ignored and RECEIVE is not called.

```
! MODULAR ONE 1.61 MULTIPLEXOR HANDLER.
! INTERRUPT CODE.
!
!          ASSEMBLERCODE MULTIPLEXOR DRIVER.
! THE FOLLOWING COLLECTION OF ASSEMBLER ROUTINES GIVE THE MPXR THE
!    APPEARANCE OF A HALF-DUPLEX SINGLE LINE CONTROLLER TO THE IMP
!    PACKAGE.
! UNDER CONTROL OF TWO VARIABLES (INOUTSTAT & FINDSYNC) WHICH ARE SET
!    BY THE IMP THESE ROUTINES PERFORM FUNCTIONS SUCH AS FINDING SYNCH
!    IN RECEIVE MODE AND STRIPPING ALL LEADING SYN'S, AND GENERATING A
!    SPECIFIED NUMBER OF LEADING SYN'S IN TRANSMIT MODE.
! VALUES OF THESE TWO VARIABLES ARE AS FOLLOWS :-
!    INOUTSTAT=0 - IGNORE ALL INPUT/OUTPUT REQUESTS FROM THE MPXR.
!    INOUTSTAT=1; FINDSYNC=1 - RECEIVE MODE, SCANNING FOR SYNCH
!    INOUTSTAT=1; FINDSYNC=0 - SYNCH FOUND ,STRIPPING LEADING SYN'S
!    INOUTSTAT=1; FINDSYNC=-1 - ALL CHARACTERS PASSED TO IMP PACKAGE
!    INOUTSTAT=-1; FINDSYNC=N(>0) - TRANSMIT MODE. GENERATE N SYN'S.
!    INOUTSTAT=-1; FINDSYNC=0  - CALL IMP FOR NEXT OUTPUT CHAR.
!          THESE ARE THE 'DATABASE' VARIABLES, THEY ARE INITIALISED
!                SERIALLY AND SHOULD REMAIN A SINGLE GROUP.
INOUTSTAT       DC     0
FINDSYNC DC     0
CHAR     DC     0
LASTCHAR DC     0
STATUS   DC     0
TIMECOUNT DC 0
SYNCSHIFT DC 0
CHANN    DC     0
LINK9 DS
DTAONE DS
MASTER DS
ERMSTR DS
CBSAVE DS
CBFPTR DS
LINK5 DS
SAVEA DS
LINK DS
LINK2 DS
DEDPTR    IND   YB:DEDPAGE
SYNTABPTR IND M:SYNTABLE+7
SYSSTATUS DC     0          SYSTEM STATUS REPORT
SYN EQU 50
DEDPAGE  EQU   256
```

[cont'd]

```
!      DATA (1.5MS) INTERRUPTS
! THIS ROUTINE SERVICES THE CIRCULAR BUFFER AND PASSES
!   MASTER WORDS TO ERCCBUFF SINGLY.
 ENTPT                                     ENTRY POINT
          LDB    Y:CBFPTR             DISCOVER LAST C.B. LOCN. SERVICED.
AAGN      LDA    L:0
          EXC    I:DEDPTR             !EXTRACT CONTENTS OF NEXT C.B. LOCN.
          TSTL   A=0                 !CHECK FOR PRESENCE OF MASTERWORD
          JMP    CBSEND;              !NO MASTERWORD.CEASE SERVICING C.B.
          STB    Y:CBFPTR;            !SAVE CIRCULAR BUFFER POINTER.
          SRE    BUFFER              !CALL CHARACTER ROUTINE
          LDB    Y:CBFPTR             !RECALL C.B. POINTER
          LDA    L:127
          CPY    ASUBB
          TSTL   A=0
          LDB    L:239;               !MOVE TO SECOND HALF OF C.B.
          LDA    L:255;               !CHECK FOR END OF 2ND HALF OF C.B.
          CPY    ASUBB
          TSTL   A=0
          LDB    L:111;               !MOVE TO START OF FIRST HALF OF C.B.
          ADB    L:1
          JMP    AAGN
CBSEND    STB    Y:CBFPTR;            !SAVE C.B. POINTER.
          LDA    TIMECOUNT
          TSTL   A=0
          JMP    TIMESTOPPED
          SBA    L:1
          TSTL   A=0
          JMP    REPORTTIMEOUT
          STA    TIMECOUNT
TIMESTOPPED EQU .
          LDB    MPXRDEDLOCS
          ADB    L:8
          ENDINT                              RE-PERMIT THE INTERRUPT
 DEBUG
          DROPOUT                             HAND CONTROL BACK TO E2.
 DEBUG
          DEBUG
REPORTTIMEOUT STA TIMECOUNT
          LDA    L:2
          STA    STATUS
!         CODE HERE TO CALL ANALYSESTATUS.READSTATUS WILL FIND
!             STATUS=2,TIMEOUT.
          LDA    L:ANALSTATUS
          STA    Y:IMPROUTINE
          SRE    IMPLINK
          JMP    TIMESTOPPED
!
BUFFER    DC     ERCCBUFF
```

[cont'd]

```
ERCCBUFF  EQU    .
          STB    Y:LINK2        SAVE RETURN LINK
          STA    Y:MASTER         AND CIRCULAR BUFFER ENTRY.
          SFTL   S,A,L,1        ISOLATE CHANNEL NUMBER (0=>107) AT
          SFTL   S,L,R,9           BOTTOM OF A.
          STA    Y:CHANN        SAVE IT
          CPYL   B=A           MOVE TO B
          LDA    Y:MASTER       TEST FOR INPUT OR OUTPUT INTERRUPT
          TSTL   A<0            IF INPUT INTERRUPT
          JMP    TSTINPUT          GO TO TEST SOFTWARE STATUS.
          LDA    Y:INOUTSTAT      IF OUTPUT INTERRUPT,
          TSTL   A<0               AND SOFTWARE IS OUTPUTTING
          JMP    OUTSTAT               GO TO DEAL WITH INTERRUPT.
          LDA    OUTPAD          ELSE OUTPUT A PAD CHARACTER.
          STA    IY:DEDPTR        STORE PAD INTO DEDLOC
          LDP    Y:LINK2            AND RETURN.
OUTSTAT   EQU    .
         LDA   Y:FINDSYNC       ARE WE SENDING LEADING SYN'S
          TSTL   A=0            IF ZERO, REAL DATA OUTPUT
          JMP    TXDATA         GO TO ENTER JOB ON OUTPUT Q.
          SBA    L:1            REDUCE SYN COUNT
         STA   Y:FINDSYNC          AND RESTORE FOR NEXT ENTRY
          LDA    OUTSYN         LOAD A SYN
          STA    IY:DEDPTR        CHARACTER INTO DEDLOC.
          LDP    Y:LINK2        RETURN
TXDATA    EQU    .
!        ENTER TRANSMIT,
!        WRITEDATA WILL PLACE NEXT OUTPUT CHARACTER IN DEDLOC
!            INDICATED BY 'CHANN'.
         LDA   L:TRANSMIT
         STA   IMPROUTINE
         SRE   IMPLINK
          LDP    Y:LINK2        !THEN RETURN.
OUTSYN    EQU    .
          DC     B'1100000100110010' IBM EBCDIC SYN
!
```

[cont:d]

```
TSTINPUT  EQU    .
         LDA    Y:INOUTSTAT       TEST SOFTWARE I/O STATUS
          TSTL   A<0,A=0        IF OUTPUT, OR IDLE
          LDP    Y:LINK2           THEN IGNORE THIS INPUT INTERRUPT
          LDA    Y:MASTER        SELECT
         LDB    L:255             DATA PORTION
          CPYL   A=B.AND.A             OF MASTER WORD.
         STA    Y:CHAR                     AND SAVE IN CHAR
          SFTL   S,A,L,8        SHIFT TO TOP OF WORD
         ADA    LASTCHAR             INCLUDE PREVIOUS 8 BITS
         STA    LASTCHAR               AND SAVE ALL 16 BITS.
         LDA    Y:FINDSYNC        ARE WE SYNC SEARCHING?
          TSTL   A<0,A=0
          JMP    INSYNC         NO - GO TO OBTAIN CHARACTER
         LDA    Y:LASTCHAR        LOAD LAST 16 BITS
          LDM    L:0            SET UP LOOP COUNT
          SBM    L:7               OF 8.
TESTSYN   EQU    .
          LDB    IY:SYNTABPTR NOW SCAN THE SYN TABLE
          CPYL   B=B.NEV.A      TEST IF A=B.
          TSTL   B=0            IF B=0 THEN SYNC HAS BEEN FOUND
          JMP    FOUND          GO TO COMPUTE THE PHASE SHIFT
          TSTL   M=0,IM         ELSE TEST LOOP COUNT
          JMP    OUT                AND EXIT
          JMP    TESTSYN               OR TRY AGAIN.
SYNTABLE  EQU    .
          DC     B'0011001000110010' M=-7 SHIFT=8
          DC     B'0001100100011001' M=-6 SHIFT=7
          DC     B'1000110010001100' M=-5 SHIFT=6
          DC     B'0100011001000110' M=-4 SHIFT=5
          DC     B'0010001100100011' M=-3 SHIFT=4
          DC     B'1001000110010001' M=-2 SHIFT=3
          DC     B'1100100011001000' M=-1 SHIFT=2
          DC     B'0110010001100100' M=0  SHIFT=1
FOUND     EQU    .
          LDA    SHIFTCON       SYNC FOUND - CALCULATE PHASE SHIFT
          CPY    A=AMINUSM       FORM SHIFT CONSTANT
         STA    Y:SYNCSHIFT         AND SAVE
          LDA    L:0
         STA    Y:FINDSYNC          FINDSYNC=0 TO INDICATE SYNC FOUND
OUT       EQU    .
         LDA    Y:CHAR              MAKE CURRENT 8 BITS
         STA    Y:LASTCHAR          PREVIOUS 8 BITS FOR NEXT IME
          LDP    Y:LINK2        THEN RETURN
```

[cont'd]

```
!CHARACTER PHASE HAS BEEN FOUND. THROW AWAY LEADING SYN'S AND PASS
!    GENUINE DATA CHARS TO HANDLER.
INSYNC     EQU    .
          LDA    Y:LASTCHAR           LOAD LAST 16 BITS FROM LINE
          LDB    L:255
          SFT    SYNCSHIFT            APPLY SHIFT TO OBTAIN REAL CHAR.
           CPYL    A=B.AND.A    REMOVE UNWANTED BITS
          EXC    Y:CHAR               SAVE REAL CHAR,MAKE CURRENT8 BITS
          STA    LASTCHAR             PREVIOUS 8 BITS FOR NEXT TIME.
          LDA    Y:FINDSYNC           ARE WE SKIPPING LEADING SYNS?
           TSTL    A<0          IF NOT,
           JMP     QCHAR           GO TO Q THE CHARACTER.
          LDA    CHAR                        ELSE TEST FOR A SYN
           SBA     L:SYN
           TSTL    A=0          IF SYN FOUND
           LDP     Y:LINK2         THEN RETURN
           LDA     L:0           ELSE
           SBA     L:1            SET
          STA    Y:FINDSYNC           FINDSYNC=-1 TO INDICATE REAL DATA
QCHAR      EQU    .
!         ENTER RECEIVE,
!         READDATA WILL TAKE CHARACTER FROM 'CHAR'.
          LDA    L:RECEIVE
          STA    IMPROUTINE
          SRE    IMPLINK
           LDP     Y:LINK2      !RETURN
MASKFF00   EQU    .
           DC      B'1111111100000000'
SHIFTCON   EQU    .
           SFTC    S,L,R,1      SHIFT CONSTANT FOR CHARACTER PHASE.
A=AMINUSM  EQU    .
           CPYC    A=A-M
          .Y-AREA
IMPLINK   DC     ENTERIMP
IMPSAVE   DC     0
IMPROUTINE DC    0
W=B       CPYC   W=B
           X-AREA
ENTERIMP  STB    IMPSAVE               SAVE RETURN LINK.
          LDW    Y:WSAVE              LOAD CURRENT STACK POINTER.
          LDM    IMPROUTINE           LOAD INDEX OF IMP ROUTINE.
          ENT    IY:PVECT             ENTER IMP ROUTINE
          LDP    IMPSAVE               RETURN
W=APLUSW  CPYC   W=A+W
RECEIVE   EQU    0          ROUTINE INDEX OF IMP 'RECEIVE' ROUTINE.
TRANSMIT  EQU    2              IMP 'TRANSMIT' ROUTINE
ANALSTATUS EQU   4              IMP 'ERROR' ROUTINE.
```

[cont'd]

```
!
! HARDWARE INTERFACE SOFTWARE FOR MODULAR ONE WITH 1.61 MPXR.
!
!  INTERRUPT CODE - SEE PREVIOUS LISTING.
!
! HARDWARE CONTROL ROUTINES.
!
%INTEGERFN READDATA
*'          LDA    Y:CHAR'        ;!LOAD CHAR PLANTED BY ASSEMBLER
*'          LDP    W:0'           ;!RETURN
%END
%ROUTINE WRITEDATA(%INTEGER CHAR)
%OWNINTEGER TOPOPBITS=X'C100'
*'          LDA    W:'CHAR        ;!LOAD CHARACTER TO BE OUTPUT
*'          LDB    Y:CHANN'       ;!LOAD DEDLOC ADDRESS
*'          ADA    Y:'TOPOPBITS   ;!ADD FIXED BITS TO CHARACTER
*'          STA    I:DEDPTR'      ;!  AND PLACE IT IN THE DEDLOC
%RETURN
%END
%INTEGERFN READSTATUS
*'          LDA    Y:STATUS'      ;!FETCH STATUS SET BY ASSEMBLER
*'          LDP    W:0'           ;!RETURN
%END
%ROUTINE WRITECONTROL(%INTEGER FUNCTION)
%INTEGER SYNSYN
%SWITCH SW(0:8)
%OWNINTEGER OUTBITS=X'C003'
%OWNINTEGER TWOSECS=1400
%INTEGER SYSSTATUS
-> SW(FUNCTION)
SW(0) :                           ;!SETSYN
SYNSYN=SYN<<8!SYN                 ;!FORM DOUBLE-SYN PATTERN
*'          LDB    L:7'           ;!SET COUNT OF 8
*'          LDA    W:'SYNSYN      ;!LOAD SYN-SYN
*'SETSYN    SFTL   S,L,L,1'       ;!ROTATE PATTERN 1 LEFT
*'          STA    YB:SYNTABLE'   ;!STORE ENTRY IN SYN SCAN TABLE
*'          TSTL   B=0'           ;!8 ENTRIES FILLED?
*'          JMP    SSOUT'         ;!YES - EXIT
*'          SBB    L:1'           ;!DECREMNET BY 1.
*'          JMP    SETSYN'        ;!DO NEXT TABLE ENTRY
*'SSOUT     EQU    .'             ;!
%RETURN
!
```

[cont'd]

```
SW(1) :                                 ;!ENTERRX
*'          LDA    L:1'                 ;!SET SOFTWARE INDICATORS
*'          STA    Y:INOUTSTAT'         ;!  TO 1
*'          STA    Y:FINDSYNC'          ;!     FOR INPUT MODE
%RETURN
!
SW(2) :                                 ;!ENTERTX
*'          CPYL   A=-1'                ;!SET SOFTWARE INDICATOR TO -1
*'          STA    Y:INOUTSTAT'         ;!  FOR OUTPUT MODE.
*'          LDA    L;6'                 ;!SET LEADING SYN COUNT
*'          STA    Y:FINDSYNC'          ;!  TO 6.
*'          LDA    Y:'OUTBITS           ;!SET SUITABLE PATTERN
*'          LDB    Y:PLEXPAGE'          ;!  INTO ALL 4
*'          STA    YB:0'                ;!     OUTPUT DEDLOCS
*'          STA    YB:1'                ;!        TO MAKE SURE
*'          STA    YB:36'               ;!           THEY ARE
*'          STA    YB:37'               ;!              ACTIVE.
%RETURN
!
SW(3) :                                 ;!STARTTIMER
*'          LDA    Y:'TWOSECS           ;!SET INTERRUPT COUNT TO
*'          STA    Y:TIMECOUNT'         ;!  TIME 2 SECONDS
%RETURN
!
SW(4) :                                 ;!RESETCHANNEL
*'          LDA    L:0'                 ;!SET ALL
*'          STA    Y:TIMECOUNT'         ;!  SOFTWARE
*'          STA    Y:INOUTSTAT'         ;!     INDICATORS
*'          STA    Y:FINDSYNC'          ;!        TO ZERO.
%RETURN
!
SW(5) :                                 ;!PARITY CHECKING
SW(6) :%STOP                            ;!  NOT USED FOR IBM CODES.
!
SW(7) :                                 ;!ENABLE INTERRUPTS
SYSSTATUS=X'C'                          ;!SET MPXR STATUS FOR INTERRUPTS ON
*'SETSTAT   LDA    MPXRCHAN'            ;!NOW SEND STATUS TO MULTIPLEXOR
*'          CHACCESS'                   ;!POINT Z-AREA AT MPXR
*'          DEBUG'                      ;!IF REJECTED
*'          LDB    FIVE12'              ;!LOAD FIXED ADDRESS BITS
*'          LDA    W:'SYSSTATUS         ;!LOAD STATUS
*'          STA    ZB:4'                ;!SEND STATUS TO MPXR.
*'          DEBUG'                      ;!IF REJECTED
*'          CPYL   A=-1'                ;!NOW DO CHACCESS WITH -1
*'          CHACCESS'                   ;!  TO RESTORE Z-AREA.
*'          DEBUG'                      ;!IF REJECTED
%RETURN
!
SW(8) :                                 ;!DISABLE INTERRUPTS
SYSSTATUS=X'2C'                         ;!SET MPXR STATUS FOR INTERRUPTS OFF
*'          JMP    SETSTAT'             ;!  AND SEND IT TO MPXR.
%END
```

With these hardware interfacing routines, it is therefore possible to make the input channel appear to conform to the behaviour defined for the idealized hardware interface.

Data Output

When the software wishes to perform an output transfer, it calls the WRITECONTROL routine with an 'enter transmit mode' command. This is interpreted by setting appropriate software flags and placing a short sequence of 1-bits in the reserved multiplexor register. When the multiplexor hardware has sent this bit pattern to the line, it makes an entry in the circular buffer requesting more output on that channel. When this entry is detected by the interrupt routine scanning the circular buffer, it can examine the software flags to see what action is required. Normally, these will be set to indicate output of the appropriate number of leading SYN characters, which can be generated directly by the circular buffer scanning routine. When the requested number of SYN characters has been generated, the first real character of the message is required.

At this point the interrupt routine can call the TRANSMIT routine which will eventually call the WRITEDATA routine with the character to be output. The WRITEDATA routine will place the character in the multiplexor register reserved for this output channel. This will eventually generate another circular buffer entry so that the process can continue until the whole message is output.

The WRITECONTROL routine will then be called to cancel transmit mode, which will be interpreted as setting the relevant software flags so that the circular buffer routine will ignore any more entries for that output channel.

Therefore, the output channel can also be made to conform to the desired interface by simple interfacing routines.

Since the multiplexor synchronous channel is able to work in full-duplex mode, the software interfacing routines must ensure that receive and transmit modes are mutually exclusive as far as the main software body is concerned. Software flags must be set by the various WRITECONTROL functions so that the circular buffer scanning routine can ignore input circular buffer entries when the software is in output mode, and vice versa. There is no mechanism in the hardware by which input data can be ignored. This must be handled by the software.

Error reports

The error interface, defined by the ANALYZESTATUS routine, must also be accommodated so that effective error control can be achieved. The most important error monitor is the timeout control. Since the multiplexor provides a regular interrupt at a fixed time interval it is possible to use an interrupt count as a means of implementing a timer facility. The STARTTIMER function sets a storage location to a positive number equivalent to the interrupt count for the required time interval.

The circular buffer scan routine, which is run once per interrupt, decrements this count by one if it is positive. If the count reaches zero, a storage location reserved for holding the status report is set to indicate timeout and the ANALYZESTATUS routine is called. This will call READSTATUS which retrieves the current status report from the reserved location. The multiplexor hardware generates other interrupts for conditions such as LOSTCARRIER and DATA OVERRUN, and these can be mapped onto the ANALYZESTATUS routine in a similar manner.

Thus, it can be seen that it is possible to reproduce all the desired communications hardware characteristics on the 1.61 multiplexor even though the actual hardware is radically different from the idealized communications channel proposed.

The first-level interrupt-handling routines, needed to map from the real interrupt structure onto the conceptual interrupt structure, although more complicated than on any of the other small computers, still did not require very many instructions (about 120 machine instructions). Also, the hardware control routines, by which the software issues instructions to the hardware, were very simple and mostly just involved communication with the first-level interrupt routines through common variables.

The coding for all the hardware interfacing software is given above.

Executive Interface

Because the Modular One runs with a permanently resident executive, in this case E2[21], and all interrupts cause direct entry to the executive, some initialization code was necessary to set up the interrupt links. This was done by means of the E2 LINK facility, which is an executive call used to LINK a user program to a particular interrupt. An executive call was necessary because the Modular One executive store area is completely protected from user programs, and so it is not possible to modify executive tables directly. The executive itself included a facility to reset the interrupt links to a neutral value if the user program terminated for any reason, which could only be done through the executive.

It was also necessary to perform some initialization of the multiplexor hardware to ensure that it was in the correct mode, and this also required a special executive facility known as CHACCESS, since physical control of peripherals could only normally be done by the executive.

Because of the nature of the Modular One hardware and software, the initialization code was more extensive than on any other system.

User Interface

Both the communications package and the user program for the Modular One were written in IMP and so the user interface routine were invoked by standard IMP routine calls and parameters were passed through global variables.

10.6 PDP-11 with DP 11 communications controller

Hardware Interface

The DP 11 synchronous controller (22) is a single-line full-duplex communications controller with hardware detection of incoming SYN characters, the particular SYN character being set by software.

It is possible to separately enable and disable the input and output channels, although with a complication in connection with output noted later. There is no hardware timer facility, but there are error interrupts associated with modem conditions such as LOST CARRIER.

Data Input

If the input interrupt is enabled, the receive channel will start to generate input character interrupts as soon as character synchronization is achieved unless synch stripping is selected. This is a facility to remove all SYN characters from input data. If this is used, the facility must be disabled once the message starts as the SYN pattern can occasionally occur in real data. If the facility is not used, leading SYNs can be removed by the first-level interrupt routines. Once the first non-SYN character is seen, a flag is set to inhibit any further SYN removal and all characters are then passed to the RECEIVE routine for message analysis. Therefore, after the initial stage all input interrupts can be routed directly to the RECEIVE routine, matching the conceptual interface immediately.

```
;
; HARDWARE INTERFACE SOFTWARE FOR PDP-11 WITH DP11 CONTROLLER.
;
; INTERRUPT VECTOR SERVICE ROUTINES.
;
; RECEIVE INTERRUPT
;
REVECT:  JSR    R5,SAVE         ;SAVE REGISTERS
         MOVB   RB,VECBUF       ;FETCH CHAR FROM RECEIVE BUFFER
         TSTB   INOTST          ;TEST FOR INPUT MODE
         BLE    VECRET          ;IF NOT, IGNORE THIS INTERRUPT
         BIC    #1,RSR          ;DISABLE STRIP SYNC IN DP11
         JSR    R5,RCEIVE       ;CALL RECEIVE ROUTINE
VECRET:  JSR    R5,REST         ;RESTORE REGISTERS
         RTI                    ;RETURN TO INTERRUPTED PROGRAM
;
; TRANSMIT INTERRUPT
;
TRVECT:  JSR    R5,SAVE         ;SAVE REGISTERS
         TSTB   INOTST          ;TEST FOR OUTPUT MODE
         BGE    NOTX            ;IF NOT, OUTPUT A PAD
         TSTB   LEADSN          ;TEST IF LEADING SYNS SENT
         BEQ    CALLTX          ;YES - CALL TRANSMIT
         DECB   LEADSN          ;REDUCE LEADING SYN COUNT BY 1
         MOVB   #SYN,TB         ;MOVE SYN TO TRANSMIT BUFFER
         BR     TXRET           ;RETURN
CALLTX:  JSR    R5,TRNSMT       ;CALL TRANSMIT ROUTINE
         BR     TXRET           ;RETURN
NOTX:    MOVB   #PAD,TB         ;MOVE PAD TO TRANSMIT BUFFER
TXRET:   JSR    R5,REST         ;RESTORE REGISTERS
         RTI                    ;RETURN TO INTERRUPTED PROGRAM.
;
; REAL-TIME CLOCK INTERRUPT.
;
TIMVEC:  JSR    R5,SAVE         ;SAVE REGISTERS
         TST    LKS             ;READ CLOCK REGISTER
         DEC    TIMCNT          ;DECREMENT INTERRUPT COUNT
         TST    TIMCNT          ;TEST IF COUNT LAPSED
         BNE    TIMRET          ;IF NOT, RETURN
         BIC    #100,LKS        ;OTHERWISE,DISABLE CLOCK INTERRUPT
         MOVB   #N2,STATUS      ;SET STATUS REPORT TO TIMEOUT
         JSR    R5,ANALST       ;CALL ANALYZESTATUS
TIMRET:  JSR    R5,REST         ;RESTORE REGISTERS
         RTI                    ;RETURN TO INTERRUPTED PROGRAM
```

[cont'd]

```
;
; HARDWARE CONTROL ROUTINES
;
RDDATA:    MOVB   VECBUF,RO       ;FETCH INPUT CHARACTER TO RO
           BIC    #177400,RO      ;REMOVE ANY EXTENDED BITS
           RTS    R5
WRDATA:    MOVB   RO,TB           ;MOVE CHAR TO TRANSMIT BUFFER
           RTS    R5
RDSTAT:    MOVB   STATUS,RO       ;FETCH STATUS REPORT
           RTS    R5
WRTCTL:    ASL    RO              ;MULTIPLY FUNCTION BY 2.
           JMP    @TABLE(RO)      ;  AND SWITCH ON FUNCTION.
           .EVEN
TABLE      .WORD SETSYN
           .WORD ENTRX
           .WORD ENTTX
           .WORD STIMER
           .WORD RESETC
           .WORD SETPAR
           .WORD USPAR
           .WORD ENINTS
           .WORD DISINT
SETSYN:    MOVB   #SYN,SR         ;LOAD SYN REGISTER IN DP11
           RTS    R5              ;RETURN
ENTRX:     MOVB   #N1,INOTST      ;SET SOFTWARE INDICATOR TO INPUT
           BIC    #4200,RSR       ;CLEAR ACTIVE AND DONE IN RX STATUS REG.
           BIS    #101,RSR        ;SET BITS TO ENABLE RECEIVE CHANNEL
           RTS    R5              ;RETURN
ENTTX:     MOVB   #M1,INOTST      ;SET SOFTWARE INDICATOR TO OUTPUT
           MOVB   #N6,LEADSN      ;SET LEADING SYN COUNTER
           RTS    R5              ;RETURN
STIMER:    MOV    TWOSEC,TIMCNT;SET CLOCK COUNTER FOR 2 SECONDS
           BIS    #100,LKS        ;ENABLE CLOCK INTERRUPT
           RTS    R5              ;RETURN
RESETC:    BIC    #100,LKS        ;DISABLE CLOCK INTERRUPT
           BIC    #4300,RSR       ;DISABLE RECEIVE CHANNEL
           CLR    INOTST          ;CLEAR SOFTWARE INDICATOR
           CLR    TIMCNT          ;CLEAR CLOCK COUNTER
           RTS    R5              ;RETURN
SETPAR:    HALT                   ;PARITY NOT USED
USPAR:     HALT                   ;  ON IBM CODES.
ENINTS:    BIS    #100,LKS        ;ENABLE CLOCK INTERRUPT
           BIS    #100,RSR        ;  RECEIVE INTERRUPT
           BIS    #100,TSR        ;     TRANSMIT INTERRUPT.
           RTS    R5              ;RETURN.
DISINT:    BIC    #100,LKS        ;DISABLE CLOCK,
           BIC    #100,RSR        ;  RECEIVE,
           BIC    #100,TSR        ;     TRANSMIT INTERRUPTS.
           RTS    R5              ;RETURN
SYN=62
PAD=377
STATUS:    .BYTE 0
INOTST:    .BYTE 0
LEADSN:    .BYTE 0
TIMCNT:    .WORD 0
TWOSEC:    .WORD 144
```

At the end of the message, the receive channel can be disabled and input interrupts inhibited to prevent any further entries to the RECEIVE routine. The first-level interrupt code is therefore minimal for the DP 11.

Data Output

Data output is started in the DP 11 by loading a character into the transmit buffer register. The DP 11 transfers this to the transmit shift register and raises 'Request to Send' to the modem. The DP 11 also starts to shift out the data from the transmit shift register in time with the transmit clock pulses from the modem. As the first bit of each character is shifted out to the line, the DP 11 requests an interrupt for the software to re-fill the transmit buffer register. The software must respond to this before the current character has been shifted out to the line, otherwise the DP 11 will cancel 'Request to Send' and cease transmitting. The interrupts occur irrespective of whether the modem has responded with the 'Ready for Sending' signal. Since any data transmitted before 'Ready for Sending' is set cannot be guaranteed to be transmitted correctly, the software must respond to output interrupt requests by outputting dummy characters until it detects that 'Ready for Sending' has been set. This signal can be examined by the software in the transmit status register. The output channel is then available for real data output.

In 4-wire mode, where continuous carrier can be maintained even though no data is being transmitted, this sequence need be performed only once as part of initialization in the INIT routine. To maintain carrier, the first-level output interrupt routine must arrange to output idle-mark (all - 1) characters whenever the package is not in transmit mode. Control over this is accomplished through a software flag which is set and reset by the 'enter/cancel transmit mode' calls on WRITECONTROL respectively.

In 2-wire mode, the sequence to start the transmit channel must be executed each time the command 'enter transmit mode' is given. At the end of a transmission, carrier is dropped by not outputting a character in response to an output interrupt request.

The transmit channel has an option for generating SYN characters automatically. If the 'idle sync' bit is set in the transmit status register, and the software does not load a character into the transmit buffer in time, the hardware copies the contents of the SYNC register into the output shift register for transmission. This feature is not much use however, since there is no simple means of controlling the number of SYNs transmitted in this way. Leading SYNs must therefore be transmitted by the first-level output interrupt routine, using a count set by the WRITECONTROL coding for 'enter transmit mode'. Subsequent output interrupts are then routed to the TRANSMIT routine as before until 'cancel transmit mode' is executed.

The first-level output interrupt routine therefore involves slightly more coding than the corresponding input routine, although it is still quite simple.

Error Reports

Since there is no hardware timer facility on the DP 11, the essential timeout function must be implemented using the CPU real-time clock. An interrupt count can be accumulated from this regular interrupt to produce the necessary timeout control.

The STARTTIMER function sets the count to the appropriate value and when the count is decremented to zero by the clock interrupt routine, the ANALYZESTATUS routine is called after setting a status variable to indicate timeout. The READSTATUS routine accesses this variable to obtain the current status report.

Error interrupts generated by other conditions such as LOST CARRIER and DATA OVERRUN are sent to the ANALYZESTATUS routine by similar routes.

The four hardware control routines use a combination of directly accessing the communications hardware registers and setting variables common to the first-level interrupt routines in order to achieve the desired hardware effects. All four routines are very simple.

The necessary hardware interfacing software for the DP 11 is thus fairly simple, mainly because the hardware is itself character-oriented.

Executive Interface

Two different executives have been used to support the communications system. The minimal executive IOX has been used in simple systems, and the executive source code was included in with the communication system to produce a stand-alone program so that interrupt links were set permanently at compile time. The disc-based executive DOS(23) has been used to support a disc-oriented communication system. DOS is normally resident, but the executive store area is not protected so the necessary interrupt links could either be compiled into the program or set dynamically when the program is first run. Neither method allows of very easy restoring of the links to a safe value at the end of a communication session, but since the executive is loaded from disc and does not support multi-programming, it is probably safesttto reload the executive after each session.

User Interface

Two versions of the communication system (communications package plus user program) have been produced - one totally in assembler and the other totally in IMP, so there were no inter-language interfacing problems and all user interface routine calls were standard for the language used with parameters passed through global variables.

10.7 Conclusions

These detailed descriptions of the five major versions of the communications package produced so far illustrate the considerable differences in the environments in which the package has been applied. Despite these considerable differences, no particular problems were encountered in the implementations and all five systems are in regular use. Once the necessary thought had been given as to how to map from the real interfaces onto the conceptual interfaces, everything that followed was fairly mechanical process, requiring only careful attention to detail to produce a working system.

A number of other systems involving different small computers, such as the NOVA and INTERDATA, have been tentatively investigated and there would seem to be no major problems in applying the communications package to these as well. So the techniques developed to produce this easily transferable system do seem generally applicable and no limitations are as yet apparent.

Chapter 11

IMPLEMENTING THE COMMUNICATIONS SYSTEM ON A NEW SMALL COMPUTER

11.1 Introduction

As has been stated previously, the original idea behind the development of the communications package was to produce a system that could be easily transferred to a new small computer. An earlier chapter described how the communications package was written in terms of certain standard interfaces as a means of achieving this ease of transferability. Inside these interfaces, everything remains the same for different implementations. Outside these interfaces a minimal set of routines is needed to map between the idealized standard interfaces of the package and the real environment. The previous chapter indicated in some detail the particular interfacing routines that had been produced for the five major versions developed so far.

This chapter will attempt to demonstrate the ease of transferability of the communications package by describing a series of steps to be followed in producing a new implementation. The amount of fresh thought required for a new system can be reduced to a minimum by following prescribed test procedures for the different component parts before putting the complete system together and testing it. The idea is to 'mechanize' the process of software production as far as possible by making use of work already done on previous systems.

11.2 General

The overall structure of the communication system has been indicated previously. This structure can be summarized by the diagram of Figure 11.1.

Each component in the system is self-contained, with a well-defined interface to the other components. It should be possible to develop and test each component independently to see if it behaves according to the defined interface. The order of testing is not normally significant and testing can proceed in parallel for the different components.

11.3 Difficulty of 'Live' Testing

In any communication system, 'live' testing, using a real communication line with an appropriate terminal or computer at the other end of the line, can be very difficult during the development stage because data is being transferred at a speed far in excess of that which can be observed by a human being. Events are occurring in real-time and it is not generally possible for the programmer to slow down the events to a convenient speed. For this reason, as much testing and development as possible should be done in a non-real-time environment with any real-time events being simulated under programmer control such that the passage of time is not critical. The other problem about 'live' testing is that it is practically impossible to control the other end of the link effectively or even to tell exactly what the other end is doing, even when voice communication can be established. The ideal first stage of 'live' communication testing should be with both ends of the link in the same room

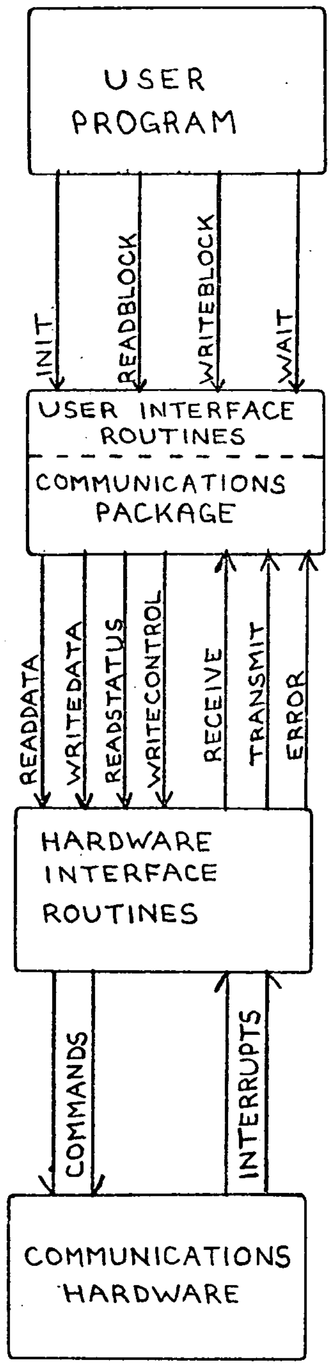


FIGURE 11.1 SYSTEM STRUCTURE AND INTERFACES

connected by a modem simulator, which replaces both modems and the line. The person doing the testing then has both ends under his control and can ensure that both ends are doing the right things.

However, before this stage of 'live' testing is reached, the major software components can be checked out in a non-real-time environment. The hardware and hardware interface routines can also be checked out in a simplified environment. These test procedures are described in the following sections.

11.4 User Program

The user program component is the one least likely to be transferable from one system to another. The user program has responsibility for local-peripheral handling, operator control, etc. The implementation of this is likely to vary considerably from one system to the next, since the actual peripherals used are likely to be different and the facilities already provided by the standard executives are likely to vary widely.

The facilities required are basically of the sort 'READRECORD' and 'WRITERECORD', together with some facilities for operator communication such as 'OPERATORMESSAGE' and 'OPERATORREPLY'. At the one extreme, a sophisticated executive and I/O routine library will provide these facilities directly. At the other extreme, there is no standard executive, and all the facilities have to be programmed completely from scratch. The systems so far implemented have varied from the latter extreme to somewhere approaching the former.

In general, then, a certain amount of work will be necessary in order to provide the sort of facilities mentioned above for a new system. However, the facilities are reasonably standard, and any difficulties are likely to arise from peculiar characteristics of the peripherals or executive used rather than any conceptual difficulties.

Testing of User Program

The implementation details of the user program are not relevant here, since they are concerned with local-peripheral handling, operator control, etc. which are not directly relevant to the communications package. The only aspect of the user program of relevance is the interface which it presents to the communications package. This can be thoroughly checked out by substituting dummy test routines for the four communications user interface routines. These dummy routines can check parameters and monitor calls and allow the programmer to interact with them to perform debugging operations.

A dummy INIT routine would do very little other than note that it had been called. A dummy READBLOCK routine could check parameters and arrage for a specimen data buffer in the correct format to be passed back to the user by the subsequent call on the WAIT routine, The programmer could be given the facility to feed in different specimen buffers in order to test all possible formats. A dummy WRITEBLOCK routine could check parameters and allow the programmer to inspect the contents of the buffers to check that they were being correctly formatted.

It is obviously possible to check out all aspects of the operation of the interface in this way in a totally controlled fashion without having to rely on the other end of the link providing the correct buffers at the right time.

It is even possible to carry out this type of testing on a different computer if a suitable high-level language is used and a compatible set of local-peripheral routines is available.

11.5 Communications Hardware

The provision of the communications hardware component involves two possibilities:-

a) produce a version of the ERCC Communications Controller for the new computer

b) use the product provided by the computer manufacturer.

The first option should not be difficult to implement, since the ERCC Controller was specifically designed with this objective in mind. The contents of Chapter 6 plus the relevant ERCC engineering documents would be needed in order to carry this out. The second option depends on the availability of a suitable product. Most new small computers being produced now have a synchronous communications channel as a standard peripheral option. At the time the ERCC controller was originally developed, this was certainly not the case. Providing the manufacturer's product has the appropriate programming characteristics, then it is generally preferable to choose this option. 'Appropriate programming characteristics' does not impose any great constraints. It merely requires that the controller can operate in 8-bit binary mode, with the software handling the transfers one character at a time under interrupt control. All controllers so far investigated comply with these requirements. The problems of providing effective engineering support for the ERCC controller and the cost of producing small quantities of a new version are factors which weigh against it when there is a suitable controller as a standard product line item for the new computer. The tests described below are applicable to either option.

Testing of Communications Hardware

It is assumed that normal checkout of the communications hardware will be performed by engineering diagnostic programs provided with the hardware. However, most diagnostic programs test only the local operation of the hardware using special diagnostic functions and possibly special hardware test boxes used instead of the modem. It is not uncommon for the hardware to function correctly when tested in this mode, but not work correctly when tested in a real environment. It seemed desirable to devise some very simple test programs that used the hardware in a way similar to its real use by the communications package. It is also important for test programs to be so simple that it is obvious that they are correct.

The communication package generates output consisting of leading SYN characters followed by one or more message characters, and expects input in the same format. It also expects to use the timer facility, if this is incorporated into the communications controller.

A simple test program for the transmit channel therefore consists of the generation of a single-character output message, with leading SYN characters being provided by the hardware or software, as appropriate. For simplicity, this can be coded without using interrupts, if it is possible to drive the hardware in this way. Sample versions of this test program are given in the following pages for some of the systems in use. Using the timer facility, this output message can be repeated at fixed intervals of, say, one second.

A similarly simple program can be used for the input channel. This program waits for the input channel to signal that synchronization has been obtained and then prints the first non-SYN character on the teletype in some suitable binary notation, before re-enabling the receive channel and looping back to wait again. Some sample versions of this simple program are given on following pages.

These two programs can be used together end-to-end to test new or suspect communications hardware from a system that is known to work.

If the communications hardware can carry out both these tests correctly, then it indicates that the main logic of the transmit and receive channels is operating correctly and also that the modem interface is operational. If the communications hardware can handle the character sequence involved in the short messages, then it is probable that it will also handle long messages correctly. Faults in long messages after short messages have been transmitted correctly usually indicate transmission line problems and these can be tested by a further level of test programs which transmit data blocks consisting of a string of 8-bit characters in a binary progression from 0 to 255. The receiving program can check for any deviations from the binary progression as a means of detecting line errors.

A simple test program can also be used to test the hardware timer independently of any data transfer. This compares the timer interrupt interval with the number of times around a fixed instruction loop.

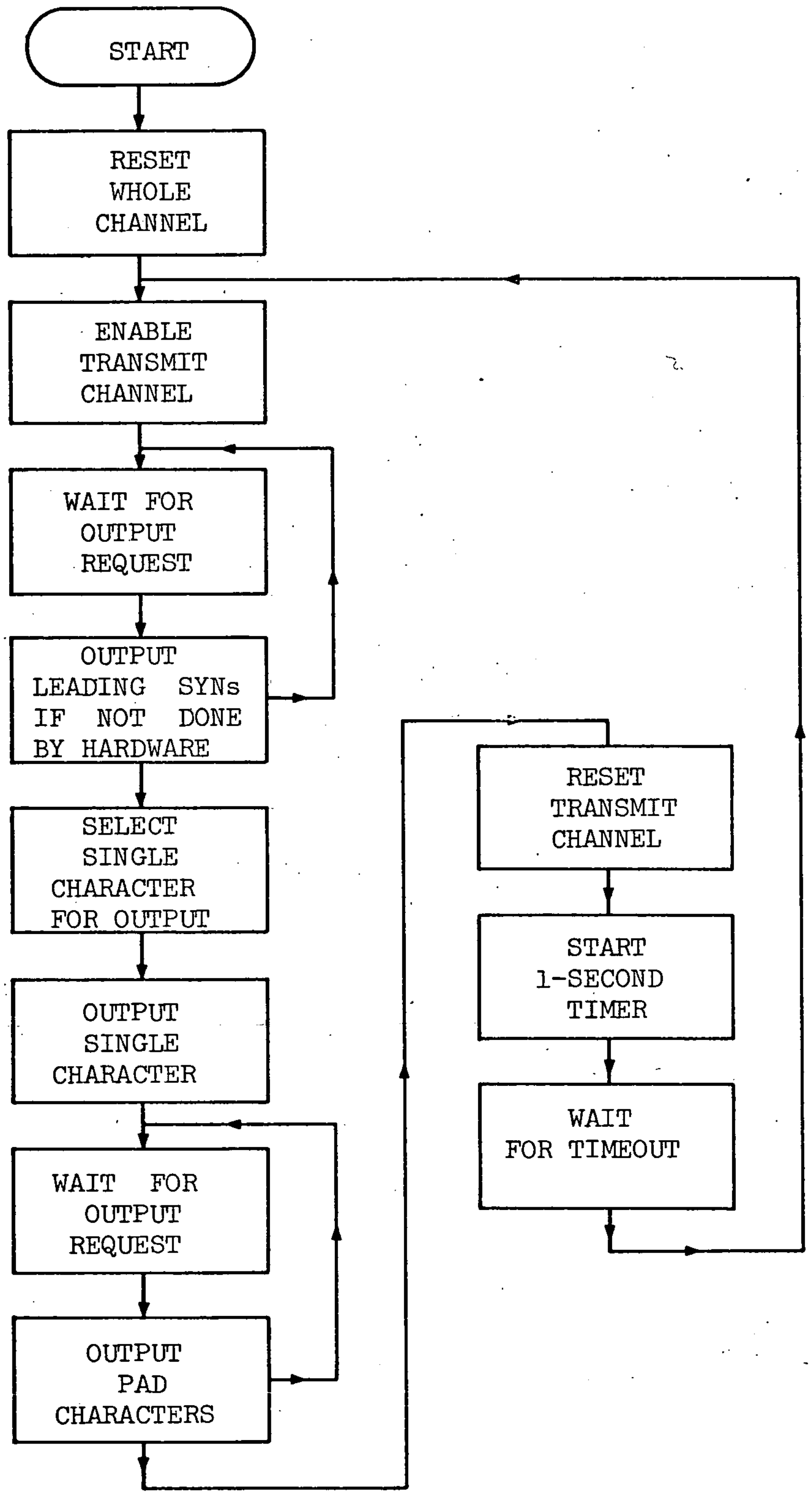
Flowcharts for these simple test programs are given on following pages.

11.6 Communications Package

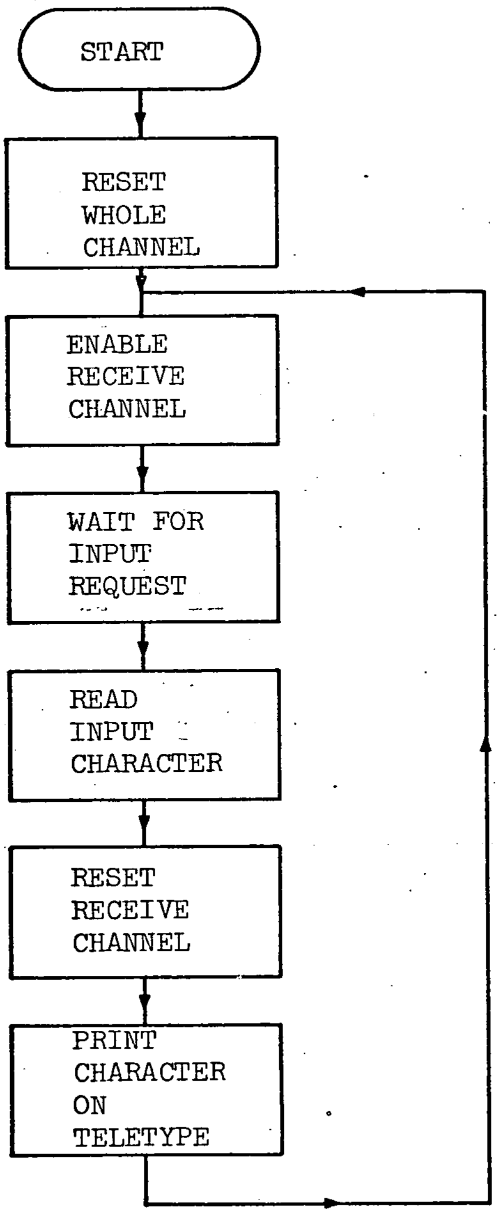
The procedure for making a new version of the communications package is quite simple. The amount of work involved is least when there is a suitable IMP compiler available for the new computer. 'Suitable' in this context means that the compiled program must be in a form which can be easily fitted into the assembler-coded environment. The environment must be able to reference routines in the IMP code and vice versa. The IMP code may also need to reference named variables in the assembler code. The simplest way of accomplishing this is for the compiler to produce symbolic assembler code output. This can then be easily combined with the hand-coded assembler prior to assembling the complete system. If the compiler produces binary output, that it must also provide suitable linkage information so that any cross references can be satisfied when the package is combined with the hand-coded software.

Flowcharts for simple test programs

Output test

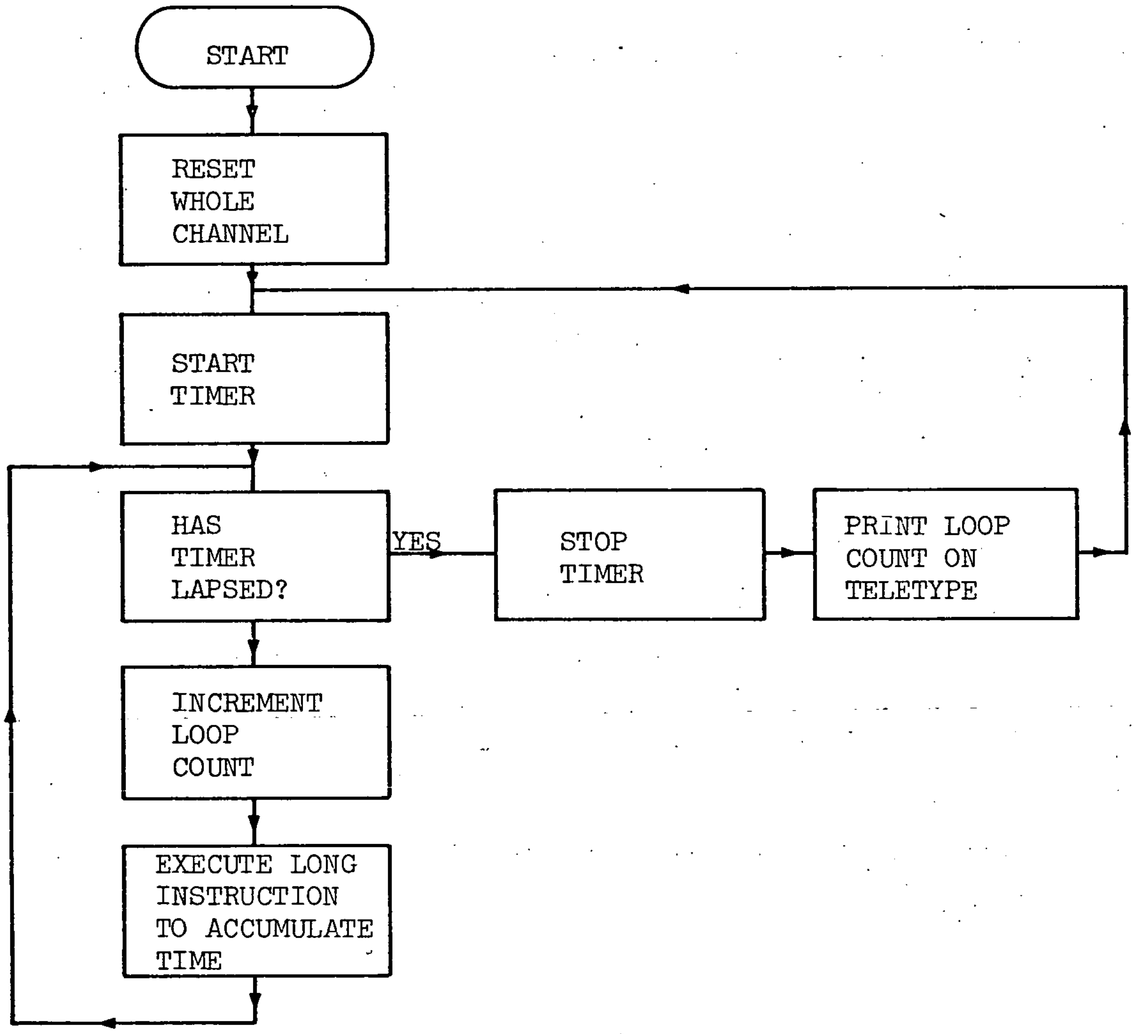


Input Test



START
RESET WHOLE CHANNEL
ENABLE RECEIVE CHANNEL
WAIT FOR INPUT REQUEST
READ INPUT CHARACTER
RESET RECEIVE CHANNEL
PRINT CHARACTER ON TELETYPE

Timer Test



START
RESET WHOLE CHANNEL
START TIMER
HAS TIMER LAPSED?
YES
STOP TIMER
PRINT LOOP COUNT ON TELETYPE
INCREMENT LOOP COUNT
EXECUTE LONG INSTRUCTION TO ACCUMULATE TIME

```
/ CODE FOR SIMPLE HARDWARE TESTS ON PDP-8 WITH ERCC CONTROLLER.
/ TEST OUTPUT OF SINGLE CHARACTER MESSAGE
START1,  CLA
         6304                   /ISSUE GENERAL RESET
         TAD    CODE            /SELECT
         6304                   /  CORRECT
         CLA                    /    CODE.
GO,      TAD    TSMIT           /LOAD COMMAND TO START TRANSMIT
         6304                   /ISSUE COMMAND. SYNS SENT AUTOMATICALLY
         6311                   /TEST FOR OUTPUT REQUEST FLAG
         JMP    .-1             /LOOP BACK IF NOT SET
         LAS                    /LOAD CHARACTER FROM SWITCHES
         6314                   /OUTPUT CHARACTER
         CLA
         TAD    PADCOUNT        /SET COUNTER TO
         CIA                    /  MINUS NUMBER OF
         DCA    COUNTER         /    PADS REQUIRED.
         TAD    PAD             /LOAD PAD CHARACTER
         6311                   /WAIT FOR
         JMP    .-1             /  OUTPUT FLAG.
         6314                   /OUTPUT PAD CHARACTER
         ISZ    COUNTER         /INCREMENT AND TEST PAD COUNT
         JMP    .-4             /LOOP BACK IF NOT DONE
         CLA                    /ISSUE GENERAL RESET
         6304                   /  TO CANCEL TRANSMIT MODE.
         TAD    STIMER          /START HARDWARE
         6304                   /  TIMER.
         CLA
TESTTIME,6301                   /WAIT FOR
         JMP    .-1             /  STATUS REPORT
         6302                   /READ STATUS REPORT
         TAD    TIMEOUT         /TEST FOR TIMEOUT
         SZA CLA
         JMP    TESTTIME        /TRY AGAIN IF NOT
         6304                   /OTHERWISE ISSUE GENERAL RESET
         JMP    GO              /  AND DO TEST AGAIN.
CODE,    2000                   /IBM EBCDIC TRANSMISSION CODE COMMAND
TSMIT,   2400                   /ENTER TRANSMIT MODE COMMAND
PADCOUNT,3                      /NUMBER OF PADS TO CLEAR REGISTERS
COUNTER, 0                      /LOOP COUNTER
PAD,     377                    /PAD CHARACTER
STIMER,  3400                   /COMMAND TO START HARDWARE TIMER
TIMEOUT, -1000                  /MINUS (TIMEOUT STATUS REPORT)
/
```

[cont'd]

```
/
/ TEST INPUT OF SINGLE CHARACTER MESSAGE.
START2,   CLA                           /ISSUE
          6304                          /  GENERAL RESET COMMAND TO CONTROLLER.
          TAD    CODE                   /SELECT APPROPRIATE
          6304                          /  TRANSMISSION CODE.
TESTIN,   CLA
          6311                          /WAIT FOR
          JMP    .-1                    /  INPUT FLAG.
          6312                          /READ INPUT CHARACTER
          DCA    CHAR                   /SAVE IT
          6304                          /  AND ISSUE GENERAL RESET.
          JMS    NEWLINE                /THEN
          TAD CHAR                      /  PRINT CHARACTER
          JMS    OCTPRINT               /     IN OCTAL ON NEWLINE.
          JMP    TESTIN                 /DO TEST AGAIN
CHAR,     0
/
OCTPRINT,0                              /OCTAL PRINT ROUTINE
          RAL                           /POSITION CHAR READY FOR LOOP
          DCA    OCTCHAR                /SAVE CHARACTER
          TAD    MINUS4                 /SET LOOP COUNTER
          DCA    COUNTER                /  TO PRINT 4 OCTAL DIGITS.
OCTDIG,   TAD    OCTCHAR                /LOAD CHAR OR RESIDUE
          RAL;RTL                       /SHIFT NEXT DIGIT TO BOTTOM OF AC
          DCA    OCTCHAR                /SAVE NEW RESIDUE
          TAD    OCTCHAR                /SELECT LOWER
          AND    MASK7                  /  3 BITS FOR PRINTING.
          TAD    PRINTNUM               /MAKE OCTAL PRINTABLE
          JMS    PRINTCHAR              /PRINT THE CHARACTER
          ISZ    COUNTER                /INCREMENT AND TEST LOOP COUNT
          JMP    OCTDIG                 /DO NEXT OCTAL DIGIT
          JMP I  OCTPRINT               /RETURN IF 4 DIGITS PRINTED
MINUS4,   -4                            /NEGATIVE LOOP COUNT
PRINTNUM,260                            /BIT PATTERN TO MAKE OCTAL PRINTABLE
MASK7,    7                             /MASK FOR BOTTOM 3 BITS
PRINTCHAR,0                             /ROUTINE TO PRINT SINGLE CHARACTER
          TLS                           /OUTPUT THE CHARACTER
          TSF                           /WAIT FOR FLAG
          JMP    .-1                    /  TO BE SET, INDICATING PRINT COMPLETE
          TCF                           /CLEAR TELEPRINTER FLAG
          CLA
          JMP I  PRINTCHAR              /RETURN
NEWLINE,0                               /ROUTINE TO PERFORM CR-LF
          TAD    CR                     /LOAD CR
          JMS    PRINTCHAR              /PRINT IT
          TAD    LF                     /LOAD LF
          JMS    PRINTCHAR              /PRINT IT
          JMP I  NEWLINE                /RETURN
CR,       215                           /ISO CARRIAGE RETURN
LF,       212                           /ISO LINE FEED
/
```

[cont'd]

```
/
/ TIMER TEST
START3,   CLA                     /ISSUE GENERAL RESET
          6304                    /  TO CONTROLLER.
          DCA    COUNT1           /INITIALIZE TWO
          DCA    COUNT2           /  LOOP COUNTERS.
          TAD    STIMER           /ISSUE START TIMER COMMAND
          6304                    /  TO CONTROLLER.
          CLA
TESTIMER,6301                     /TEST STATUS REPROT FLAG
          JMP    LCOUNT           /INCREMENT LOOP COUNT IF NOT SET
          6312                    /IF SET, READ STATUS REPORT
          TAD    TIMEOUT          /  AND CHECK FOR TIMEOUT
          SZA                     /SKIP IF TIMEOUT
          HLT                     /HALT IF ANY OTHER
          6304                    /ISSUE GENERAL RESET TO CONTROLLER
          JMS    NEWLINE          /PRINT
          TAD    COUNT1           /  THE TWO
          JMS    OCTPRINT         /     COUNTS IN OCTAL
          TAD    COUNT2           /        ON A
          JMS    OCTPRINT         /          NEW LINE.
          JMP    START3           /REPEAT THE TEST
LCOUNT,   ISZ    COUNT2           /INCREMENT LOW ORDER COUNTER
          JMP    TESTIMER         /  AND JUMP BACK.
          ISZ    COUNT1           /IF LOW ORDER OVERFLOWS, INCREMENT
          JMP    TESTIMER         /  HIGH ORDER COUNTER.
          HLT                     /IF HIGH ORDER OVERFLOWS, TIMER US.
COUNT1,   0
COUNT2.   0
/ END OF TESTS.
/
```

[cont'd]

```
/ CODE FOR SIMPLE HARDWARE TESTS ON PDP-8
/    WITH DATA DYNAMICS CONTROLLER.
/
/ TEST OUTPUT OF SINGLE CHARACTER.
START1,    6421                    /RESET RECEIVE CHANNEL
           6442                    /RESET TRANSMIT CHANNEL
           6454                    /DISABLE TIMER FLAG
           TAD    SYNCODE          /LOAD RECEIVE SYNC REGISTER
           6404                    /  TO COMPLETE HARDWARE INITIALIZATION.
           CLA                     /
GO,        TAD    SYNCOUNT         /SET NEGATIVE COUNT OF LEADING SYNS
           DCA    COUNTER          /  INTO COUNTER VARIABLE.
           TAD    SYNCODE          /LOAD AC WITH SYN CHARACTER
           6441                    /START TRANSMIT CHANNEL
           6431                    /WAIT FOR OUTPUT FLAG
           JMP    .-1              /  TO BE SET.
           6432                    /OUTPUT SYN FROM AC TO TX BUFFER
           ISZ    COUNTER          /INCREMENT AND TEST SYN COUNT
           JMP    .-4              /  AND LOOP UNTIL COMPLETE.
           CLA                     /
           LAS                     /FETCH CHAR FROM HANDSWITCHES
           6431                    /WAIT FOR OUTPUT FLAG
           JMP    .-1              /  TO BE SET.
           6432                    /THEN OUTPUT CHARACTER
           CLA                     /
           TAD    PADCOUNT         /SET UP COUNTER VARIABLE
           DCA    COUNTER          /  TO COUNT NUMBER OF PADS.
           TAD    PAD              /LOAD PAD CHARACTER
           6431                    /WAIT FOR OUTPUT FLAG
           JMP    .-1              /  TO BE SET.
           6432                    /THEN OUTPUT A PAD
           ISZ    COUNTER          /INCREMENT AND TEST PAD COUNT
           JMP    .-4              /  AND LOOP UNTIL COMPLETE.
           CLA                     /
           6442                    /RESET TRANSMIT CHANNEL
           TAD    TIMECOUNT        /LOAD COUNTER VARIABLE WITH
           CIA                     /  COUNT OF NUMBER OF 100MS
           DCA    COUNTER          /     INTERVALS IN 1 SECOND.
           6452                    /ENABLE 100MS TIMER FLAG
           6451                    /WAIT FOR
           JMP    .-1              /  TIMER FLAG TO BE SET.
           ISZ    COUNTER          /INCREMENT AND TEST COUNT AND
           JMP    .-3              /  LOOP TO WAIT FOR 1 SECOND..
           JMP    GO               /DO TEST AGAIN
SYNCODE,  62                       /IBM SYN
SYNCOUNT,-6                         /NUMBER OF LEADING SYNS
COUNTER,  0                        /
PADCOUNT,-2                         /NUMBER OF PADS AFTER MESSAGE
PAD,      377                      /PAD CHARACTER
TIMECOUNT,12                       /NUMBER OF 100MS IN 1 SECOND
/
```

[cont'd]

```
/
/ TEST INPUT OF SINGLE CHARACTER.
START2, 6421                    /PERFORM
        6442                    /  HARDWARE
        6454                    /     INITIALIZATION
        TAD   SYNCODE           /        AS FOR
        6404                    /           OUTPUT TEST.
TESTIN, CLA                     /
        6414                    /ENABLE RECEIVE CHANNEL
RXIN,   6401                    /WAIT FOR
        JMP   .-1               /  INPUT FLAG TO SET.
        6402                    /READ CHARACTER FROM RX BUFFER
        DCA   CHAR              /SAVE THE CHARACTER
        TAD   CHAR              /CHECK FOR
        CIA                     /  LEADING
        TAD   SYNCODE           /     SYN CHARACTER.
        SNA CLA                 /IF FOUND, GO BACK
        JMP   RXIN              /  AND WAIT FOR NEXT CHARACTER.
        6421                    /OTHERWISE RESET RECEIVE CHANNEL
        JMS   NEWLINE           /  AND PRINT
        TAD   CHAR              /     CHARACTER IN OCTAL
        JMS   OCTPRINT          /        ON MEW LINE.
        JMP   TESTIN            /THEN REPEAT TEST
CHAR,   0                       /
OCTPRINT,0                      /OCTAL PRINT SUBROUTINE AS BEFORE
NEWLINE, 0                      /NEWLINE SUBROUTINE AS BEFORE
/
/
/ TIMER TEST.
/
START3, 6421                    /RESET RX CHANNEL
        6442                    /RESET TX CHANNEL
        6454                    /DISABLE TIMER FLAG
TIMETEST,DCA  COUNT1            /INITIALIZE TWO
        DCA   COUNT2            /  LOOP COUNTERS.
        6452                    /ENABLE TIMER FLAG
        6451                    /WAIT FOR
        JMP   .-1               /  TIMER FLAG TO SET.
/ IGNORE FIRST FLAG AS TIMER IS FREE RUNNING.
TIMEWAIT,6451                   /WAIT FOR SECOND FLAG AND
        JMP   LCOUNT            /  COUNT THE INTERVAL BETWEEN TWO FLAGS.
        6454                    /DISABLE TIMER FLAG
        JMS   NEWLINE           /PRINT THE TWO COUNTS
        TAD   COUNT1            /  IN OCTAL
        JMS   OCTPRINT          /     ON A
        TAD   COUNT2            /        NEW
        JMS   OCTPRINT          /           LINE.
        JMP   TIMETEST          /REPEAT THE TEST
LCOUNT, ISZ   COUNT2            /INCREMENT LOW ORDER COUNTER
        JMP   TIMEWAIT          /  AND LOOP.
        ISZ   COUNT1            /IF LOW ORDER OVERFLOWS, THEN
        JMP   TIMEWAIT          /  INCREMENT HIGH ORDER COUNTER
        HLT                     /IF HIGH ORDER OVERFLOWS, TIMER ERROR.
COUNT1, 0                       /
COUNT2, 0                       /
/ END OF TESTS.
```

Also, the compiler should not include the necessary run-time support software as part of the compiled program. This would include such functions as space allocation for stacks and arrays, register initialization before program start. Such functions will be provided by the hand-coded environment, which needs to make special arrangements for space allocation and stack use since some IMP code will not be run as a normal sequential program but will be activated by interrupts.

If a suitable IMP compiler is available, then the original IMP code for the communications package can be taken over directly onto the new system. This IMP code should, of course, be a correct implementation since it has already been used but it is probably still advisable to carry out some tests according to the procedures suggested below. This should bring to light any incompatibilities between the IMP implementations, such as word-length dependencies, or any possible compiler faults if the compiler has been newly developed, which is frequently the case.

If there is no IMP compiler available, then an alternative high-level language might be considered. If an alternative high-level language is to be used, then the same considerations apply about the suitability of the compiler as for the IMP. The compiler must also produce reasonably efficient code, since a significant amount of the IMP is executed at interrupt level. This normally means that extensive run-time diagnostic facilities intended as programmer aids should be removable by specifying appropriate compiler options. Two languages presently available for a number of small computers which might well be suitable are CORAL (24) and BCPL (25).

These languages were produced for this type of application and a cursory examination of two implementations indicates that they possess the required characteristics. (A more detailed discussion of the use of high-level languages for this type of programming is given in a later chapter).

The translation of the original IMP code into a different high-level language is a fairly simple, mechanical operation, especially if the new language is also block-structured. In the case of CORAL, the languages were sufficiently similar that a simple program was written which translated about 90% of the IMP automatically and flagged the remainder which it could not translate. Even a hand translation to FORTRAN was completed and tested in about one month, although this version was never used because the compiler proved to be unsuitable.

Any conversion involving hand-translation must obviously be subjected to the tests prescribed below before trying to use the new package in a real-time environment.

A third alternative is to produce a hand translation of the IMP into the assembler code of the computer. This is also a fairly simple operation, providing the ultimate in efficiency in the final code is not required. Once it has been decided how to translate each type of IMP statement in terms of register usage, etc. the process becomes quite mechanical. If code efficiency proves to be a problem, critical sections can be improved later.

An assembler version should obviously be tested thoroughly before trying to use it.

A fourth alternative, where high efficiency is a requirement, is to go back to the state diagram stage and completely re-code the package in assembler language using any special features of the particular machine instruction set to increase the efficiency. This alternative obviously involves the most work and a lot of new thought would have to be applied. This alternative has not yet proved necessary.

Providing the state diagram is strictly adhered to, the standard tests could still be applied to such a version of the package.

Testing of Communications Package

The communications package is potentially the most difficult software component on which to carry out comprehensive testing and development, It accepts standardized requests across the user interface, which it then processes in an asynchronous manner under interrupt control.

It is a fairly simple matter to check out the actions performed directly as a result of the user request but these are actually very few. By far the largest amount of code is executed as a result of interrupts. All the code to check message formats, analyze control characters, perform error recovery, check acknowledgements, etc., is performed as a result of interrupts in order to achieve a simple autonomous block transfer effect at the user interface.

Because of the very simple executive structure assumed, there were only two program levels-interrupt level and user level. There was no intermediate'supervisor' level at which code could be executed which was interruptible but not at the user level. Any code which was not executed directly as a result of the user program call had to be executed at the interrupt level.

Consequently, most of the logic to be tested in the communications package is being executed in real-time and the time between successive executions is very short, e.g. about 3.3ms at 2400 baud or 300 characters per second. It is therefore quite impossible to follow any changes in the variables used or trace the execution path followed in a particular instance without including special monitoring code which dumps relevant information into a reserved store area using a 'circular buffer' technique. This information can then be accumulated 'on-the-fly' for subsequent examination by the programmer at the end of the message transfer.

Although this method has its uses in particular circumstances it is considered to be rather cumbersome, especially in the early stages of testing since it assumes that the program is working reasonably well in order that the monitoring can be successfully carried out. Also, it is not generally possible to monitor all variables, if there are a large number, or monitor all the relevant execution paths. Some smaller choice then has to be made and it is frequently difficult to know which to choose unless a specific fault or path is being investigated, which again assumes that the rest of the package is working reasonably well. This dynamic monitoring method is useful when the package is generally working quite well but exhibiting certain infrequent faults. A monitoring of the real events sequence can then be very useful, particularly when investigating time-dependent faults.

The method of testing being proposed here is applicable in the earlier stages of testing as a means of thoroughly checking out program logic before it is applied to the real-time environment. The method depends on the fact that it is much easier to check out the behaviour of a sequential process rather than one which involves asynchronous events or any form of multi-threading, where the timing of events is not under the control of the programmer. A 'sequential program' is defined as one in which the thread of execution passes in a deterministic way from one instruction to the next, irrespective of any time delay between the instructions. The passage of the program from one instruction to the next is determined by the values of the state variables at each instruction and changes in the state variables can only be made by instructions executed in the sequence and not by any external events. The idea is to write the communications package in such a way that it can be tested, including the interrupt-driven parts, as a sequential program.

This can be done if the actions of the communications package can be described by a finite-state machine. A finite-state machine sits in a passive state until it receives a stimulus. It then performs some activity in order to generate a response to the stimulus. The response may involve some external effect and/or change in state. At the end of the activity, the machine may have changed its state or remained in the same state, but it always returns to some stable, defined state ready to receive a further stimulus. The machine is not performing any activity unless it is in the process of generating a response to a stimulus. The stimulus - activity - response sequence is the only action the machine can perform, and between such actions the machine is always in a defined state.

The progression of such a machine in response to a series of stimuli is therefore a strictly sequential process, since it will not accept a stimulus unless it has completed processing a previous stimulus and returned to a stable state. Transitions between different stable states can only take place as the result of processing a stimulus. The behaviour of a finite-state machine can therefore be described by a sequential program with a set of state variables to define the states. This accepts stimuli on its input interfaces, generates appropriate response on its output interfaces and effects state changes, and performs no action in between while waiting for a stimulus.

It will now be demonstrated that the communications package acts like a finite-state machine and therefore an implementation of it can be tested as a sequential program.

The communications package is a passive component in the overall communications system. It takes no action unless requested to do so by the user program. It can be regarded as a black box which accepts certain inputs and generates certain outputs in response to the inputs. It generates no outputs except in response to an input. The inputs, or stimuli, comprise the requests for action on the user interface (INIT, READBLOCK, WRITEBLOCK) and the interrupt requests occurring on the interrupt interface (RECEIVE, TRANSMIT, ANALYZESTATUS). The outputs, or responses, comprise the calls issued to the four hardware control routines (READDATA, WRITEDATA, READSTATUS, WRITECONTROL) and signals back to the user interface via the WAIT function. The sequence of responses is determined entirely by the sequence of stimuli. When the package is not processing a user request or an interrupt request, it is in a stable, defined state.

Gross state diagrams describing the general behaviour of the communications package are given in Figures 11.2 and 11.3. These gross diagrams treat certain sequences of stimuli as a single stimulus for the purposes of simplicity. For example, the series of RECEIVE interrupts involved in receiving a complete part of a message, e.g. the cyclic redundancy check - two characters, is treated as if the whole part arrived as one stimulus. The transition to the next state is then controlled by a count of interrupts received in the current state. A similar situation can occur for TRANSMIT interrupts.

The total state diagram for READ and WRITE is a representation of a finite-state machine to perform the required functions of the communications package. This state diagram implies certain precautions to be taken in the implementation of the machine by a program to ensure that the processing of one stimulus cannot be interrupted by another stimulus. For example, the processing of a request on the user interface must not be interrupted by a request on the interrupt interface. Similarly, the processing of a RECEIVE or TRANSMIT interrupt request must not be interrupted by an ERROR interrupt request. The characteristics of the particular hardware being used for the implementation will not necessarily afford this protection, in which case the software must provide it. This precaution is essential if the machine is to be implemented strictly in accordance with the state diagram.

Notes on State Diagrams

1. The READY state is the common state between the two state diagrams. All other states are specific to READ or WRITE.

2. Only those stimuli which are valid for the state are shown. All others are ignored and have no effect.

   The following abbreviations are used:-

RX	RECEIVE interrupt
TX	TRANSMIT interrupt
TO	ERROR (TIMEOUT) interrupt
UR	unrecognized input

   The timer does not run in transmit mode, so TO can only occur on input.

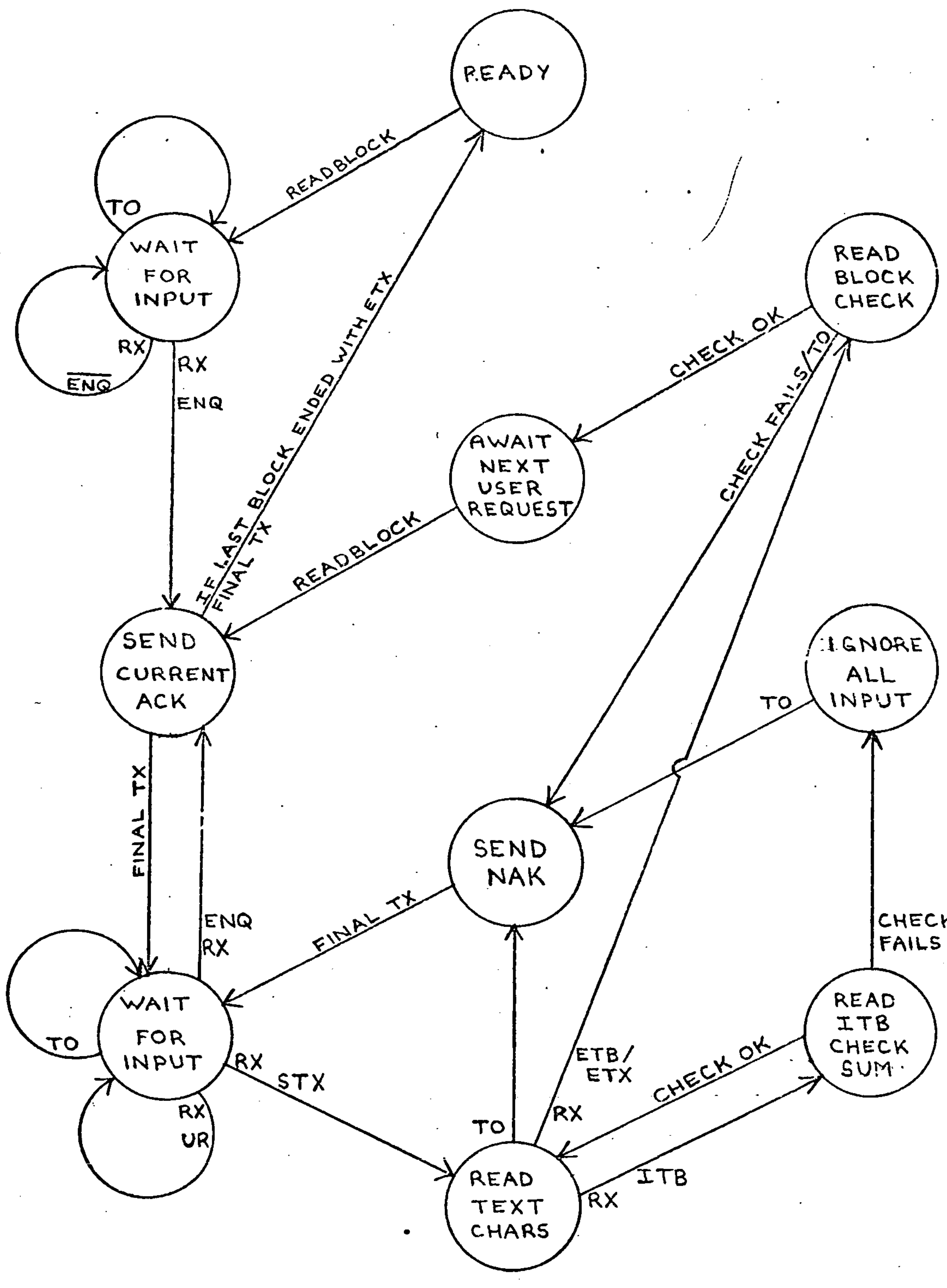
3. In a SEND state, more than one character may be output, so the transition to the next state only occurs on the FINAL TX.

4. RX always produces a READDATA response

   TX always produces a WRITEDATA response

   TO always produces a READSTATUS response

   WRITECONTROL is called to enter input mode (with ENTERRX and STARTTIMER), to enter transmit mode (ENTERTX) and to cancel both receive and transmit mode (with GENERAL RESET)



RX - RECEIVE INTERRUPT
TX - TRANSMIT INTERRUPT
TO - TIMEOUT ERROR INTERRUPT
UR - UNRECOGNIZED

FIGURE 11.2 READ MODE STATE DIAGRAM

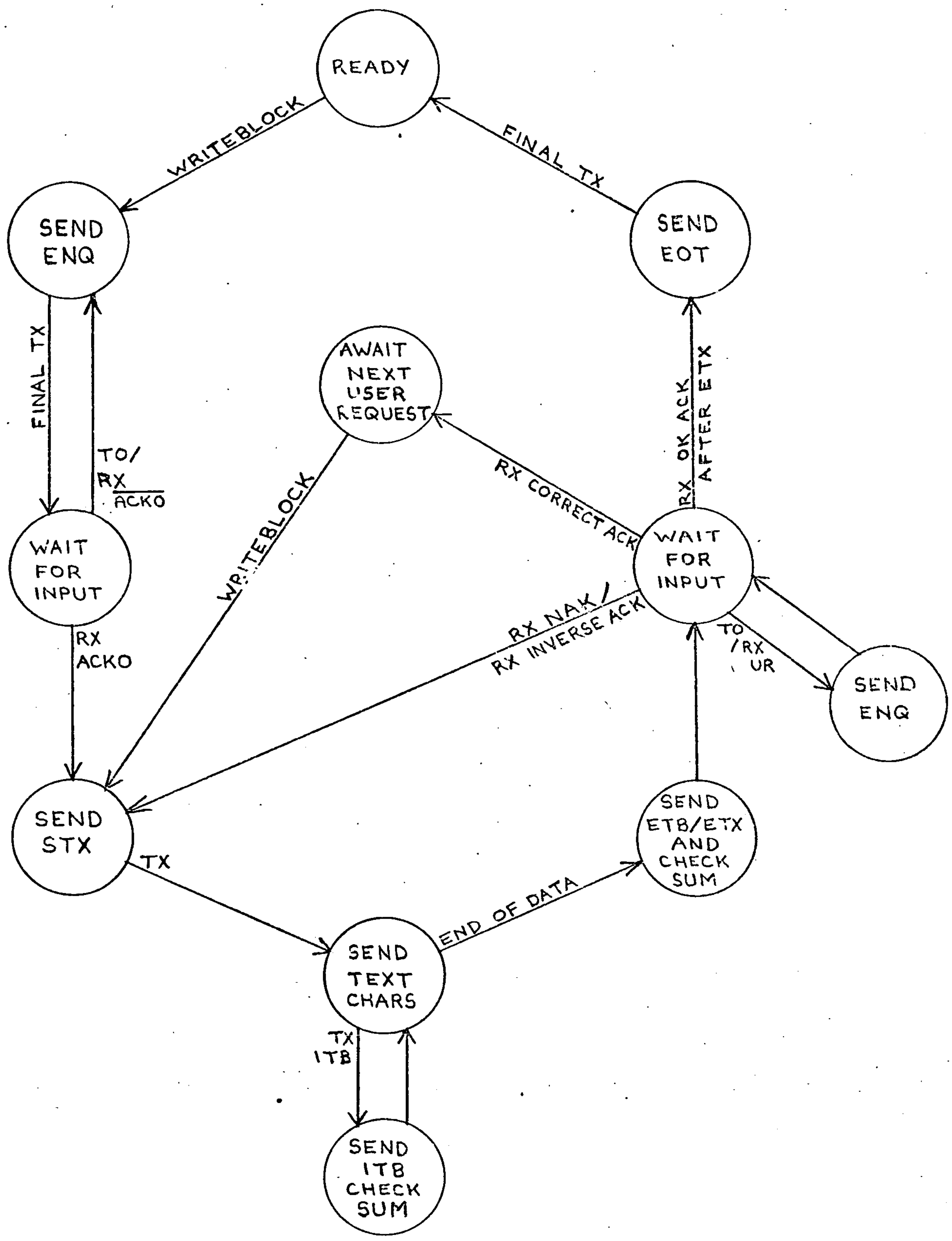


FIGURE 11.3 WRITE MODE STATE DIAGRAM

Certain other precautions are also necessary if the implementation of the machine is to behave according to the state diagram. These relate to the fact that certain stimuli are only accepted and acted upon when the machine or package is in certain defined states. For example, a user request stimulus is only accepted when a previously requested function has been completed. Similarly, a RECEIVE interrupt stimulus is not allowed when the package is in transmit mode. Such stimuli are invalid and should not be allowed to have any effect on the state of the communications package. They should be rejected or ignored at an early stage.

That the communications package as implemented is a correct representation of the state diagram can now be tested by enclosing the package in a simulated environment which takes the package through the various paths in the state diagram. The various stimuli can be simulated by routine calls on the appropriate interfaces. Where a complicated sequence of stimuli is required, e.g. to simulate the arrival of a complete message, the sequence can be controlled by a steering file, which can be prepared in advance to provide specimens of the various message formats to be handled by the package.

In this simulated environment, the progress of the package from step to step can be controlled directly by the programmer, and the state of the package can be investigated after each step if necessary. The routine calls simulating the interrupts obviously do not have to be made at the same speed as the real interrupts. They can be made one-at-a-time under programmer control so that the effect of any one interrupt stimulus can be investigated at leisure. A check can be made that the correct state transitions are being made and that the correct responses to the stimuli are being generated.

This method of testing is intended to check out the logical correctness of the communications package. At the end of this testing, it can be confidently asserted that the package is logically correct, is handling all the stimuli correctly and generating the correct responses. The package can be subjected to all the distinct message sequences in simulated form to test all the different paths. Also various error conditions can be simulated by providing stimuli via the ERROR interrupt mechanism. It is obviously far easier to detect logic errors when running in a simulated environment than in a real environment where successive events occur too quickly for any investigation of state variables to be carried out.

Since this method of testing deliberately ignores the real-time aspect, it is clearly not going to show up any errors that are strictly timing-dependent. For instance, the existence of any time-critical sections of code will not be shown up and nor will any timing inter-dependency with the rest of the system, e.g. in relation to the operation and interrupt characteristics of the local peripherals. Any such questions will require detailed investigation of the particular system in use, particularly in relation to the executive. The actual time taken for particular code paths will have to be calculated by detailed instruction counts. It is advisable to carry out such measurements before trying the system out, since otherwise random faults may occur which cannot be traced to any logical errors. This is obviously one of the difficulties of working with real-time systems. At least by using the testing method proposed, it should be possible to remove all logical errors before trying the system in a real-time environment.

A detailed example of the use of this method to test the communications package will now be given.

Sequential tests for communications package

A simple test driver can be written which provides the complete environment expected by the communications package. The driver will generate stimuli by performing routine calls on the relevant interface routines. The correct sequence of stimuli is provided by a steering file. The driver reads coded directives from this steering file for each stimulus. As a result of a stimulus the communications package will generate calls on the four hardware control routines. The driver includes versions of these routines to simulate the real ones. These dummy routines can also be used for monitoring purposes. Calls on the two hardware control routines that provide input data (READDATA and READSTATUS) will obtain this from the steering file. Calls on the two routines that produce output (WRITEDATA and WRITECONTROL) will send this output to a file for later inspection.

The controlling information obtained from the steering file can be checked against the current state of the package to check that they are keeping in step. If there is any discrepancy, then an error has been found in the communications package, assuming that the steering file has been correctly prepared.

The driver program should ideally have access to a general-purpose debugging and monitoring package, especially if the tests can be carried out on an interactive system. This package is just referred to as 'MONITOR' in the examples that follow. The particular implementation will be very system-dependent.

A simple version of the driver can be produced to test one function of the package at-a-time, e.g. READ mode. The calls on READBLOCK will then be explicitly included in the driver and only the interrupt stimuli will be controlled by the steering file. This makes the driver program a little simpler, but then a slightly different driver has to be written to test WRITE mode.

The driver shown in the examples here is a completely general purpose one which will test all functions of the communications package by taking READBLOCK and WRITEBLOCK directives from the steering file as well as the interrupt directives. The driver includes an explicit call on the INIT routine, since it is assumed that this only needs to be done once.

These examples will be coded in IMP since the communications package was originally coded in IMP. Where the package is implemented in a different language on a particular computer, the test driver would also be coded in that language. In the driver, it is assumed that all the definitions and routines of the communications package are included as part of the total program. The driver coding is given on the following pages.

```
%BEGIN
! <ALL CODE AND DEFINITIONS FOR COMMS PACKAGE INCLUDED HERE.>
! GENERAL PURPOSE TEST DRIVER FOR COMMUNICATIONS PACKAGE.
! DIRECTIVES ARE READ FROM A STEERING FILE TO CONTROL THE SEQUENCE
!   OF STIMULI APPLIED TO THE PACKAGE.
! THE CODED DIRECTIVES ARE DEFINED AS FOLLOWS:-
! 1 - READBLOCK, PARAMETERS PROVIDED BY DRIVER
! 2 - WRITEBLOCK, DATA AND PARAMETERS FOR WRITEBLOCK FOLLOW
!                    ON THE STEERING FILE.
! 3 - WAIT, WHEN THIS DIRECTIVE APPEARS, THE COMMUNICATION PACKAGE
!              SHOULD HAVE COMPLETED THE LAST REQUESTED FUNCTION.
! 4 - RECEIVE, INPUT CHARACTER FOLLOWS ON STEERING FILE.
! 5 - TRANSMIT
! 6 - ERROR, STATUS REPORT FOLLOWS ON STEERING FILE.
! 7 - STOP, USED TO STOP TEST.
!
%SWITCH SW(1:7)
%INTEGERFNSPEC READSF          ;!FUNCTION TO READ NEXT NUMBER FROM
                               ;!  STEERING FILE.
%ROUTINESPEC WRITEOUT(%INTEGER N);!ROUTINE TO OUTPUT TO MONITOR FILE
%INTEGERARRAY BUFFER(1:400) ;!COMMUNICATIONS BUFFER
INIT                           ;!INITIALIZE COMMUNICATIONS PACKAGE
GO: ->SW(READSF)               ;!SWITCH ON DIRECTIVE FROM STEERING FILE
!
! READBLOCK DIRECTIVE.
SW(1):
%IF STATE=2 %THEN MONITOR      ;!CHECK PACKAGE NOT IN WRITE MODE
BUFFADDR=ADDR(BUFFER(1))       ;!SET UP PARAMETERS
BUFFSIZE=400                   ;!  FOR CALL.
READBLOCK                      ;!MAKE CALL
-> GO                          ;!GO BACK TO STEERING FILE
!
! WRITEBLOCK DIRECTIVE.
SW(2):
%IF STATE=1 %THEN MONITOR      ;!CHECK PACKAGE NOT IN READ MODE
BUFFSIZE=READSF                ;!READ PARAMETERS FOR
BUFFTRANSP=READSF              ;!  WRITEBLOCK FROM
BUFFEOF=READSF                 ;!    STEERING FILE.
%CYCLE I=1,1,BUFFSIZE          ;!ALSO READ
BUFFER(I)=READSF               ;!  BUFFER CONTENTS FROM
%REPEAT                        ;!    STEERING FILE.
BUFFADDR=ADDR(BUFFER(1))       ;!SET UP LAST PARAMETER
WRITEBLOCK                     ;!MAKE CALL
-> GO                          ;!GO BACK TO STEERING FILE
!
! WAIT DIRECTIVE.
SW(3):
%IF WAIT #0 %THEN MONITOR      ;!CHECK PACKAGE HAS FINISHED REQUEST
! MONITOR HERE IF NECESSARY TO CHECK SUCCESSFUL EXECUTION OF REQUEST.
-> GO                          ;!GO BACK TO STEERING FILE
```

[cont'd]

```
!
! RECEIVE DIRECTIVE.
SW(4):
%IF INTADDR=2 %THEN MONITOR ;!CHECK PACKAGE NOT IN TRANSMIT MODE
RECEIVE                     ;!MAKE CALL
-> GO                       ;!GO BACK TO STEERING FILE
!
! TRANSMIT DIRECTIVE.
SW(5):
%IF INTADDR#2 %THEN MONITOR ;!CHECK PACKAGE IN TRANSMIT MODE
TRANSMIT                    ;!MAKE CALL
-> GO                       ;!GO BACK TO STEERING FILE
!
! ERROR DIRECTIVE.
SW(6):
%IF INTADDR=2 %THEN MONITOR ;!ERROR SHOULD NOT OCCUR IN TRANSMIT
ANALYZESTATUS               ;!MAK CALL
-> GO                       ;!GO BACK TO STEERING FILE
!
! STOP DIRECTIVE.
SW(7):
%STOP                       ;!FINISH TEST
! A SET OF HARDWARE CONTROL ROUTINES ARE INCLUDED TO SIMULATE
!   THE REAL ONES.
%INTEGERFN READDATA
! OPTIONAL MONITOR HERE TO FOLLOW PROGRESS.
%RESULT=READSF              ;!GET CHARACTER FROM STEERING FILE
%END
!
%INTEGERFN READSTATUS
! MONITOR HERE?
%RESULT=READSF              ;!GET STATUS REPORT FROM STEERING FILE
%END
!
%ROUTINE WRITEDATA(%INTEGER CHAR)
! MONITOR HERE?
WRITEOUT(1)                 ;!SIGNAL CALL ON WRITEDATA
WRITEOUT(CHAR)              ;!  FOLLOWED BY CHARACTER.
%END
!
%ROUTINE WRITECONTROL(%INTEGER FUNCTION)
! MONITOR HERE?
WRITEOUT(2)                 ;!SIGNAL CALL ON WRITECONTROL
WRITEOUT(FUNCTION)          ;!  FOLLOWED BY FUNCTION.
%END
!
! SUITABLE VERSIONS OF READSF AND WRITEOUT MUST BE INCLUDED, BUT THE
!   IMPLEMENTATIONS OF THESE WILL BE VERY DEPENDENT ON THE PARTICULAR
!   I/O SYSTEM AVAILABLE ON THE COMPUTER BEING USED FOR THE TESTS.
%ENDOFPROGRAM
```

Steering File

Two examples of a typical steering file to control the testing will be given here. Onee example is a test of READ mode, and the other a test of WRITE mode. For clarity, the separate items in the steering file are separated by commas, although these will not necessarily appear in an actual steering file. Comments are interspersed where necessary to explain the significance of a particular directive or group of directives. Also interspersed are portions of the output file (again with suitable explanatory comments) which would be generated by the WRITEDATA and WRITECONTROL routines, assuming the package to be working correctly. Contents of the steering file and output file are clearly distinguished. The '<data character>' following a 'RECEIVE' directive can be any valid character. For simplicity, where an arbitrary length sequence of (RECEIVE, <data character>) pairs can occur in a message, this is shown by round brackets. A similar convention is used for the output file.

For the sake of clarity, information which would be coded numerically in practice is shown symbolically here to make it easier to follow the sequences. The following symbolic abbreviations are used:-

Directives	RB - READBLOCK
	WB - WRITEBLOCK
	WT - WAIT
	RX - RECEIVE
	TX - TRANSMIT
	AS - ERROR (ANALYZESTATUS)
Status Reports	TO - TIMEOUT
	LC - LOST CARRIER
	DO - DATA OVERRUN
Output File Entries	WD - WRITEDATA
	WC - WRITECONTROL
Control Functions	GR - General Reset on communications channel
	ER - Enter Receive
	ET - Enter Transmit
	ST - Start timeout interval
	SS - Select SYN character

Steering File and Output File for READ test

OUT WC,SS	Select SYN generated by INIT routine
SF RB	First READBLOCK directive
OUT WC, ER, WC, ST	Enable receive for arrival of first ENQ
SF RX,ENQ	Arrival of first ENQ
OUT WC;GR,WC,ET	Reset receive and enter transmit for ACK
SF TX,TX,TX	TRANSMIT interrupts for acknowledgement
OUT WD,DLE,WD,ACKO,WD,PAD	Output of acknowledgement
OUT WC,GR,WC,ER;WC,ST	Cancel transmit and wait for data block
SF RX,STX,(RX,<char>),RX ETB,RX,bcc,RX,bcc	Single Record block
OUT WC,GR	Cancel receive after block
SF WT,RB	Call WAIT followed by next READBLOCK
OUT WC,ET	Enter transmit for acknowledgement
SF TX,TX,TX	Interrupts for acknowledgement
OUT WD,DLE,WD,ACK1,WD,PAD	Output of acknowledgement
OUT WC,GR,WC,ER,WC,ST	Cancel transmit and wait for block
SF RX,STX,(RX,<char>),RX, ITB,RX,bcc,RX,bcc, (RX,<char>),RX,ETB,RX, bcc,RX,bcc	Block with 2 records and ITB
OUT WC,GR	Cancel receive after block
SF WT,RB	Call WAIT followed by READBLOCK
OUT WC,ET	Enter transmit for ACK
SF TX,TX,TX	Interrupts for ACK
OUT WD,DLE,WD,ACKO,WD,PAD	Output of ACK
OUT WC,GR,WC,ER,WC,ST	Cancel transmit; enter receive
SF RX,STX,(RX,<char>),RX, ETB,RX,bad bcc,RX,bad bcc	1 record block with block check failure
OUT WC,GR,WC,ET	Enter transmit to send NAK
SF TX,TX	Interrupts for NAK
OUT WD,NAK,WD,PAD	Output of NAK
OUT WC,GR,WC,ER,WC,ST	Cancel transmit and await input
SF RX,STX,(RX,<char>),RX ETB,RX,bcc,RX,bcc	Re-transmission of block in error
OUT WC,GR	Cancel receive after OK block
SF WT,RB	Call WAIT followed by READBLOCK

```
OUT WC,ET                          Enter transmit for ACK
SF  TX,TX,TX                       Interrupts
OUT WD,DLE,WD,ACK1,WD,PAD          Output of ACK
OUT WC,GR,WC,ER,WC,ST              Cancel transmit;  wait for input
SF  RX,STX,(RX,<char>),AS,TO       Block not properly terminated
OUT WC,GR,WC,ET                    Enter transmit to send NAK
SF  TX,TX                          Interrupts
OUT WD,NAK,WD,PAD                  Output of NAK
OUT WC,GR,WC,ER,WC,ST              Cancel transmit;  wait for input
SF  RX,STX,(RX,<char>),RX,
    ETB,RX,bcc,RX,bcc              Correct re-transmission of previous block
OUT WC,GR                          Cancel receive after block input
SF  WT,RB                          WAIT followed by READBLOCK
OUT WC,ET                          Enter transmit for ACK
SF  TX,TX,TX                       Interrupts
OUT WD,DLE,WD,ACKO,WD,PAD          Output of ACK
OUT WC,GR,WC,ER,WC,ST              Cancel transmit;  wait for input
SF  AS,TO                          Timeout waiting for input
OUT WC,GR,WC,ER,WC,ST              Wait again after checking retries
SF  RX,ENQ,RX,PAD                  Receive ENQ requesting repeat of last ACK
OUT WC,GR,WC,ET                    Enter transmit to send ACK again
SF  TX,TX,TX                       Interrupts
OUT WD.DLE,WD,ACKO,WD,PAD          Output of repeated ACK
OUT WC,GR,WC,ER,WC,ST              Cancel transmit;  wait for input
SF  RX,STX,(RX,<char>),RX
    ETX,RX,bcc,RX,bcc              Last block in file
OUT WC,GR                          Cancel receive after block input
SF  WT,RB                          WAIT followed by READBLOCK
OUT WC,ET                          Enter transmit for last ACK
SF  WD,DLE,WD,ACK1,WD,PAD          Output of last ACK
OUT WC,GR                          Cancel transmit after last ACK
SF  WT                             Final WAIT,EOF parameter should be set
SF  STOP                           End of Test
```

Steering File and Output File for WRITE test.

OUT	WC,SS	Select SYN generated by INIT routine
SF	WB,40,0,0,<40 data chars>	WRITEBLOCK directive, followed by parameters and 40 data chars.
OUT	WC,ET	Enter transmit for first ENQ
SF	TX,TX	Interrupts to output first ENQ
OUT	WD,ENQ,WD,PAD	Output of first ENQ
OUT	WC,GR,WC,ER,WC,ST	Cancel transmit; wait for input
SF	RX,DLE,RX,ACKO	Receive positive ACK in response to ENQ
OUT	WC,GR,WC,ET	Enter transmit to send first block
SF	TX,40(TX),TX,TX,TX	Transmit interrupts for block
OUT	WD,STX,40(WD,<char>), WD,ETB,WD,bcc,WD,bcc	Output of block
OUT	WC,GR,WC,ER,WC,ST	Cancel transmit; await input
SF	RX,DLE,RX,ACK1	Receive correct acknowledgement
OUT	WC,GR	Cancel receive after ACK
SF	WT	Call WAIT to check termination
SF	WB,31,0,0,<16 chars, ITB, 14 chars>	Next WRITEBLOCK with data of 2 records and ITB
OUT	WC,ET	
SF	TX,16(TX),TX,TX,TX, 14(TX),TX,TX,TX	Interrupts
OUT	WD,STX,16(WD,<char>), WD,ITB,WD,bcc,WD,bcc, 14(WD,<char>),WD,ETB, WD,bcc,WD, bcc	Output of block
OUT	WC,GR,WC,ER,WC,ST	Cancel transmit and wait for input
SF	RX,DLE,RX,ACKO	Receive correct acknowledgement
OUT	WC,GR	Cancel receive after ACK input
SF	WT	Call WAIT
SF	WB,20,0,0,20<chars>	Next WRITEBLOCK with 20 chars
OUT	WC,ET	Enter transmit for next block
SF	TX,20(TX),TX,TX,TX	Interrupts
OUT	WD,STX,20(WD,<char>), WD,ETB,WD,bcc,WD,bcc	Output block
OUT	WC,GR,WC,ER,WC,ST	Cancel transmit and wait for input
SF	RX,NAK,RX,PAD	Receive NAK indicating transmission error
OUT	WC,GR,WC,ET	Cancel receive, enter transmit again

SF TX,20(TX),TX,TX,TX	Interrupts for repeat of same block
OUT WD,STX,20(WD,<char>), WD,ETB,WD,bcc,WD,bcc	Repeat block
OUT WC,GR,WC,ER,WC,ST	Cancel transmit; wait for response
SF RX,DLE,RX,ACK1	Receive correct acknowledgement
OUT WC,GR	Cancel receive after ACK
SF WT	Call WAIT
SF WB,25,0,0,<25 chars>	Next WRITEBLOCK of 25 chars
OUT WC,ET	Enter transmit for next block
SF TX,25(TX),TX,TX,TX	Interrupts
OUT WD,STX,25(WD,<char>), WD,ETB,WD,bcc,WD,bcc	Output block
OUT WC,GR,WC,ER,WC,ST	Cancel transmit; wait for response
SF AS,TO	No response - timeout error
OUT WC,GR,WC,ET	Enter transmit to send ENQ
SF TX,TX	Interrupts
OUT WD,ENQ,WD,PAD	Output of ENQ to request repeat of ACK
OUT WC,GR,WC,ER,WC,ST	Cancel transmit and await response
SF RX,DLE,RX,ACK1	Receive wrong acknowledgement
OUT WC,GR,WC,ET	Prepare to send same block again
SF TX,25(TX),TX,TX,TX	Interrupts
OUT WD,STX,25,(WD,<char>), WD,ETB,WD,bcc,WD,bcc	Output Block
OUT WC,GR,WC,ER,WC,ST	Cancel transmit; await input
SF RX,DLE,RX,ACK0	Receive correct acknowledgement
OUT WC,GR	Cancel receive
SF WT	Call WAIT
SF WB,30,0,1,<30 chars>	WRITEBLOCK with 30 chars and EOF
OUT WC,ET	ENTER transmit to send block
SF TX,30(TX),TX,TX,TX	Interrupts
OUT WD,STX,30(WD,<char>), WD,ETX,WD,bcc,WD,bcc	Output final block
OUT WC,GR,WC,ER,WC,ST	Cancel transmit; await response
SF RX,DLE,RX,<grot>	Receive invalid acknowledgement
OUT WC,GR,WC,ET	Enter transmit to request repeat
SF TX,TX	Interrupts
OUT WD,ENQ,WD,PAD	Output ENQ to request repeat of ACK

```
OUT WC,GR,WC,ER,WC,ST          Cancel transmit;  await response
SF  RX,DLE,RX,ACK1             Receive correct final ACK
OUT WC,GR,WC,ET                Enter transmit to send EOT
SF  TX,TX                      Interrupts
OUT WD,EOT,WD,PAD              Output EOT
OUT WC,GR                      Cancel transmit
SF  WT                         Final WAIT
SF  STOP                       End of test
```

11.7 Hardware Interfacing Routines

As was mentioned in the previous chapter, the hardware interfacing routines form the one software component that has to be planned and written anew for each new system. These routines provide the real environment into which fits the communications package. These routines must be designed to produce the standard interfaces to the real environment which the communications package expects.

The characteristics of this standard interface have been described previously, but it is useful to summarize them here.

Hardware to software:-

RECEIVE - next input character assembled; only occurs after package has selected receive mode and all leading SYN characters have been removed

TRANSMIT - next output character required; only occurs after package has selected transmit mode and all leading SYN characters have been generated

ANALYZESTATUS - (or ERROR) - an error condition has been detected by the communications hardware which must be notified to the software; TIMEOUT can only occur after a STARTTIMER command has been issued; LOST CARRIER and DATA OVERRUN can only occur when the package is actively receiving data; PARITY can only occur when the package is actively receiving data and parity-checking mode has been selected; MODEM FAULT can occur at any time after the hardware has been initialized.

These three interrupts should not interrupt each other.

Software to Hardware:-

READDATA - get latest input character (after RECEIVE)

WRITEDATA - put next output character (after TRANSMIT)

READSTATUS - get latest error report (after ERROR)

WRITECONTROL-- perform control function from list -

SELECT SYN CHARACTER,SET/RESET PARITY
CHECKING,ENTER RECEIVE MODE,ENTER
TRANSMIT MODE,START TIMER, RESET COMMUNICATIONS CHANNEL,
ENABLE/DISABLE COMMUNICATIONS INTERRUPTS

When implementing this interfacing software, it should be noted that it is not always necessary or even possible to implement the complete specification. For example, the communications hardware may not give any indication of LOST CARRIER or MODEM FAULT error conditions and there is no way in which the software can determine the modem status independently. Also, it is not necessary to implement the parity checking features if it is known that the transmission code to be used does not make use of character parity. The ENABLE/DISABLE INTERRUPTS control commands are only needed as a means of making the user interface routines READBLOCK and WRITEBLOCK non-interruptible by communications interrupts. Normally, execution of these routines would not be interrupted anyway because neither the receive or transmit channel is enabled when these routines are called (see state diagrams).

However, there are special optional protocol sequences which may be implemented to prevent timeout errors at the other end when there are long delays between successive calls on READBLOCK and WRITEBLOCK, such as might occur when very slow peripherals are being used, e.g. incremental graph plotter. The special sequences are described later in the chapter on communications protocols. The effect of them is that the communications channel may be active when READBLOCK or WRITEBLOCK is called, and communications interrupts may occur. In this case, it would be necessary to implement the ENABLE/DISABLE INTERRUPTS facility in order to prevent the interruption of the execution of these routines, which alter a number of state variables.

For normal operation, the minimum features of the hardware interfacing routines which must be provided for the communications package to work are:-

RECEIVE

TRANSMIT

ERROR with TIMEOUT

READDATA

WRITEDATA

READSTATUS

WRITECONTROL with SELECT SYN, ENTER RECEIVE, ENTER TRANSMIT,
START TIMER, RESET COMMUNICATIONS CHANNEL

No particular guidelines can be given about how these hardware interface routines should be implemented on a particular system, although the versions produced so far could usefully be studied. If the IMP version of the communications package is being used, it is essential that the first-level interrupt routines set up the correct IMP run-time environment, such as stack pointers, before calling the IMP interrupt routines. It is also necessary to preserve the state of the interrupted IMP run-time environment, where this involves unique store locations, for a safe return from the interrupt.

Testing the hardware interfacing routines

The hardware interface routines can only be properly tested using the real hardware in a real-time environment. The code involved is generally sufficiently simple that there is little point in running it in simulated mode first. If the code involved is at all complex on a particular system, as was the case with the Modular One where input synchronisation had to be performed by software, the complicated logic paths can first be checked out in an off-line mode using special tests.

After any initial off-line tests have been performed, the way to check out this component is to include a very simple set of routines in place of the communications package. This simple set of routines accepts interrupts via the interrupt interface and generates calls on the hardware control routines to test all the functions. This simple set of routines does not implement any protocol, but just provides for input or output of very simple messages. A minimal set can be used to mirror the functions of the simple hardware test programs described under communications hardware checkout procedures. These provide for input and output of single character messages only, and a timeout test.

The coding for an IMP version of these tests is given on following pages. It is a trivial matter to code equivalent tests in the particular language being used. The tests shown will test out the minimum functions of the hardware interface software as defined above. The tests can be easily extended to test any other functions. If these tests are performed correctly, then the system should be ready for trying the communications package proper.

11.8 User Interface routines and executive interface

The functions of the four user interface routines have been described previously. It may be necessary to re-code parts of these routines for a new system, depending on the particular mechanism used for calling routines and passing parameters. The executive interface is also considered here, since this is normally initialized as one of the functions of the INIT routine, and this will certainly need to be re-coded for a new system.

The four parameters interchanged between the user program and the communications package are buffer address, buffer size or character count, transparency mode flag, end-of-file flag. There is no particular difficulty associated with transferring those parameters except for the buffer address parameter. This will normally involve the manipulation of actual machine addresses which is not always possible in a high-level language. This will either have to be resolved by the use of in-line machine code, or by having the data buffers defined as arrays and explicity referenced by name by both user program and communications package.

```
%BEGIN
! TEST PROGRAM FOR HARDWARE INTERFACE ROUTINES.
%INTEGER CHAR,NUMPADS,STATUS,TEST
! <CODE FOR HARDWARE CONTROL ROUTINES HERE.>
%ROUTINE INIT
! SIMPLE INIT ROUTINE FOR HARDWARE AND EXECUTIVE INITIALIZING.
WRITECONTROL(RESETCHANNEL)
WRITECONTROL(SETIBMSYN)
! <PERFORM NECESSARY HARDWARE INITIALIZATION HERE.>
! <PERFORM NECESSARY EXECUTIVE INITIALIZATION HERE.>
%END
%ROUTINE RECEIVE
! SIMPLE RECEIVE ROUTINE TO INPUT ONE CHARACTER.
CHAR=READDATA                 ;!FETCH THE INPUT CHARACTER
WRITECONTROL(RESETCHANNEL)    ;!  AND CANCEL RECEIVE MODE.
%END
%ROUTINE TRANSMIT
! SIMPLE TRANSMIT ROUTINE TO OUTPUT ONE CHARACTER.
%IF CHAR>=0 %THENSTART        ;!TEST IF DATA CHAR OUTPUT
WRITEDATA(CHAR); CHAR=-1      ;!IF NOT , DO IT AND SET SWITCH.
10: %IF NUMPADS=0 %THEN WRITECONTROL(RESETCHANNEL) ;!CANCEL TRANSMIT
%RETURN                       ;!  UNLESS PADS TO BE OUTPUT.
%FINISH
WRITEDATA(PAD)                ;!OUTPUT 1 OR MORE
NUMPADS=NUMPADS-1             ;!  PAD CHARACTERS
-> 10                         ;!    AFTER DATA CHARACTERS.
%END
%ROUTINE ANALYZESTATUS
STATUS=READSTATUS             ;!FETCH STATUS REPORT
WRITECONTROL(RESETCHANNEL)    ;!  AND RESET CHANNEL.
%END
! MAIN CODE BEGINS HERE.
%SWITCH SW(1:3)
INIT
READ(TEST); ->SW(TEST)
SW(1):                        ;!INPUT TEST
10: CHAR=-1                   ;!SET INVALID CHARACTER
WRITECONTROL(ENTERRX)         ;!ENABLE RECEIVE CHANNEL
11: ->11 %UNLESS CHAR>=0      ;!WAIT FOR INPUT CHARACTER
WRITEOCT(CHAR)                ;!PRINT IT OUT
-> 10                         ;!REPEAT TEST
SW(2):                        ;!OUTPUT TEST
20: CHAR=READOCT              ;!FETCH A CHARACTER FOR TRANSMISSION
NUMPADS=1                     ;!SET CORRECT NUMBER OF PADS (0,1 OR2)
WRITECONTROL(ENTERRX)         ;!ENTER TRANSMIT MODE
21: -> 21 %UNLESS CHAR<0 %AND NUMPADS=0 ;!TEST OUTPUT COMPLETE
PRINTSTRING('TX')             ;!INFORM OPERATOR
NEWLINE
-> 20                         ;!REPEAT TEST
SW(3):                        ;!TIMEOUT TEST
30: STATUS=-1                 ;!SET INVALID STATUS REPORT
WRITECONTROL(STARTTIMER)      ;!START TIMER
31: ->31 %UNLESS STATUS>=0    ;!WAIT FOR STATUS REPORT
WRITEOCT(STATUS)              ;!PRINT IT FOR OPERATOR
-> 30                         ;!REPEAT TEST
! SUITABLE VERSIONS OF READOCT AND WRITEOCT MUST BE INCLUDED TO
!   ALLOW COMMUNICATION WITH THE OPERATOR.
%ENDOFPROGRAM
```

Any required hardware initialization should be included as part of the INIT routine. This involves selecting the particular SYN character to be used and setting the hardware into a state which can be controlled by the use of the four hardware control routines.

The executive interface must be correctly initialized from the INIT routine such that any communications interrupts are routed through to the appropriate first-level interrupt handler.

Testing of user interface routines and executive interface

The user interface routines form part of the communications package as tested by the steering file method. Since these tests are performed using the language to be used for the final implementation, the same user interface routines are applicable to the final working version. Problems in connection with passing machine addresses as parameters can be resolved at that early stage.

Those parts of the INIT routine which are different on a new system, i.e. hardware initialization and executive interface initialization, can be tested as part of the tests on the hardware interface routines. Those tests are carried out in a real environment and as such include a minimal INIT routine to perform initialization of hardware and executive interface.

If those tests, which use the real hardware and the real executive are performed correctly then the same INIT code will work in the complete system.

11.9 Conclusions about transferability

This chapter has attempted to give a complete and detailed description of a step-by-step procedure for transferring the communication system to a new small computer. This requires no understanding of the detailed internal workings of the system, merely an appreciation of the interfaces it presents to the real environment and sufficient knowledge of the new small computer to be able to match these interfaces correctly. The system therefore has the desired 'plug-in' capability.

Using the procedures described in this chapter, new versions of the system have been produced, as described in the previous chapter, by people with no previous experience of the system or of communications and with little previous experience of the particular small computer used. The time taken has been three to four man-months, including hand translation from IMP to assembler. For an experienced person using the IMP version directly, implementation of a new version should take no more than one month, including familiarisation with the new hardware.

Chapter 12

COMMUNICATIONS PROTOCOLS FOR INTER-COMPUTER WORKING

12.1 Introduction

This report has concerned itself so far with just one type of communications protocol, namely half-duplex point-to-point protocol for one way working. All the programming work described relates only to this type of protocol, and primarily to one particular implementation of it, namely IBM Binary Synchronous protocol (BSC), although all half-duplex point-to-point protocols are similar, as has been mentioned previously.

Further work has been carried out, which relates to the development and implementation of other protocols. The details of this work have not been described here, since it adds nothing new to the ideas that have been developed in this report except in so far as confirming the usefulness of the ideas in new applications, since the same techniques have been used. However, the experience gained from this work on other protocols could usefully be applied in comparing different types of protocol and judging their suitability in different applications. That is the purpose of this chapter.

It is not proposed to give an exhaustive account of protocols since the general subject is a large one, and the detailed study of particular aspects of protocols could form the basis of a thesis in its own right, as has already happened(26). It is intended rather to indicate the essential features of protocols and to bring out the points where comparisons between different ones should be made, illustrating this by reference to some examples of existing protocols. Only protocols used for point-to-point links are considered, since this is the normal method of inter-computer connection.

12.2 General characteristics of protocols

A communications protocol can be defined as a set of rules to be followed by two ends of a communications link to ensure reliable and error-free transmission over that link. In order that a data communication technique can be justifiably described as a protocol, it must demonstrate certain essential characteristics in order to comply with these criteria of reliable and error-free transmission.

If the data transfer is to be reliable, then no data should be sent unless the receiver indicates that it is willing to receive it. This means that the receiver must have complete control over the rate at which data is sent to it so there must be a 'stop/continue' response, or 'logical acknowledgement', which is generated by the receiver after receiving data.

To ensure error-free transmission of data, each data block must include a redundance check, which may be character parity and block parity or other forms of block check sum. The receiver should send a 'positive acknowledge' response if the block is received correctly or a 'negative acknowledge' response to request a re-transmission if the block is received incorrectly. This response is called the 'physical acknowledgement'. Thus, any data which does suffer transmission errors is recoverable.

Although redundancy check schemes cannot guarantee 100% error detection, the methods currently in use come very close to this. For example, a VRC/LRC check will detect all single error bursts up to 8 bits in length and over 99% of others; a CRC check will detect all single error bursts up to 16 bits in length and over 99.99% of others[27].

The protocol must also include other error control procedures in addition to the basic block check in order to recover from lost or completely corrupted transmissions. The use of block sequence counts, checking of all control sequences and the use of timeouts are techniques used to prevent lost or duplicated data blocks going undetected.

12.3 Examination of existing protocols

A number of such protocols have been devised and implemented, varying considerably in the way the link can be used. There is currently no widely implemented international standard protocol and computer manufacturers generally support a protocol peculiar to their own systems, although the simpler protocols have a lot in common. There is currently work in progress in ISO to define a new, sophisticated protocol[28] which should be suitable for a wide range of applications.

This protocol has not yet been agreed by the major manufacturers, and even if they do agree, it will be many years before it becomes widely available because it requires completely different communications hardware, incompatible with any presently used on major computers.

In the meantime, protocols currently in use or envisaged can be grouped under three headings:-

a) half-duplex with data transmission one way at a time

b) half-duplex with alternate two-way data transmission interleaved

c) full-duplex with two-way data transmission simultaneously

These three groups will now be considered separately, giving a description of the protocols in terms of the criteria above and commenting on the usefulness and areas of applicability, efficiency and complexity of implementation of the protocols.

12.4 Applicability of protocols

Areas of applicability relate to such things as bulk data transmission, Remote Job Entry applications, facilities for operators of bulk data transmission systems, interactive applications where a rapid interchange of data is a requirement, data collection applications where the data is not stored on a physical medium but is generated dynamically at a rate independent of the line speed.

12.5 Efficiency of protocols

Efficiency is a measure of the useful data traffic over the link and is defined as the ratio of actual user data transmitted per second to the quoted transmission rate of the link (e.g. 2400 baud), this figure being expressed as a percentage. The calculations of efficiency assume that the line speed is the limiting factor in the system. It is assumed that any peripherals involved are fast enough to keep the line fully loaded and that sufficient buffering is performed to permit maximum overlap of line activity and peripheral activity. This obviously depends on the particular application and implementation but economic line speeds are still relatively so low that maximum line efficiency is considered to be an important objective of a communication system. Obviously, there may be applications where high line efficiency is not important and a high speed link is intentionally under-utilized. In such systems these efficiency considerations are not relevant.

A major factor in efficiency considerations relates to the use of a 2-wire or 4-wire communications link. A 4-wire link provides two independent circuits for the transmit and receive paths, while a 2-wire link provides a single circuit which must be used alternately for transmit and receive. A 4-wire link can support both half-duplex and full-duplex protocols, while a 2-wire link can support only a half-duplex protocol. A 4-wire link can only be obtained by renting a private line from the G.P.O. Links that make use of the normal telephone network can only be 2-wire, unless two separate links are used in parallel, i.e. using 4 modems.

The difference between the two types of link arises from the use of half-duplex protocols. In a half-duplex protocol, transmission over the link only ever takes place in one direction at a time. Each end is either transmitting or receiving, never both at the same time. A half-duplex protocol relies on a 'hand-shaking' arrangement whereby protocol messages are exchanged one-for-one, with each end transmitting a message and then waiting to receive a response.

In order to transmit a message, data carrier must be generated and stabilised before any actual data can be sent and this process can take up to about 100ms(29). At the end of the message, carrier is removed to return the circuit to a quiescent state and this also takes a finite time to ensure that all oscillations have subsided. On a 4-wire link, data carrier can be maintained permanently in both directions even though no data is being transmitted. This means that data can be transmitted immediately without waiting to establish the carrier and similarly there is no delay at the end of a message when carrier is removed.

On a 2-wire link, since the same circuit is used for both transmitting and receiving, carrier must be established and removed for each message to free the circuit for the response in the opposite direction, which likewise involves establishing and removing carrier. Each change of direction of transmission, or line turnaround, therefore involves establishing and removing carrier, which can add up to 200 ms to the actual message transmission time.

Any protocol which involves frequent line turnaround or has a short average message length comparable with the line turnaround time will have a much lower efficiency on a 2-wire link than a 4-wire link. These considerations obviously do not apply to full-duplex protocols which can only operate over a 4-wire link.

The line efficiency can be increased by using a bigger average block size, such that the message transmission time becomes significantly larger than line turnaround time. This process cannot be continued to give indefinitely increasing line efficiency, however, since a larger block is more susceptible to transmission errors requiring a retransmission of the whole block. There is thus an optimum block size which is determined by the amount of buffer store available and the line error rate for any particular situation.

A graph of line efficiency against average block size would have the general shape shown in Figure 12.1. The position of the peaks and the maximum efficiency attainable for a particular protocol is very much dependent upon the error rate for the particular link in use.

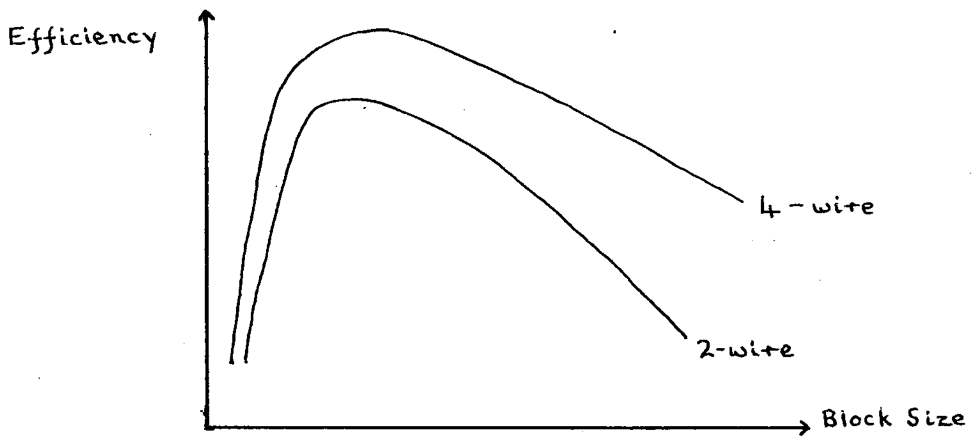


Figure 12.1 Effect of block size on data transmission

The 2-wire link is assumed to have a smaller optimum block size because the use of switched network facilities normally gives a higher error rate.

A detailed analysis of line efficiency in relation to average block size and line error rates is given in references (30) and (31).

12.6 Implementation complexity

Complexity of implementation relates to the implementation of a communications system using the particular protocol on a small computer. This includes the number of distinct protocol sequences that have to be recognised and generated, the overall complexity of the system, including the user program part, and whether this needs to support multiprogramming.

12.7 Half-duplex protocol with one-way data traffic

This is the simplest type of communications protocol and is one level of the IBM Binary Synchronous protocol(32) (BSC) and one level of the ISO Basic Mode Control Procedures(33), as used by ICL.

In the following description, the terms 'master' and 'slave' will be used to denote respectively the end transmitting data and the end receiving data. A pictorial representation is given in Figure 12.2.

This protocol starts with a link idle condition and permits either end to bid for control of the link to send data by transmitting an enquiry (ENQ) character. This can obviously produce a contention situation if both ends bid for control of the link at the same time. This can be resolved either by operator intervention or by designating one end as 'primary' and the other end as 'secondary'. The distinction lies in giving the primary a shorter timeout than the secondary for ENQ retransmission. The secondary will give up its attempt for the line if it receives an ENQ in response to its own ENQ. If the end receiving ENQ wishes to receive data,

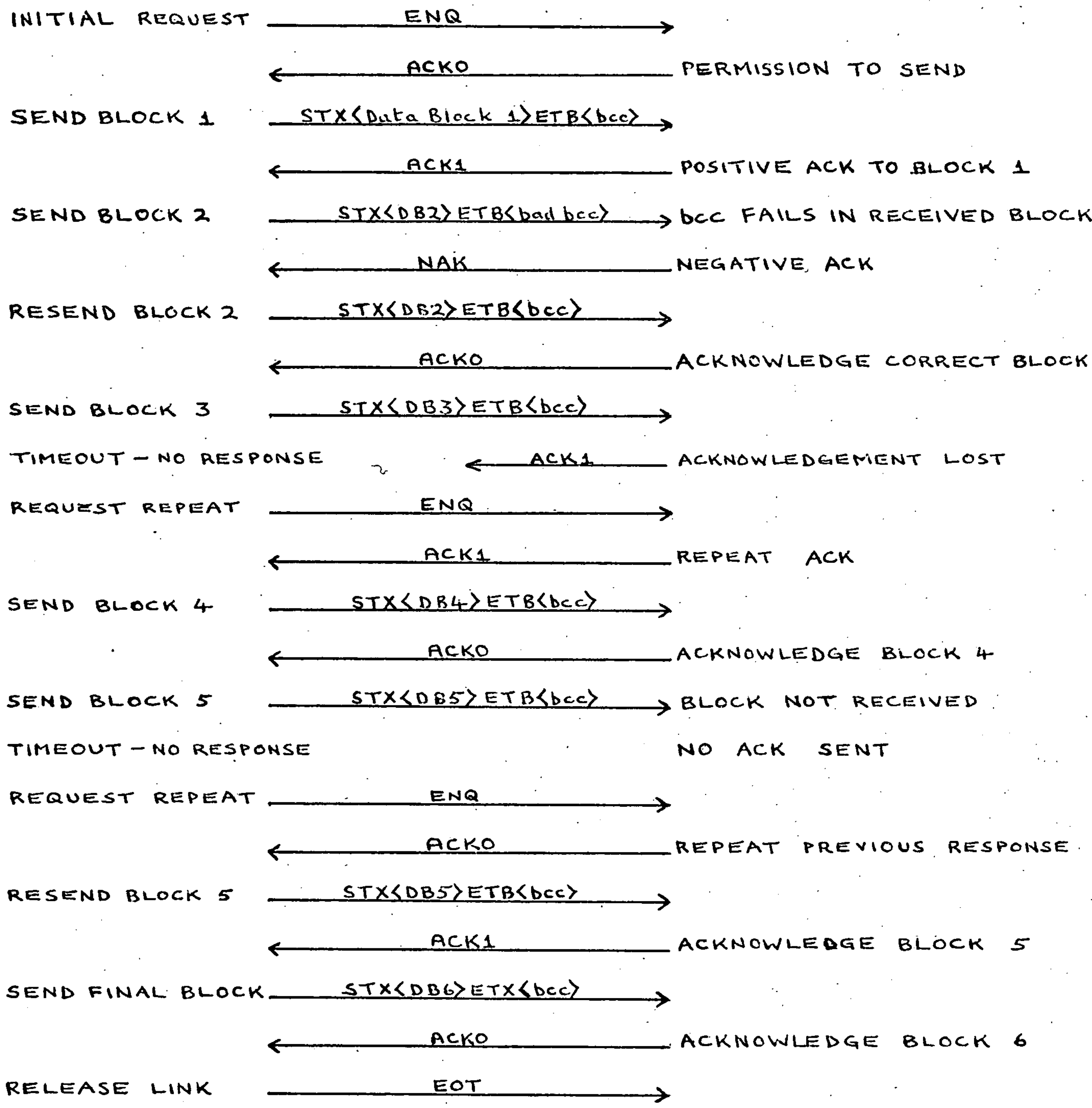


FIGURE 12.2 LINE EXCHANGES WITH HALF-DUPLEX ONE-WAY PROTOCOL

it responds with a positive acknowledgement (ACKO) and assumes slave status. Upon receiving this response, the other end assumes master status and can transmit data.

A complete file of data is then transmitted by the master as a series of message blocks, each block containing error-checking information. If a data block is received correctly, the slave responds with a positive acknowledgement when it is ready for the next block. This then constitutes both the physical and logical acknowledgement. All blocks are therefore individually acknowledged and the transmission proceeds in a 'hand-shaking' fashion. Alternating odd and even positive acknowledgements (ACKO and ACK1) are used to ensure that no block is lost as a result of an error situation. If a block is received incorrectly, the slave transmits a negative acknowledgement (NAK) to request a re-transmission. When the master has sent a block, it starts a time interval to await the response. If the time interval lapses before the response is received, the master sends an 'ENQ' character to request a repeat of the last response.

If the slave does not recognize the data being received as a data block or a control sequence, or if it receives nothing at all for a period of time, it should make no response. Using these measures, lost or completely corrupted transmissions are recoverable.

Transmission continues block by block until the master has no more data to send, which it indicates by transmitting end-of-file either using a different block-ending character (ETX instead of ETB) or an end-of-transmission control character (EOT) or both.

After an end-of-file, the link returns to an idle condition, and the appropriate procedure for starting a transmission must be used before any more data can be sent in either direction.

With these protocols, it is also possible to send addressing information with the data in order to select one of a number of alternative peripherals to receive the data. The way in which this addressing mechanism is implemented differs from system to system. The peripheral address may be included with the initial ENQ, it may be included in every data block or it may just be sent when it is required to switch from the current peripheral to a different one. This mechanism permits, for example, the operator's console typewriter to be addressed directly so that relevant information can be easily displayed for the operator.

This describes the basic rules for this type of protocol, which can be seen to have a number of limitations. Firstly, once data transmission has started in one direction, there is no way that the direction of transmission can be turned around until after end-of-file, even if the slave has an urgent message to transmit. Secondly, if the slave is outputting the data to a very slow peripheral such that it cannot accept any more data down the link because its buffers are not empty, the slave must withhold the positive acknowledgement for the previous block since sending it would imply that it was ready for another block. This will cause timeout errors at the master which will be indistinguishable from real error conditions. A similar situation exists when the master is taking the data from a very slow peripheral, which will cause timeout conditions at the slave. Neither condition causes any loss of data, but if either end interprets a long timeout as a permanent error condition, it is liable to abort the transmission and not continue.

Additional protocol sequences are defined within the BSC protocol to cater for both the above situations, although it is not mandatory that they be implemented. There are two solutions to the problem of the slave having an urgent message to transmit. The first allows the slave to transmit a single data block instead of a positive acknowledgement. Transmission then continues as before and a short reverse message has been transmitted without changing the general direction of data transmission. This is known as a 'conversational acknowledgement' and is useful for operator commands or messages against the general data flow. The other solution allows the slave to transmit a 'reverse interrupt' (RVI) acknowledgement in place of a positive acknowledgement as a signal that it wishes to reverse the direction of data transfer. The master should then empty its buffers and send an immediate end-of-file, after which the direction of transmission can be turned around. This then allows for a complete reversal of data direction on demand from the slave. This solution implies that one end has an overall higher priority for data transmission than the other end, since if both ends try to use the facility it would be difficult to achieve data transfer in either direction without continual interruptions.

The solution to the second problem above also involves the use of a special acknowledgement instead of a positive acknowledgement. This is known as 'wait before transmit' or WABT. This is sent to prevent the master from timing-out and means that the previous block has been correctly received but there is not yet a buffer available for the next one. The master responds with 'ENQ' and the slave can continue to transmit WABT until it has a buffer available when it will transmit the correct positive acknowledgement, and transmission can proceed. This mechanism allows the physical and logical acknowledgements to be separated. A similar solution can be used by a master if it has no data immediately available to send and wishes to prevent the slave from timing-out. It sends a 'temporary text delay' (TTD) sequence in place of a text block, to which the slave should respond with NAK. This exchange can continue until the master has data available again.

Sequences are also defined within the ICL use of ISO Basic Mode protocol to cater for the above limitations. The form of acknowledgement used in this protocol is <status> ACK, where the status field includes the odd/even acknowledgement switch. Other flags are included in the status field to indicate 'buffer full', meaning that the slave cannot receive any more data at the moment, and 'attention', meaning that the operator wishes to send an input message. These two responses are roughly analagous to WABT and RVI respectively in BSC. There is no direct equivalent of the TTD Sequence. If the master temporarily has no data to send, it transmits EOT without the preceding end-of-file indication normally sent when no more data is available, so that the slave can distinguish between the two conditions.

Applicability

This type of protocol is, therefore, applicable where bulk transmission of data is required between two points, e.g. Remote Job Entry, and where the limitation of predominantly uni-directional data transfer does not present any problems. This implies that the data will normally be stored on some physical medium, since there is no guarantee that data gathered or generated dynamically can be transmitted immediately unless the circuit is dedicated to transmission in one direction permanently.

The protocol is not suitable for situations requiring a rapid exchange of data, such as in interactive applications, since the procedure for reversing the direction of transmission is cumbersome and comparatively slow compared with the actual data rate.

However, most bulk data transmission applications requiring the higher speeds provided by synchronous transmissions are not seriously hampered by these limitations and this type of protocol is probably more widely used than any other for this sort of application. The main problem arises from the inability of the operator to send or receive control information while transmission is in progress and this problem can be overcome if the 'conversational' response (BSC) or the 'attention' response (ICL/ISO) is implemented.

Efficiency

The efficiency of line utilization with this type of protocol can be very high and is dependent on the maximum block size used and the delay involved in line turnaround. This latter factor is dependent upon the use of a 2-wire or 4-wire communications link, as explained previously, and the time taken to analyze the input message and generate the appropriate response, which time will be the same for 2-wire and 4-wire links.

Assuming a simple non-transparent text block with framing characters STX and ETX and two block check characters, an acknowledgement of two characters, and six leading SYN characters on each message, the redundancy due to control characters for a single data block is eighteen characters. With the IBM BSC protocol, a common maximum block size is 400 characters with an average of about 360, so that control characters use only about 5% of the line capacity. This gives a maximum line utilization of about 95% of useful data, assuming a 4-wire link with negligible turnaround delay. On a 2-wire circuit with a 200 ms turnaround delay, assuming a 2400 baud line and average block size of 360 characters, approximately 30% of line capacity is lost in turnaround time since there are two line turnarounds for each block. The effective line utilization then drops to a much lower 65% of quoted data rate, so the difference is considerable.

The standard ICL terminal using a protocol of this type employs a maximum buffer size of 80 characters and an average of only 50 characters. The average useful line utilization on a 4-wire line is then only about 70%, and on a 2-wire line drops considerably to well under 50%.

The efficiency of this type of protocol can therefore be close to 100% on a 4-wire line provided the average block size is much larger than the number of control characters required for each block transmitted. The problem of turnaround delay on a 2-wire link is common to all protocols and can only be minimised by a large average block size. The larger the block size, the greater the susceptibility to transmission errors requiring a re-transmission, so a suitable compromise must be reached (see introductory section). Since 2-wire links are normally associated with the Public Telephone Network and have a very variable quality, it is difficult to make any confident estimates of what this compromise is likely to be. The 400 character maximum block size used by ERCC in an extensive trial of Public Network operation at 2400 baud produced a recoverable error rate of about 10%(34), but no tests were done with a different block size.

Implementation complexity

A communication system using this type of protocol can be implemented with the minimum of complexity since the protocol imposes a limitation of only one active function at any one time. The implementation details of the communications protocol package have been given in a previous chapter and require some code at the user program level and some code at the interrupt level. The overall structure of the system is very simple requiring only single-thread programming at the user level and the capability to handle a single interrupt level. A minimal executive can be used with no multi-programming capability.

The user program, once started, traces a single-thread execution driving only one peripheral and making the necessary calls on the communications package. There is no necessity for any asynchronous activity in the user program if the executive handles all local peripheral interrupts to provide an autonomous transfer capability for the user program. The user program runs as a sequential process and can be easily tested in a controlled fashion, as described previously. There is no problem on controlling the use of any shared resources, such as core space or processor time, since the single activity user program has the sole use of all resources at its level, the interrupt program execution being transparent to the user level.

As an example of the minimal complexity involved in systems of this type, PDP-8 and PDP-11 implementations require only 4K core to support a configuration of, say, teletype, card reader and line printer, and this includes two 400 character buffers to double-buffer line activity.

A minor extra complication arises in the communications package if the optional extra facilities (conversational response, RVI, WABT, TTD as described above) are implemented. In the case of 'conversational response' and RVI, this merely involves changes in detailed coding, including error recovery, and an extension of the user interface. The use of WABT and TTD, however, involve the communications package in self-generated activity independent of any call on the user interface, since these features are intended for use as time-fill sequences in the absence of user requests. Since a user request may occur at any time while one of these sequences is in progress, care must be taken to ensure the proper synchronization of fresh user line activity with line activity caused by one of these sequences, as the two must not interfere with each other.

The line handler for an implementation of BSC point-to-point protocol with all the optional features has to recognize a number of distinct communication sequences off the line. These are ENQ,ACKO and ACK1,NAK,WABT,TTD,RVI,EOT, transparent and non-transparent data blocks, and the ones to be recognized depend on the current direction of data traffic. The actions to be taken on errors are also dependent on the traffic direction. This means that there is a considerable amount of decision making needed at the interrupt level, as is evidenced by the complex state diagrams shown previously which apply mostly to the interrupt level.

12.8 Half-Duplex with two-way data transmission interleaved

In applications where the limitation of one-way data traffic is unacceptable but where the restriction to a half-duplex communications facility still applies, a number of protocols have been devised to permit two-way transmission of data on an interleaved basis. The single transmission facility is used alternately in either direction, with rapid switching of direction, in some cases on a block-at-a-time basis. Depending on the actual line speed and the amount of buffering used, this can give the impression of simultaneous input and output operation.

There are two different types of protocol in this group, one typified by the upper level of the ISO Basic Mode(33) protocol as used by ICL and the other by the 'Multi-Leaving' protocol(35) as used by IBM.

With a communications link using ISO Basic Mode (see Figure 12.3) one end of the link is designated as the control station and this end controls the flow of data in both directions by the use of polling sequences. The controlled station performs no action unless in response to a signal from the controlling station, and can never take the initiative in starting a data transfer. In the general case, there can be multiple streams of data in both directions, all of which can be open simultaneously. Each stream has a unique address on the controlled station, e.g. a card reader stream might be stream 1, paper tape reader stream 2, operator console input stream 3, line printer output stream 4, etc., although streams do not have to be associated with actual physical devices. The controlled station usually has a unique address (different from stream addresses) associated with it as well to allow for shared use of the link by different stations on a switched basis.

The first action taken by the controlling station is to send a poll sequence of the form EOT - <station address> - ENQ. If the address matches its own, the controlled station should respond with a sequence of the form <address> <status> - ACK. The status field is a bit pattern which gives an indication of which data streams are available for data transfer, e.g. if the line printer were operational, this stream would be signalled as available for data transfer, but if there were no cards in the card reader, then this stream would be signalled as unavailable.

The controlling station examines the status field of the response and decides on which stream to initiate a data transfer. It then

CONTROLLER		CONTROLLED
GENERAL POLL	EOT<STATION ADD>ENQ →	
	← <STATION ADD><STATUS>ACK	RESPOND WITH OVERALL STATUS
SELECT OUTPUT STREAM	EOT<OUTPUT ADD>ENQ →	
THIS END NOW MASTER	← <OUTPUT ADD><STATUS>ACK	RESPOND WITH STREAM STATUS
BLOCK 1 OF MESSAGE	STX<DATA BLOCK 1>ETB<bcc> →	
	← <OUTPUT STATUS>ACK	RESPOND OK
FINAL BLOCK OF MESSAGE	STX<DATA BLOCK 2>ETX<bcc> →	RECEIVED IN ERROR
	← NAK	REQUEST REPEAT
SEND BLOCK 2 AGAIN	STX<BLOCK 2>ETX<bcc> →	
	← <OUTPUT STATUS>ACK	CORRECT ACK
GENERAL POLL AGAIN	EOT<STATION ADD>ENQ →	
	← <STATION ADD><STATUS>ACK	RESPOND WITH OVERALL STATU
SELECT INPUT STREAM	EOT<INPUT ADD>ENQ →	THIS END NOW MASTER
	← STX<BLOCK 1>ETX<bcc>	SEND SINGLE BLOCK MESSAGE
RESPOND OK	<CONTROL STATUS>ACK →	
	← EOT	RELEASE LINK
GENERAL POLL	EOT<STATION ADD>ENQ →	
	← <STATION ADD><STATUS>ACK	OVERALL STATUS
SELECT OUTPUT STREAM	EOT<OUTPUT ADD>ENQ →	
THIS END NOW MASTER	← <OUTPUT ADD><STATUS>ACK	RESPOND WITH STREAM STATUS
BLOCK 1 OF MESSAGE	STX<BLOCK 1>ETB<bcc> →	
	← <OUTPUT STATUS>ACK	RESPOND OK
FINAL BLOCK	STX<BLOCK 2>ETX<bcc> →	BLOCK NOT RECEIVED
TIMEOUT - NO RESPONSE		
REQUEST REPEAT	ENQ →	
	← <OUTPUT STATUS>ACK	REPEAT LAST RESPONSE
SEND BLOCK 2 AGAIN	STX<BLOCK 2>ETX<bcc> →	
	← <OUTPUT STATUS>ACK	RESPOND OK
START CYCLE AGAIN	EOT<STATION ADD>ENQ →	

NOTE THE USE OF NORMAL ONE-WAY PROTOCOL ONCE MASTER IS ESTABLISHED

FIGURE 12.3 LINE EXCHANGES FOR ISO HALF-DUPLEX TWO-WAY PROTOC

sends out a sequence EOT -<stream address> - ENQ, specifying the stream it has selected. If this is an inbound stream to the controlling station, the controlled station should respond with a block of data for that stream. If the controlling station responds with <status> - ACK, the controlled station can send another block of data for the same stream. In other words, once the stream has been selected, the protocol reverts to the simple one-way point-to-point protocol and data transfer can continue on the same stream, with the normal point-to-point error recovery procedures being used. If the controlled station has no more data to send on that stream, it transmits an end-of-transmission (EOT) character. The controlling station can then either try to select one of the other available streams or send a general poll to the station again. Alternatively, at any point during transmission on the inbound stream, the controlling station can send a new stream select sequence, instead of the acknowledgement, to select a different stream even though there is still data available for the first stream selected. The controlling station can also set a status bit in its acknowledgement to request the controlled station to stop transmission after the current buffer. Thus, the controlling station can switch streams dynamically at any point to any one of the streams marked as available after the last general poll.

If the controlling station selects an outbound stream, the controlled station should respond with <stream address> - <status> - ACK, this time giving the status of the selected stream, e.g. data transfer can continue, data buffer temporarily full, output device non-operational, etc. Assuming the status indicates that data transfer can proceed, then the controlling station sends a block of data for the stream. Data transfer then continues under control of a one-way point-to-point type protocol in a similar way to inbound streams. If at any point, the status response from the controlled station indicates that data transfer cannot proceed, the controlling station will stop sending data on that stream and will send out another stream select sequence for a different stream or a general poll to the controlled station. Alternatively the controlling station can send out a new stream select sequence at any point instead of a data block, thereby switching to a different stream.

The responsibility for initiating transfers on any stream, therefore, always rests with the controlling station and it makes decisions on the basis of information received from the controlled station about the status of the streams.

With this protocol, it is possible to support multiple two-way data traffic by switching frequently between streams. If a number of streams were active, the controlling station would have to schedule use of the link between them, possibly on a round-robin basis, in order to keep them all serviced. The controlling station must send out a general poll at frequent intervals so that the availability of a new stream can be detected as soon as possible. This protocol is inherently unsymmetrical and will not operate if two controlling stations or two controlled stations try to communicate with each other.

With the other type of protocol in this group, namely Multi-Leaving (refer to Figure 12.4), both ends of the link are equivalent once communication has been established and neither end has control over the other. Communication is established by one end or the other sending the character sequence SOH-ENQ. The end that receives this responds with an acknowledgement, ACKO, and the link is then established and any further use of the link is entirely symmetrical.

There are only three distinct communication sequences that can be transmitted by either end. These are a data block, which may contain data and/or control information, ACKO, which is transmitted when there is no data block to be sent, and NAK, which is sent to request a re-transmission. The reception of ACKO or a data-block implies a positive physical acknowledgement for the last data block transmitted. The control signals of the protocol, used to start and stop/continue streams, are sent in the form of data blocks rather than as special communication sequences. Multiple two-way streams can be used with this protocol, and all streams are assigned a unique address of 8 bits.

When the link has been established, but no data transfer has been started, the ACKO sequence is transmitted back and fore from end to end at frequent intervals.

The responsibility for the next transmission over the link lies with the end that last received something. Control over the link is therefore exchanged everytime a transmission takes place and communication proceeds on a hand-shaking basis. The one exception to this rule arises in certain line error conditions.

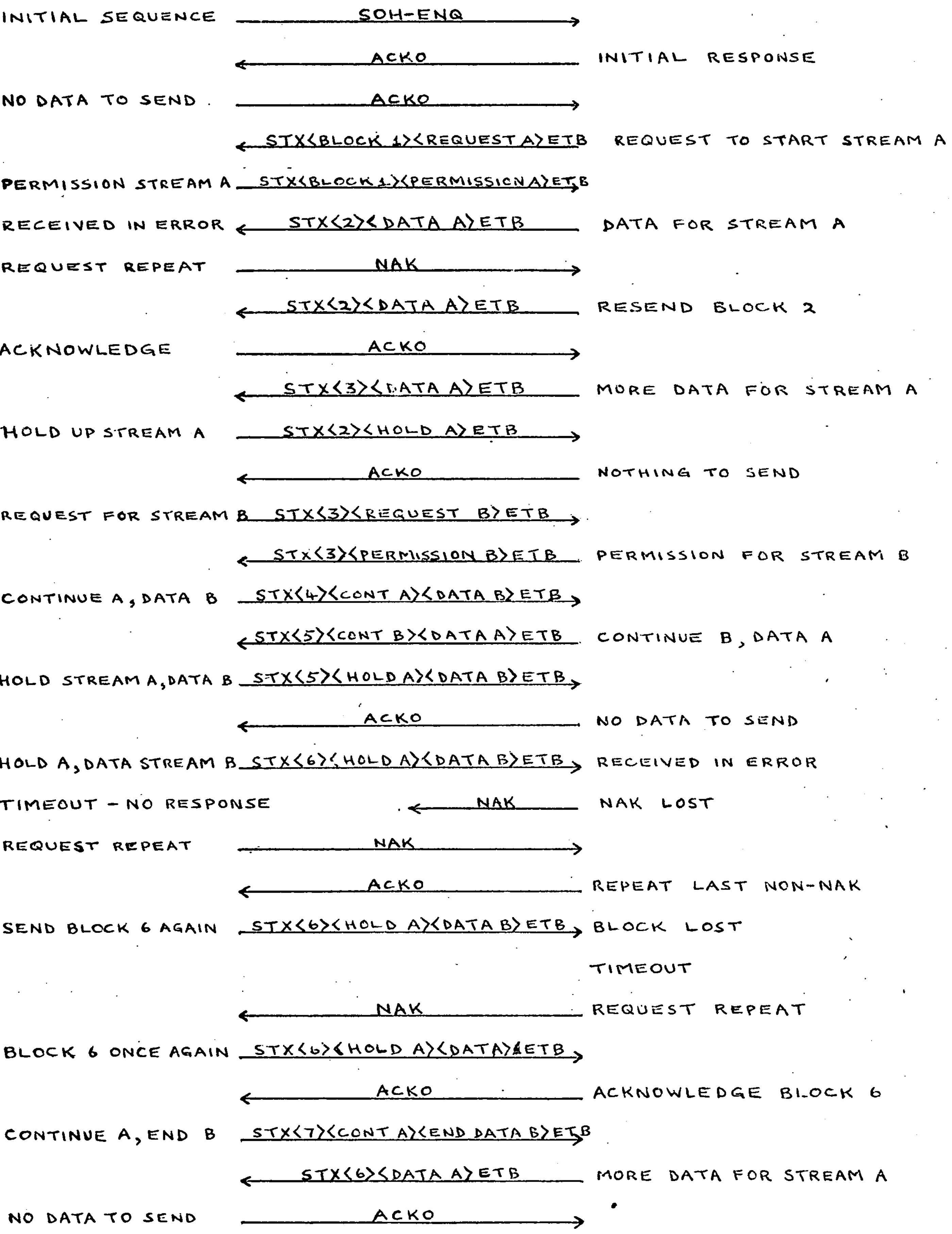


FIGURE 12.4 LINE EXCHANGES FOR HALF-DUPLEX MULTI-LEAVING

Whenever one end completes a transmit sequence, it reverts to receive mode but sets a timeout for 3 seconds. This timeout is to prevent the link hanging up with both ends in receive mode after a transmission has been lost for any reason. If the timeout expires, then that end transmits a NAK sequence to request a repeat of the last transmission. If one end is in receive mode and receives a NAK, then it repeats the last non-NAK transmission, whatever it was.

Clearly, if one end has just received something and has no data immediately waiting to be sent in response, then it must send some response, say, ACKO, within the timeout period to prevent the other end from timing out. This requirement produces the regular ACKO exchange when no data is being transmitted. This idle ACKO sequence should not be sent more frequently than, say, once per second, to prevent excessive interference with the system at either end when the line is idle, assuming that one or other end is performing other work as well.

The responsibility for initiation of data transfer on a particular stream lies with the source of data for that stream. To start an outbound stream, therefore, e.g. when a card reader has been loaded with cards, the source end constructs a data buffer with a 'start stream request' control sequence, specifying the required stream. If the source end currently has control of the link, the buffer can be sent immediately, otherwise it must wait until the other end transfers link control by transmitting something. If the receiving end for that stream wishes to accept data, it likewise constructs a data buffer with a 'start stream permission' control sequence, again specifying the stream address. When this is transmitted back to the source end, the source can then send data blocks for that stream.

Any number of streams up to the maximum configuration can be started in this way in either direction. It is always the responsibility of the source end of a data stream to start data transfer on that stream by means of the appropriate control sequence. Since this control sequence must be acknowledged by the receiver of the data stream, there is no possibility of unsolicited data being transmitted. An end-of-file control sequence from the source end indicates an end of data on that stream until a further start stream request is made.

Once data transfer on a particular stream has been started, data will continue to flow from the source unless the receiving end cannot handle, process or output the data at the rate the source is generating it. To achieve control over this, a series of bits called a stream control sequence is included at the start of every block transmitted. This sequence has one bit assigned for every inbound stream, and the bit is set on or off to indicate whether more data can be sent on that stream or not. The state of each bit in this sequence must be examined by the link handler at the receiving end to decide what outbound streams can proceed. This control bit then performs the function of the logical acknowledgement.

The amount of data buffering at the receiving end of each stream is controlled entirely by the receiving handler and does not affect the protocol. If a number of outbound streams are active and can proceed, it is the responsibility of the line handler to ensure that each is serviced regularly, which may require some scheduling and this can be organized on a priority or a round-robin basis, depending on the nature of the data streams.

In normal use, with data streams active in both directions, data traffic consists entirely of a one-for-one exchange of data blocks, with each data block implying a positive acknowledgement of the previous block in the opposite direction. Because alternating odd and even acknowledgements are not used, a modulo 16 block count field is included in each block. This should ensure the detection of duplicate blocks or lost blocks.

Applicability

The area of applicability of both these protocols is obviously much wider than that for one-way point-to-point protocol, since multiple two-way data transmission is permitted. The protocols can still be used for bulk data transmission and can provide a more powerful facility in this area, allowing for traffic in both directions. If necessary for bulk transmission, a high-speed transmission link can support shared use by a number of peripherals, as in complicated RJE applications. The problem of operator commands and messages in parallel with data traffic is also easily handled with these protocols, since the operator merely represents two additional data streams which can be accommodated without any special sequences.

These protocols could also handle interactive traffic in addition to more powerful bulk data transmission. The rapid exchange of short interactive messages is easily accommodated in both protocols, although it is rather more cumbersome with ISO protocol, requiring continuing use of stream selection seqqences.

The inherently unsymmetrical nature of the ISO protocol limits its usefulness in general applications. It is oriented primarily towards communications between terminal devices and a single main computer, and implées a 'big end' and a 'little end' for each communication link that uses it. The main computer is always designated as the controlling station so that it can achieve complete control over terminal use and general work flow. It is unfortunate that the protocol cannot be used to communicate between terminals or between the main computers if the normal protocol control programs are used. Special versions of the protocol programs could be written to make a main computer behave like a terminal or vice versa, but this seems an unsatisfactory way of handling the problem. The Multi-Leaving protocol, being symmetrical after initialization, could handle both cases above with no difficulty.

Both these protocols essentially perform concentrator functions, replacing a number of parallel, simultaneous uni-directional links with one link of normally higher speed shared between all the single streams. This should produce a considerable cost reduction in charges for modems, transmission lines and other equipment at the expense of rather more complicated line handling software or hardware.

Efficiency

The efficiency of these protocols depends on the extent to which the multiple two-way data traffic facility is actually used. Both protocols can, of course, be used to support just a single one-way data stream if required.

At the one extreme, when just a single one-way data stream is active, the ISO protocol reverts to the simple one-way point-to-point protocol described previously, with the transmission of only blocks and acknowledgements. In this case, the efficiency considerations are exactly the same as for the one-way point-to-point protocol, and the efficiency can approach 100% on a 4-wire link if the average block size is much larger than the fixed overhead for control characters.

The Multi-Leaving protocol used in this way is slightly less efficient as there is a larger overhead of control characters. Although transmission consists entirely of the exchange of data blocks and acknowledgements, each data block contains a small number of control characters. These currently consist of one count character, two stream control sequence characters and a single stream address character identifying each record in the block. For an average record size of 40 characters (measured in RJE applications) and a maximum block size of 400 characters, this will contain an average of 9 records, so that the overhead is 12 characters per block of about 370 characters. The overhead for transmission control characters is the same as before at 18 characters, so the useful character rate is about 360 out of 390, just over 90%. The efficiency is therefore still quite high, but not so high as point-to-point or ISO protocol when used in this way. The use of a 2-wire link or the existence of long turnaround delays will affect both protocols in the same way when used in this mode.

At the opposite extreme of use, these protocols will be used to transmit multiple streams in both directions at the same time and this mode of use affects the two protocols in different ways.

With the ISO protocol, it is assumed that one data block will be transmitted for each stream in turn in order to give them all regular service. This means that the controlling station must send out a new stream select sequence after each block in order to switch streams, and also possibly a general poll in order to keep up to date information about stream availability. In the worst case, the controlling station sends out a general poll, followed by a stream select sequence for an outbound stream, followed by a data block for that stream, with the controlling station making the appropriate acknowledgement at each stage. This therefore involves six line exchanges to transmit one data block, and a total of 55 control characters consisting of 6 SYNs for each message, 3 characters for a poll or stream select, 3 characters for an acknowledgement and 4 non-data characters in the block. Assuming negligible turnaround delays, this gives an efficiency of about 87% for an average block length of 360 characters. If an inbound stream is selected, the number of line exchanges per block reduces to 4 and the efficiency rises to about 90% on 360 character blocks.

The efficiency thus generally remains quite high despite the large number of line exchanges, and this is due to the large average block size. In some uses of this protocol, the block size is constrained to only one record per block. Since the average record size is around 40 characters in typical applications, this results in a very low line efficiency of less than 50%, even on a 4-wire link.

The effect of a 2-wire link on ISO protocol used in this way is considerable because of the large number of line turnarounds required for each data block. For the first two examples given above, 6 lines turnarounds of 200 ms each would take 1.2 secs in total, which is the same time as the transmission time for a 360 character block at 2400 baud, so the efficiency drops to around 50%. For 4 line turnarounds, this takes .8 secs to give an efficiency of about 60% for a 360 character block. The use of multiple streams with ISO protocol on a 2-wire link therefore causes a considerable drop in efficiency, even with large block size. If small block sizes are used, as in the third example given above, the efficiency becomes very low indeed.

The result of using Multi-Leaving protocol to transmit streams in both directions is actually to increase the line efficiency, since explicit acknowledgements are not needed if data blocks are being sent in both directions. A data block implies positive acknowledgement of the previous transmission in the opposite direction. The overhead on each data block then consists of the 10 transmission control characters (6 SYN; STX; ETB; 2 block check characters) plus 3 control characters in the block (block count; 2 stream control sequence) plus 1 stream address character for each record. Assuming a maximum block size of 400 characters and an average record size of 40 characters as before, this overhead amounts to an average of 22 characters on a block of about 360 data characters. This gives a line efficiency of about 95%, which compares with the maximum figure obtained with the one-way point-to-point protocol.

The effect of using a 2-wire link is to add an overhead of 1 line turnaround for each data block, or about 200 ms extra every 1.2 seconds at 2400 baud. This would reduce the line efficiency to about 80% when data is flowing in both directions.

The Multi-Leaving protocol is therefore seen to give a higher line efficiency than ISO protocol when transmitting data in both directions. This is particularly true when a 2-wire link is used. The effect of using a smaller block size is also less noticeable with Multi-Leaving protocol.

One can conclude then that the most efficient line utilization for applications where two-way traffic on ahhalf-duplex facility is required is obtained by the Multi-Leaving type of protocol, which tries to minimise the number of line turnarounds per data block. This is particularly significant on 2-wire links and since the use of 2-wire links is likely to increase since the introduction of the GPO Dial-Up 2400 baud service, this conclusion is very relevant.

Implementation complexity

The implementation of a system employing one of these protocols is an order more complex than one using the simple one-way point-to-point protocols. Since there can be several data streams in transit at the same time, this implies the existence of a number of parallel activities in the system, one for each stream.

For example, on an RJE system, there might be separate activities for the card reader, line printer and operator console, or on a teletype concentrator there might be one activity for each teletype. If each activity is written as a separate sequential process, the overall system must provide some way of sharing the resources such as CPU time, core store, etc., between the different activities. Thus, there must be some kind of multiprogramming scheduler to control the whole system.

Also, since the communications link is now shared between all the activities, the line handler is more complex and must service requests from a number of activities and arrange to multiplex them properly. The problem of synchronization between the activities and the line handler is more complicated. Instead of one activity having complete control of the line as in the one-way point-to-point protocol, one activity may make a request while the line is already active handling a previous request from a different activity. This means that a system of queues must be used to provide adequate buffering between the activities and the line handler, rather than the simple double-buffering mechanism which is adequate for the single-activity system. Also, the line handler itself with a half-duplex protocol must make scheduling decisions about the use of the line, whether to use it for input or output at any particular time in order to achieve a proper balance between inbound and outbound streams, depending on the traffic demand for the different streams.

The whole system, then, becomes more complicated because of the need to schedule the allocation of shared resources in order to achieve optimum efficiency of the overall system.

Compared with the 4k core store systems needed to implement one-way point-to-point protocol on PDP-8 and PDP-11, implementation of ISO and Multi-Leaving on a PDP-11 each required 8k of core store. An indication of the complexity of the line handler for Multi-Leaving in scheduling line use is shown by the decision tree used by the line handler after the termination of a line exchange in reference (35).

The line handler for ISO protocol has to recognize 5 distinct communications sequences off the line, these being EOT - <address> - ENQ; <status> ACK; NAK; ENQ; data block. It must generate 5 sequences, as the above but with EOT instead of EOT -<address> - ENQ. The sequences it must recognize and generate depend on whether an inbound or outbound stream is currently being transfered. The error recovery procedures are also different for inbound and outbound streams, and are similar to those for one-way point-to-point protocol.

There is thus a considerable amount of decision making necessary at the interrupt level in order to recognize input messages and perform error recovery. The implementation for the controlled end of the link does not make scheduling decisions for the link as these are made by the other end, and the controlled end merely has to do as requested.

The Multi-Leaving line handler has to recognize and generate only 3 distinct sequences, namely ACK0,NAK; data block, apart from the SOH-ENQ sequence used for initialization. Since the Multi-Leaving protocol is symmetrical, both ends use the same error recovery under all conditions, and both ends must implement the scheduling decisions for use of the link. The amount of decision-making at interrupt level is therefore less with Multi-Leaving than with ISO, but the scheduling decisions are complicated.

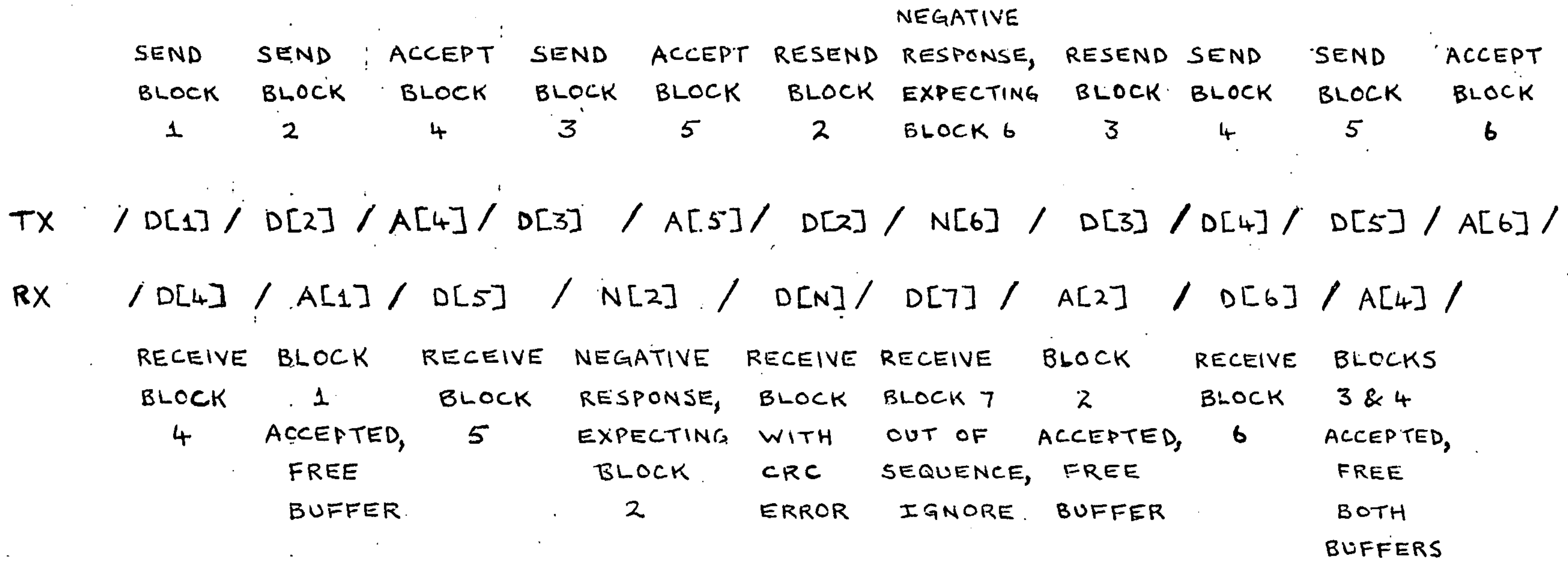
12.9 Full-duplex protocol with two-way simultaneous data traffic

The previous section has shown that the use of half-duplex links to support two-way traffic introduces some awkward problems of scheduling line activity in order to achieve a balance between inbound and outbound data streams. These problems arise from the general difficulty of trying to share a common resource between two completely different uses. The obvious solution to these problems is to provide two separate resources to satisfy the two different uses, instead of trying to share a single resource. This is the idea behind the introduction of full-duplex protocols to serve general two-way transmission systems. Each of the two ways has its own dedicated transmission circuit and the use of this is always controlled by the same end. No allowance has to be made for the other end wanting to use the circuit for responses.

The transmitting line handler merely has to schedule the use of the circuit between all the outbound streams, on a priority or round-robin basis. Also, since the nature of protocols requires that some response is made to a transmission, responses to data received on the inbound circuit must be sent on the outbound circuit, but these can be scheduled in along with the outbound streams.

ISO is currently considering a wide-ranging future standard for data communications, referred to as HDLC, or High-Level Data Link Control.(28) This standard is intended to cover both full and half-duplex transmission and many application areas, but only one level of it will be considered here as an example of full-duplex computer-computer communications, namely the full-duplex primary-primary procedures, and Figure 12.5 shows the use of this.

All exchanges in HDLC consist of data blocks - there are no other communications sequences. All necessary control information is contained as fields within the data blocks. A circulating block sequence count is included in each block transmitted along the same circuit so that the receiver can keep a check on any blocks lost or duplicated. Each block has a cyclic redundancy check for error detection. Physical acknowledgements are accomplished by transmitting the highest block number of the blocks so far correctly received forming a consecutively numbered sequence. Thus, if blocks 2,3,4,6 have been received, an acknowledgement is sent indicating the number 4. It is the responsibility of the transmitter of the blocks, which knows it has transmitted blocks 2,3,4,5,6, to deduce that blocks 2,3,4 were correctly transmitted, and free the corresponding buffers, but that an error occured in


SEND BLOCK 1
SEND BLOCK 2
ACCEPT BLOCK 4
SEND BLOCK 3
ACCEPT BLOCK 5
RESEND BLOCK 2
NEGATIVE RESPONSE, EXPECTING BLOCK 6
RESEND BLOCK 3
SEND BLOCK 4
SEND BLOCK 5
ACCEPT BLOCK 6
TX / D[1] / D[2] / A[4] / D[3] / A[5] / D[2] / N[6] / D[3] / D[4] / D[5] / A[6] /
RX / D[4] / A[1] / D[5] / N[2] / D[N] / D[7] / A[2] / D[6] / A[4] /
RECEIVE BLOCK 4
BLOCK 1 ACCEPTED, FREE BUFFER
RECEIVE BLOCK 5
NEGATIVE RESPONSE, EXPECTING BLOCK 2
RECEIVE BLOCK WITH CRC ERROR
RECEIVE BLOCK 7 OUT OF SEQUENCE, IGNORE
BLOCK 2 ACCEPTED, FREE BUFFER
RECEIVE BLOCK 6
BLOCKS 3 & 4 ACCEPTED, FREE BOTH BUFFERS

ABBREVIATIONS -	D[M]	DATA BLOCK M
	/	ONE OR MORE CONSECUTIVE BLOCK FRAMING CHARACTERS
	A[M]	POSITIVE RESPONSE TO DATA BLOCK M
	N[M]	NEGATIVE RESPONSE, BLOCK M EXPECTED NEXT

FIGURE 12.5 FULL DUPLEX PROTOCOL – TX AND RX DATA FLOW AT ONE END OF LINK

block 5 which must be re-transmitted. Block 6 would also be re-transmitted, and this would be ignored by the receiving end unless it had discarded the original block 6 because it occurred out of sequence.

If a block is received in error, or some line error condition is detected on input, then a negative acknowledgement is transmitted giving the block number for the next input block expected. The transmitter then knows from what point to retransmit. The transmitter can also send an enquiry sequence, requesting a repeat of the previous response. Responses are also defined to indicate lack of availability of buffers at the receiver.

This level of definition of the protocol only covers the physical aspects. Logical control over individual streams is effected at a higher level by further control fields within the data block.

Requests to initiate streams and logical acknowledgements are sent as control sequences in a data block and all data records are prefixed by a stream address for identification. Thus, the physical level is concerned only with the transmission and reception of blocks in correctly numbered sequences. All other aspects of the protocol are handled at a logical level based on block contents.

The transmit and receive line handlers are completely separate and operate independently. The necessary physical acknowledgements generated by the receive handler must be placed on a queue for the transmit handler, which will schedule them along with other outbound requests. Also, the physical acknowledgements received by the receive handler must be handed over to the transmit handler so that it can decide which buffers to free and which buffers to send again.

Applicability

The HDLC protocol is intended to be applicable to all sophisticated transmission systems as a general purpose data transport facility. This full-duplex primary-primary subset of it is a symmetrical protocol and should be suitable for the powerful RJE terminal and interactive applications, both at the same time if necessary, provided a sufficiently fast line is used. No limitations are apparent for large scale systems, although, of course, it has not actually been used anywhere yet.

It is less suitable for simple applications that could be adequately handled by one-way point-to-point protocols because of the extra facilities built in for sophisticated systems, such as multiple streams. It should be possible, though, to define a further subset of it for simple applications that was compatible at the hardware level with the full specification. This would allow extra facilities to be added later by software enhancements. The minimum hardware level involves only the block framing characters, which are always the same, and the assembly of whole characters. All other features could be handled by software on a simple system.

It should be possible to define a one-way subset that can also be used with half-duplex facilities. This would then be compatible with present arrangements for dial-up connections.

Efficiency

The efficiency of HDLC in full-duplex mode, and of full-duplex protocols in general, can be very high as they can make full use of a 4-wire link. There are no line turnaround delays, and, provided the transmitter has sufficient buffering to transmit ahead of acknowledgements, it can attain practically continuous use of the transmit circuit with only negligible delays between the end of one transmission and the beginning of the next to select the next item for transmission. The overhead on an HDLC block is only two framing characters, two block check characters, two block control characters and one control character for each record.

If data traffic is flowing in the other direction as well, there are also acknowledgement characters to be sent, but these are probably best sent as separate blocks so that they can be acted upon as soon as possible. An acknowledgement block will then consist of a total of six characters. Assuming that one acknowledgement is sent for every block and that an equal traffic is flowing in both directions, the overhead is then 21 characters for an average block of 360 characters with 9 records. This gives an efficiency on each circuit of about 95%, or a total efficiency for the link of double that of a half-duplex protocol over the same link. Of course, it should not be difficult for any full duplex protocol to achieve a significantly higher efficiency than a half-duplex protocol over the same link since it will be using both circuits simultaneously.

<u>Complexity</u>

The complexity of implementation for a system based on HDLC full-duplex protocol is the same in terms of the overall system structure as for the half-duplex protocol supporting multiple two-way transmission. In other words, the system will consist of a number of separate parallel activities which must be scheduled in respect of the use of shared resources. In fact, exactly the same supporting software could be used for both systems if a compatible logical record interface were defined to the communications part of the system. Only the communications package would need to be different.

The communications package should itself be somewhat simpler for a full-duplex protocol and consists of two largely independent handlers - one for receive and one for transmit. The minimal amount of interaction necessary between the two handlers to control acknowledgements can be accomplished using store locations accessible to both handlers. There is no requirement for any interaction of execution paths - both handlers can progress independently of each other. The requirement to schedule line use for input or output disappears completely. A system of queues between the user program interface and the line handlers will be necessary in order to provide effective buffering of line activity, but this is the same as that required for the half-duplex two-way protocol.

Since all transactions over the line are in the form of data blocks, the line handler only has to recognize or generate one distinct communication sequence. Also, the error recovery is much simpler. If an input block is found to be in error, then it is ignored and a negative acknowledgement scheduled for output.

No timeout error recovery is needed for the receive handler. The transmit handler will send an enquiry sequence for any block which has not been acknowledged within a certain time and a timeout mechanism will be needed by the transmit handler to control this. If acknowledgements are contained within blocks containing other data, the maximum waiting time for an acknowledgement should be the time taken to transmit two full blocks in the opposite direction, including an allowance for any possible delay between the two blocks. As an example, if the maximum block transmission time is 1 second, a wait time of 2.5 seconds should be adequate. If acknowledgements are transmitted on their own as special blocks, a maximum wait time of 1.5 seconds should be adequate in this example. Blocks that contain only acknowledgements should not themselves be acknowledged.

On the basis of these comparisons between a system using half-duplex two-way protocol and one using full-duplex protocol, it should be possible to support any one application in the same amount of core with either protocol.

12.10 Conclusions on communications protocols

This section has described a small number of protocols from the simple to the sophisticated and some general conclusions can be made about what type of protocol is most suitable in a particular situation.

For straightforward bulk data transmission systems where two-way transmission is not an important requirement, there is no doubt that the one-way point-to-point protocol is the easiest to use and also provides a good line efficiency. If the 'conversational response' feature is implemented as well, this overcomes the main drawback of this type of protocol by providing for operator communication in parallel with data transfer.

This type of protocol can also use the economical dial-up 2400 facility if the volume of data involved is not very large. A system using this type of protocol is the simplest to implement and can be fitted into minimum core size machines. This type of protocol is therefore favoured for simple, low-cost applications. If necessary, higher speed lines can be used to increase the throughput capability provided that a fast enough peripheral is used.

For applications where two-way transmission is essential, e.g. concentrators, or for increasing the throughput of RJE systems by using a number of peripherals in parallel, a full-duplex protocol would be most suitable. A full-duplex system has potentially double the line efficiency and also is less complicated to implement than a half-duplex two-way protocol.

A full-duplex protocol, of course, cannot be used on a 2-wire line, so if there is a requirement to run on dial-up facilities a full-duplex solution is automatically ruled out and a two-way half-duplex protocol would have to be used.

If a two-way half-duplex protocol has to be used, then one of the Multi-Leaving type, which is symmetrical and minimises line turnarounds, would be most suitable. This gives a better efficiency, and the same implementation can be used at both ends of the link.

Chapter 13

SYSTEM PROGRAMMING ON SMALL COMPUTERS WITH HIGH-LEVEL LANGUAGES

13.1 Introduction

The communications package described earlier in this report was implemented using the high-level language, IMP. This proved to be an important factor in making the package machine-independent up to a set of low-level interface routines which had to be hand-coded to use the package on any particular configuration.

The use of a high-level language forces the programmer to think in machine-independent terms since he is not able to make use of any individual features of a particular machine (other than wordlength) except by direct use of machine-code embedded within the high-level coding. If all such code is put into small routines and not included 'in-line' then the main body of the program will automatically have a measure of machine-independence, since there is a strong possibility that the machine-code subroutines can be re-written to provide the same function on a different machine.

On certain systems, the 'user program' part of the communications system was also coded in IMP. Also, the total coding for a system using the 'Multi-Leaving' protocol has also been written in IMP. This included independent activities for a number of different peripheral streams as well as the Multi-Leaving communications package.

These various programs have involved a number of different techniques for using a high-level language as well as a number of different compiler implementations of IMP. The experience gained from this work has been used to formulate a number of general conclusions relevant to the use of high-level languages for system programming, with particular reference to small computers. These conclusions will be described in this chapter.

The reference to small computers is significant because they are becoming increasingly important in computing systems. They are being used in stand-alone mode in more and more applications as well as in remote-connection mode to large systems where they can relieve the main system load by processing small jobs themselves. Also, the time is approaching when even large powerful systems are likely to be constructed out of a number of interconnected small computers, each performing a specific function of the overall system, e.g. control of communications network or control of filing system.

Therefore, anything which makes the job of programming these small systems easier and more efficient is considered important, and there is no doubt that the use of a 'suitable' high-level language can contribute to this. The suitability of a language involves the language itself in terms of the statements and data and program structures that can be used and also involves the compiler in terms of the sort of code produced, the way the object program is produced and the control the programmer has over the environment in which the object program runs. Both these aspects will be considered in this chapter.

13.2 Compilation on a different computer

The programming work done in IMP has been used on three different small computers - PDP-8, PDP-11 and Modular One. Obviously three different compilers were used but they all had one thing in common. This was that none of the compilers actually ran on the computer for which code was being produced. All three compilers ran on much larger computers, namely the ICL System 4 or the IBM 360/370 at ERCC.

Although the three compilers were written in IMP, and therefore capable of self-compilation, they would have produced programs which were far too big to be run on all except large-core configurations of the small computers concerned. The amount of core necessary to run the compiler, in the majority of cases, would have been much more than that required to run the applications for which the small computer was being used. In other words, the compiler would have been the largest single program running on the small computer.

This demonstrates an important point, which is that if the high-level language used is to have a full set of facilities, then the compiler for it is going to be a very large program and will be unable to run on small or medium size configurations. Conversely, if the compiler is constrained to be capable of running on all sizes of configuration, then it will only be possible to implement a limited set of facilities in the language. Furthermore, there would be strong pressures for writing the compiler in assembler language in order to keep the size to a minimum, and this would obviously be undesirable and against all current trends.

Another measure used to reduce compiler core requirements is to segment the compiler into a number of phases such as syntax analysis and code generation and run a multi-pass compiler. This technique requires that some form of backing-store be available and could not sensibly be applied to a computer without it. The compiler could not therefore be used on simple configurations. Anyway, even a segmented compiler can still require a large amount of core for each pass and symbol table requirements etc., can be very significant.

Some figures for core requirements of small machine compilers, some single-pass and some multi-pass, are given in Table 13.1 to illustrate this argument.

The conclusion to be drawn from this argument is that the capability to run in small configurations of the target computer should <u>not</u> be a requirement for a language compiler for small machines. This would be likely to constrain the set of facilities that could be provided in the language. The compiler should be written to run on a large computer, not necessarily the same one as the target machine. Since communications access from small computers to large computers is becoming increasingly more widely used, the problem of remote access to these compilers is easily overcome, using systems similar to those described in the main body of this report.

Even if the compiler is developed on a different machine from the target machine, it is still desirable to make the compiler capable of self-compilation, if possible, to cater for the larger configurations which might be capable of running it. This, of course, assumes that the language being compiled is already available on the chosen large computer.

COMPILER	COMPUTER	NUMBER OF PASSES	APPROX. SIZE
CORAL	MOD 1	2	12k
BCPL	MOD 1	3	8k
IMP	MOD1	1	12k
IMP	PDP-11	1	22k

All sizes are in words, for largest pass.

TABLE 13.1 COMPILER SIZES FOR SMALL COMPUTERS

Use of this method of compiling means that compilers for new small computers can be developed in parallel with the hardware and problems encountered during compiler design can even influence the hardware facilities provided on the small computer. This would also mean that a high-level language would be available with the earliest deliveries of a new computer, instead of the present depresssing situation whereby only an assembler is available in the early days. This would greatly speed up the development of a complete set of software for the new computer and could be used for both basic software, such as assemblers and linkage editors, and applications packages.

In this chapter, examples will be given in terms of four languages currently used on small computers - IMP, CORAL, BCPL, FORTRAN.

13.3 Language Facilities - General Considerations

On the basis of the argument above, the main criteria for the inclusion of a particular facility in a language for a small computer are whether it is useful and whether it can be compiled into efficient object code with the hardware facilities available on that computer. The potential difficulty of implementing the facility in the compiler is also a criterion to be considered, as each new facility increases the size and complexity of the compiler. However, this should not be an overriding criterion, but should merely be used as a guide if desirable facilities have to be ranked in priority order for implementation.

The consideration of 'efficient object code' relates to both space and speed, as small computers are normally short of both. However, savings of space can always be achieved at the expense of speed by the use of semi-interpretive compilation techniques, i.e. by generating subroutine calls to built-in routines for a particular operation or statement in preference to a lengthy sequence of in-line code. Such techniques are commonly used for array indexing, string manipulation and stack operations in the absence of suitable hardware facilities. The use of such techniques can produce very compact, but slow object code. The emphasis in this chapter will be on basic facilities which can be implemented with 'speed efficiency' on almost all existing small computers. For the purpose of the following discussion, the term 'efficiency' then has the following connotation. Where, for a particular statement, the compiler can produce code as good as a hand-coded version of the same statement, this is considered to be an efficient high-level language implementation of the facility embodied in the statement.

This would seem to imply that the use of such a high-level language would be just as efficient as a hand-coded implementation for a complete program. That this is unlikely ever to be the case is caused by the fact that an assembly language programmer can take advantage of special knowledge about the pattern of execution of individual sections of the program in order to make optimal use of registers over that section. Since the equivalent section of code in a high-level language may need several statements, it is very difficult for the compiler to make such optimal use of registers, since it does not have the same special knowledge. The compiler can attempt to optimize over a group of statements by remembering register contents, but, without the special knowledge, is unlikely to make as good a job of it as the assembler programmer. Therefore a complete program can always be coded more efficiently in assembler than a high-level language by an experienced and competent programmer.

However, there are disadvantages with such hand-optimised code as follows. Firstly, it requires experienced and competent programmers who are skilful in the use of the particular machine code, and such people are generally in short supply, whereas an optimizing compiler will optimize for a programmer who has not had to give much thought to the production of optimal code.

Secondly, if it becomes necessary to make modifications to a program, the changes in the machine-code program may upset the hand optimization such that other parts of the code may need to be re-written to allow for this. If modifications are made to a high-level language program, the compiler will automatically adjust the code produced for adjacent statements to compensate for this. There have been many instances of hand-coded programs requiring much effort to modify successfully for this reason. Successful program modification requires a well-structured program and this can be more easily achieved using a suitable high-level language. And when was a program written which didn't subsequently require modification?

The facilities considered in this chapter are oriented towards use of the languages as a system programming tool. Although some of the facilities will also be useful in applications programs, facilities related only to applications programs, such as real arithmetic, are not considered. On a general-purpose machine there are also two types of system program - those which are necessary to control the operation of the hardware such as device drivers, schedulers, interrupt handlers, which normally involve multiple execution paths and asynchronous events and which the user is not normally aware of, and those which enable the user to make use of the system, such as compilers, assemblers, editors, linkers, input/output routine library, etc. This latter category is really system support software and does not require multi-threading or asynchronous activity. It is normally considered by the system to be at the same level as the user program. It is the former category, that of the low-level system software, which will be considered here. This is the type of software that is needed for the small computer when used in a dedicated or special-purpose application, such as real-time control.

Of the three categories of software - applications, system support (or 'middleware') and system - considerable attention has been given to the use of high-level languages for the first category, a certain amount to the second, and relatively little to the third, with some notable exceptions such as Burroughs MCP and the EMAS and MULTICS systems. Eventually, the use of high-level languages will become common in all three categories. It is likely that facilities suitable for one category will not be suitable for the others. This will become apparent in the remainder of this chapter.

A general point worth making is that system programs need a more intimate level of access to the actual machine than do other programs. For this reason, language facilities which are necessary for system programs may have little use in user programs. In fact, some of the facilities which will be recommended, such as manipulation of machine addresses, are potentially dangerous for ordinary user programs. Where such facilities are recommended, it is intended that they should be used only by programmers who appreciate the full consequences of using them. It is certainly the case than an inexperienced user could hang himself through indiscriminate use of the low-level facilities. Such facilities are, however, essential if the system programmer is to have full control over the machine and not be prohibited from making use of the full capabilities of the hardware.

13.4 Data Elements

Only those data elements which can be directly referenced and manipulated by the machine should be implemented. All small computers can access and manipulate data in whole word form and the different ways in which this basic unit can be manipulated determines the allowed data types. A type of INTEGER should be applicable on all small computers, with the maximum value being determined by the particular word size.

Other data types such as BYTEINTEGER or CHARACTER should not be implemented unless the machine can manipulate information in byte form. Similarly, type STRING should not be implemented unless the machine can manipulate information in character form. For some uses of character strings, e.g. storing fixed messages, it is more efficient to store the characters in packed form, whereas for other uses involving moving and comparing strings, it is more efficient to store them in unpacked form. It would be very difficult for the compiler to decide which was most efficient in a particular case, and so it is best left to the programmer who knows how particular strings will be used in the program.

A number of different representations for integer constants should be allowed e.g. decimal, octal, hexadecimal, binary, character, packed character. These can all be converted to integer types as far as the object program is concerned and therefore introduce no inefficiency into the object program, while, at the same time, increasing the convenience for the programmer and making the source program more intelligible.

It should be possible to initialize any declared variable at compile time, using a facility such as OWN. The use of a facility such as CONSTINTEGER (IMP) or MANIFEST (BCPL) should also be allowed, as this can frequently result in a saving of core space and execution time through the use of an 'immediate' or 'literal' operand.

13.5 Data Structures

A large amount of information used in a system program is in the form of tables and lists. Since all small computers provide a means of addressing tabular data, facilities for representing this should be included in the language. The two addressing techniques most commonly used are indirect addressing and indexed, or base plus displacement, addressing. The second method requires that the machine has one or more index or base registers.

Using either of these addressing technique, an ARRAY facility can be implemented efficiently but using only the data types that can be directly manipulated, e.g. INTEGER.

Another facility that can be used to provide efficient addressing of tabular data is the IMP RECORD facility. At the worst, this can be treated as a simple ARRAY for which all index values are known at compile time, and elements of the RECORD referred to with indirect addressing. A RECORD can be more efficiently accessed if indexed addressing is available, because the relative position of any RECORD element is known at compile time. Valid record elements are INTEGERs, INTEGERARRAYs and RECORDs themselves as well as any additional data types which may be implemented.

An ARRAY of RECORDs can also be efficiently implemented using facilities already available for ARRAYs and RECORDs. A list of control tables is something which occurs very frequently in system programs, e.g. device control tables, program control tables, and these are most clearly represented by RECORDARRAYs, which allow very efficient access.

Both RECORD and RECORDARRAY are facilities which are exceedingly useful in system program construction and avoid the use of cumbersome methods of accessing fixed format tabular data through the use of ordinary ARRAYs. Both facilities should be provided for all small computers, as in no cases do they introduce any inefficiences and in some cases they can make the referencing of tabular data more efficient. They also increase the intelligibility of the source program.

It should be possible to initialize the contents of ARRAYs and RECORDs at compile time in all cases except where dynamic bounds are used.

13.6 Machine address manipulation

The ability to manipulate actual machine addresses as used by the

CPU is also an important requirement of system programs. This is necessary for control of peripherals in order to specify buffer addresses, to access store areas not defined within the program and as a means of communicating information to another program. Such facilities are provided by the IMP ADDR and built-in mapping functions and also by the LOCATION and 'anonymous reference' facility of CORAL. It is also provided in BCPL by means of the @ and ! operators.

The ADDR function can be implemented on all small computers, since machine addresses are needed during normal program execution for things like parameter passing by NAME. Whether they are stored as constants or generated dynamically at run-time depends on efficiency considerations for a particular machine. Once computer addresses can be generated, the inverse function of referencing unnamed locations (mapping) through the use of such addresses is accomplished using indirect or indexed addressing, and at least one of these facilities is available on all small computers.

These two facilities provide the means to access single locations by machine address. It is also useful to be able to access a whole table of data by means of a pointer to the start of the table. The IMP NAME feature provides this facility for RECORDs and for ARRAYs. In BCPL any variable can be used to point to a table of data. The use of the RECORDNAME facility enables repeated references to one member of a RECORDARRAY to be made more efficient by assigning a RECORDNAME to that member and treating it as an ordinary RECORD. This is obviously a useful feature in system programs where a lot of data is kept in lists of tables.

Assuming that the RECORD descriptor is just a single word, then the RECORDNAME assignment can be simply implemented by copying the descriptor of the RECORD referred to into the NAME variable. Once the assignment has been made, the elements of the record can be accessed in the same way as an ordinary RECORD.

The assignment of an ARRAYNAME is slightly more complicated if a dope vector is being used as an array descriptor. A new dope vector must be created to fulfil the assignment. However, if the assignment is to a single-dimension ARRAYNAME and array-bound checking is not in force, then a single word descriptor will suffice here as well.

13.7 Program Structure and Statements

The previous section considered the possible types and arrangements of data that could be efficiently implemented on small computers. This section considers the types of program statement and the statement constructions in a complete program which are desirable and which can be implemented efficiently. Facilities which are thought to be desirable are those which permit a clear description of the problem being programmed, those which permit efficient use of storage and those which aid the debugging process.

Some general points can be made here. It is important with systems programs, some parts of which may be time-critical, that the programmer have some kind of feel for the likely efficiency of his program, when using different types of statement. This assumes that he has some familiarity with the general hardware structure of the machine in order to see how particular operations might be programmed in assembler. Because of this, it is important that there is not a lot of 'behind-the-scenes' activity invoked by the object program, whose presence might not be suspected by the programmer. This applies to such things as compiler-generated run-time diagnostics, dynamic-storage allocation schemes, and so on. Although run-time diagnostics facilities are very convenient during program development, it should be possible to opt for little or no diagnostics when a program is in production use.

Also there may be cases where it is expedient to sacrifice space for speed in the object program, by requesting the compiler to generate in-line code for particular operations by means of special compiler control statements. Such facilities make the compiler more complicated at no extra cost to the overall object program. However, as stated previously, compiler complication is not an overriding factor if the facility is useful to the small computer.

13.8 Expressions

There should be no restrictions on the complexity of expressions which can be written. Techniques for expression-compilation are now sufficiently well-developed that a compiler can produced code which is as efficient as that produced by hand-coding, for single and multiple-accumulator machines. Any arbitrary restrictions on expression complexity such as in certain FORTRAN statements will only force the programmer to write complicated expressions as a series of simpler ones, which will not results in any more efficiency.

Since type 'real' is not being considered in this discussion all expressions will evaluate to type 'integer'. Therefore, wherever the syntax allows the use of an INTEGER, it should be possible to use an expression, e.g. as an array index, as a value parameter in a procedure call, etc.

All these facilities are fully supported in IMP, CORAL and BCPL. This gives the programmer the capability to specify the problem in the most suitable way. It is suggested that the full range of operators, both arithmetic and logical, be made available. On some small computers which have no hardware multiply/divide and only very limited logical operations this will inevitably lead to long execution times for these operations. However, if the computer does not have the relevant hardware, these operations are also slow when hand-coded so there is not necessarily any inefficiency introduced through the use of compiled code. This, however, is one situation where it would be desirable to have a choice between the use of subroutines and in-line code so that the programmer could opt for maximum speed where necessary.

13.9 Conditional Statements

The general form of the conditional statement should be permitted as follows:-

IF <condition> THEN <statement> ELSE <statement>

where <condition> may be either a simple condition or a compound condition involving the use of AND and OR;

where <statement> may be either a simple statement or a compound statement consisting of an arbitrary sequence of statements (including further conditionals) appropriately bracketed, e.g. by BEGIN-END in CORAL or START-FINISH in IMP.

Such a facility allows a complicated series of decisions to be programmed without the use of programmed jumps. The absence of programmed jumps keeps the path of execution under greater control and makes debugging easier since there is only one route to any particular statement and it is easier to follow the path of execution. This tends to produce a better-structured program, more modular and easier to modify than one with a lot of explicit jumps which can produce a monolithic tangle of interlocking execution paths.

This capability is present in IMP, CORAL and BCPL. The absence of this capability is one of the more serious deficiences of FORTRAN.

The use of this capability should not introduce any inefficiences into the object program when compared with the use of simple IF statements (as in FORTRAN) and programmed jumps. All that happens is that the compiler will generate jumps automatically where the programmer would otherwise code explicit jumps.

The inclusion of the facility obviously implies extensive use of recursion in the compiler. This is likely to be the case anyway if modern, syntx-directed compiling techniques are used. It is from the use of such techniques that the generality and lack of arbitrary restrictions in present-day languages derives.

There are a number of additional variations on the conditional statement which allow certain conditions to be expressed more clearly. These are such things as UNLESS as an alternative to IF, two-sided conditions (e.g. 0< x <10) and reverse conditional statements, of the form <simple statement> IF <condition>. These have a certain advantage over the use of the standard conditional in particular cases but are not considered to be of major importance. These features are therefore desirable from the standpoint of achieving maximum clarity but do not rank high on the priority list.

13.10 Programmed Loops

One of the major advantages of a program definition of a problem derives from the repetitive execution of particular sequences of instructions. If it were not for the capability of executing the same sequence of instructions again and again, there would be no point in going to the trouble of writing a program to solve the problem. Looping is therefore an essential element of program construction and if there are no facilities defined in the language, the programmer has to program the loops explicitly by using conditionals and programmed jumps. If looping facilities are included in the language, the compiler can generate these tests and jumps just as efficiently as the programmer, and in some cases more efficiently by making use of facilities specially incorporated in the hardware for loop control, such as increment/decrement and skip if zero, etc. The provision of loop control statements therefore reduces the number of explicit program jumps, which is an important consideration.

The loop control statement should permit the WHILE or UNTIL form as well as the more conventional FOR N FROM L BY K TO M form since some program loops cannot be expressed by the latter form and the programmer would have to resort to explicit tests and jumps again.

13.11 Programmed Transfers

The explicit programmed jump is the standard mechanism for altering the execution sequence to another point in the program. Extensive use of program jumps can easily produce a program whose path of execution is difficult to trace back from a statement where a fault has occurred thereby making debugging more difficult. It can also produce non-modular programs which are difficult to modify since the relation of one section of code to another is not easy to see.

Many of the facilities recommended above are intended to reduce the need for explicit program jumps, if not eliminate them altogether. It should be possible to program entirely without the use of jumps if the facilities suggested above are provided, and if subroutines are used whenever the same section of code is required in more than one place. This may involve a greater use of subroutines involving only a small number of statements. However, provided the use of subroutines does not incur any great inefficiencies, then this is not a disadvantage. The use of subroutines will be covered later.

Another advantage to be gained from minimum use of programmed jumps is where the compiler is attempting to perform optimisation of the compiled code by remembering the contents of registers between statements. This produces savings where the same variable or constant, array or record base is used in consecutive statements, since the compiler can use the value in the register rather than loading it from store. This can only be done when program execution proceeds sequentially from one statement to the next. As soon as a statement with a label is encountered, the register contents must be forgotten because the statement may then be executed out of sequence. Therefore the less use of explicit labels, the more effective will be the compiler optimisation.

There is one area where the use of programmed jumps does confer distinct advantages and this is with a 'computed jump' or SWITCH facility. This provides a rapid means of decision-making when the different values of the main decision variable conform to some kind of numerical sequence, e.g. 1,2,3, etc. or A,B,C. This can lead to considerable efficiencies over the use of an explicit test for each individual case.

13.12 Machine-code

For systems programs, where it is often necessary to access directly particular elements of the hardware on a computer, it is essential to be able to resort to direct machine-code programming when necessary. This applies to such things as issuing physical commands to peripherals, accessing processor status registers, and so on.

These operations cannot be expressed in a high-level language in a way applicable to all machines since this is one area in which computers are likely to differ markedly from each other. Although all such functions should be confined to a small number of well-defined routines, it is still convenient to be able to write them as syntactically part of the high-level language program rather than as a separate machine-code library which is linked in at the object program stage. This has the advantage that subroutine linkage conventions can still be handled by the compiler and that normal high-level language statements can continue to be used where machine-code is not absolutely necessary, e.g. for loop control or conditionals.

There may also be cases where maximum speed of a particular section of code is essential, and this can be achieved only by careful hand-coding.

Where in-line machine code is used, it is important to have full access to any variables or named entities declared in the high-level language. Otherwise, communication between the high-level language and the machine-code would not be possible.

13.13 Routines and Functions

Modularity is an important objective in any program, not just system programs. To this end, the capability to break down a large program into a number of separate, independent pieces with well-defined interfaces is very important. This is the facility for routines or procedures, with functions or value procedures being a particular variation. The importance of the subroutine concept is demonstrated by the fact that even the smallest computer has a hardware instruction for a subroutine call. The extent to which the hardware assists with other aspects of subroutine use varies considerably between computers. This relates to such things as passing parameters, saving the current register context.

Because of the minimal assistance given on most small computers, the implementation of subroutine facilities must be very carefully done if it is not to impose considerable overheads, merely to effect the subroutine entry and exit. This is one area where it is possible to incur considerable 'behind-the-scenes' activity without the programmer being particularly aware of it, which is considered to be a bad thing for reasons given previously. This is particularly true if the classical ALGOL-type techniques are used.

In an ALGOL-type scheme, every routine is assumed to be potentially recursive and the local workspace for the routine execution is assigned dynamically each time the routine is entered. Also, the local workspace of the calling sequence must be preserved and its location remembered for later use. The routine exit sequence then has to restore the working context of the calling code. This is usually implemented on an ALGOL-type stack arrangement, and can involve lengthy manipulations for the saving of registers and stack pointers on any machine not specifically designed for ALGOL-type dynamic storage allocation, such as the Burroughs B5500 and its successors. Certainly, there is no small computer which supports this type of storage organisation in its basic hardware and a dynamic stack has to be implemented by software means which involves considerable overheads for routine entry and exit.

A more efficient system for small computers would assume that routines were non-recursive so that local workspace could be assigned statically at compile time and would always be at the same place at run-time. The capability for recursive routines could still be provided through an explicit declaration, e.g. RECURSIVEROUTINE, and the compiler could then generate alternative entry and exit sequences that allowed for dynamic assignments of workspace. The programmer would thus have some control over program efficiency. Since system programs have little use for recursion anyway, it would represent a potential improvement in efficiency for small computers if recursion were taken to be the exception rather than the rule. Both IMP and BCPL assume recursion at all times while CORAL assumes non-recursion unless told otherwise.

The use of statically-assigned workspace for routines is liable to use more storage than the dynamically-assigned case for the same program. This is because all routines in the program are unlikely to be active on a particular execution path and statically-assigned workspace is not available for anything else if the routine is not using it. However, to offset this is the fact that more workspace is needed per routine in the dynamically-assigned case to save the context and stack pointers relevant to the calling sequence. The differences in storage requirements in a particular case would be difficult to predict without a detailed examination of the execution paths.

13.14 Block Structure

With static storage assignment advantage can be gained from the intelligent use of block-structuring, i.e. use of BEGIN-END to bracket sections of program which are independent of each other except through common use of variables declared at outer lexical levels. BEGIN-END blocks at the same level can never be active at the same time and therefore can share workspace. (The statically-assigned local workspace is not the same as used in FORTRAN where values are assumed to be preserved between entries).

This type of storage organisation is that defined for CORAL and produces a tree-like storage allocation structure whose dimensions are all known at compile time. This corresponds exactly with standard FORTRAN OVERLAY schemes and can be represented diagrammatically as in Figure 13.2, which gives the storage layout for the program in Figure 13.1.

```
%INTEGER I1,I2,I3
%INTEGERARRAY A1,A2(1:10)
%ROUTINE R1
    %INTEGER I4,I5,I6
    ! <EXECUTABLE STATEMENTS.>
    %END
%ROUTINE R2
    %INTEGER I7,I8,I9
    ! <EXECUTABLE STATEMENTS.>
    %END
! <EXECUTABLE STATEMENTS.>
%BEGIN
    %INTEGER J1,J2,J3
    %INTEGERARRAY B1(1:100)
    %ROUTINE R3
        %INTEGER J4,J5,J6
        ! <EXECUTABLE STATEMENTS.>
        %END
    %ROUTINE R4
        %INTEGER J7,J8,J9
        ! <EXECUTABLE STATEMENTS.>
        %END
    ! <EXECUTABLE STATEMENTS.>
    %END
%BEGIN
    %INTEGER K1,K2,K3
    %INTEGERARRAY C1(1:50)
    %ROUTINE R5(%INTEGER A,B,C)
        %INTEGERARRAY C2(1:20)
        %INTEGER K4,K5
        ! <EXECUTABLE STATEMENTS.>
        %END
    %ROUTINE R6
        %INTEGER K6,K7
        ! <EXECUTABLE STATEMENTS.>
        %END
    %ROUTINE R7
        %INTEGER K8,K9
        ! <EXECUTABLE STATEMENTS.>
        %END
    ! <EXECUTABLE STATEMENTS.>
    %END
%BEGIN
    %INTEGERARRAY D1(1:5)
    %INTEGER L1,L2
    %BEGIN
        %INTEGER L3,L4,L5
        %INTEGERARRAY D2(1:100)
        %ROUTINE R8
            %INTEGER L6,L7
            ! <EXECUTABLE STATEMENTS.>
            %END
        ! <EXECUTABLE STATEMENTS.>
        %END
    %BEGIN
        %INTEGER M1,M2,M3
        %ROUTINE R9(%INTEGER X,Y,Z)
            %INTEGER M4,M5
            ! <EXECUTABLE STATEMENTS.>
            %END
        ! <EXECUTABLE STATEMENTS.>
        %END
    ! <EXECUTABLE STATEMENTS.>
    %END
%ENDOFPROGRAM
```

FIGURE 13.1 EXAMPLE PROGRAM FOR STORAGE LAYOUT

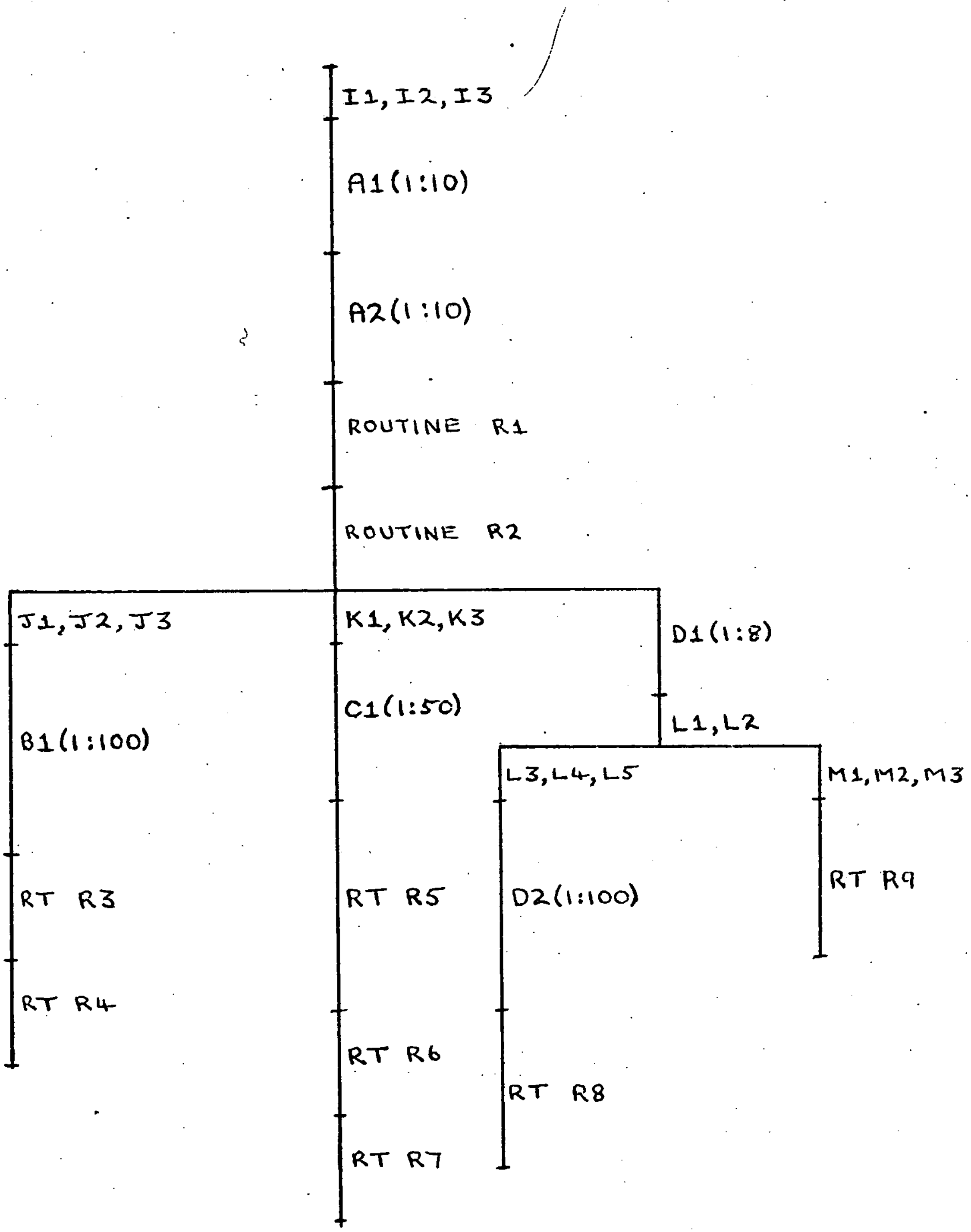


FIGURE 13.2 STATIC STORAGE ALLOCATION SCHEME

This kind of storage allocation is considered to represent an optimum balance between speed and space for system programs on small computers where recursion is not used. Of course, if recursion is necessary then a proper stack mechanism must be provided but the space for that would come outside and in addition to this static arrangement. If there is a requirement for dynamic array bounds at run-time, this also would have to be handled by a proper dynamic storage allocation scheme outside the static storage allocation.

The important thing about the static arrangement is that it gives the programmer some control over the way that space is used and allows him to make a trade off between space and speed in any particular application.

A further advantage of static storage allocation is that the compiler knows the amount of storage which will be needed by the program at run-time, and can inform the programmer of this. For a resident system program, it is essential to know this information so that the storage can be properly allocated within the system area. In the dynamic storage case, it is impossible for the compiler to predict the amount of store needed, and this has to be found out by a process of trial and error while running the program or by a careful examination of the possible execution paths of the program and a knowledge of the amount of store needed for each routine, which the compiler can generally predict.

Even for normal user programs, the inability of the compiler to predict the amount of store needed for execution in the dynamic case causes difficulties on a machine with no hardware assistance in the dynamic storage allocation. The tendency is for users to request more store than is actually needed in order to ensure that the program does not fail.

13.15 Run-time environment

Whenever the compiled code of a high-level language program is actually run, it is always accompanied by a small body of code which provides the necessary run-time environment of the program. This is commonly referred to in Edinburgh as PERM and provides such facilities as initialization of registers and stack pointers, etc., prior to entering the program, dynamic storage allocation (where used), routine entry and exit, array indexing and array bound checking, code for high-level language operations whcch are too complicated to execute in-line, run-time diagnostic checking, and so on. The user program is not normally aware of this, as it is included automatically as part of the running of his program.

For system programs, there may be requirements which render the standard PERM unsuitable. For instance, it may include a number of facilities which are not needed by a developed system program, such as run-time checking. Also, some of the high-level language code may be entered asynchronously as the result of an interrupt. In this case, PERM needs to be re-entrant and there must be no interference between the interrupt code and the interrupted code. In such cases, it is necessary to write a special PERM which is applicable to the particular run-time environment.

In order to do this easily, there must be a well-defined interface between the compiled code and the run-time environment in terms of the functions to be provided. There must also be a mechanism for dispensing with the standard PERM and using the special PERM instead, without changing the compiler itself. Therefore, the compiler should produce only the compiled code, with a set of unresolved external references to PERM which can be included subsequently.

One common way in which this is done with small computers is for the compiler to produce an assembly code version of the program with symbolic references to the required PERM functions. The special PERM (also coded in assembler) can then be added to the compiled code and the complete package processed by an assembler which automatically resolves the references to PERM.

This then allows the system programmer complete control over the run-time environment and he can, for example, implement a different sort of storage allocation scheme if the standard one is not appropriate for the particular application.

13.16 Conclusions

Many of the remarks made in this chapter can be applied to any programming application and not just systems programming. The set of facilities suggested would provide an effective programming tool for any small computer and it is considered that they can all be implemented with reasonable efficiency compared to hand-coding. Facilities which require special hardware features for efficient implementation, such as string manipulation and dynamic storage assignment, have been deliberately omitted.

Certain facilities, such as the general ability to reference by name entities defined outside the program (external references), apart from standard library routines, have not been considered although they become important for large, complex suites of programs which cannot sensibly be compiled in one operation.

The facilities needed for sound, basic program construction only have been considered. Undoubtedly other facilities can be added which have application in particular circumstances. Also, various frills can be added to provide different syntactic ways of specifying the same semantic construction, such as the alternative forms of conditionals mentioned. Additional facilities could be recommended if extra hardware were available, e.g. byte addressing.

The facilities which are thought to be the most relevant to system programs, and perhaps not so relevant to applications programs, are the store mapping and machine addressing capabilities, the RECORD, RECORDARRAY, RECORDNAME, and ARRAYNAME facilities, the ability to specify options to the compiler to generate in-line code or subroutine calls for particular operations, the capability for the programmer to exercise more control over the use of storage in his program, and the facility for in-line machine-code.

Of the various languages considered, BCPL had the largest collection of facilities considered relevant to system programs, although this is partly because IMP as presently implemented on small computers (e.g. PDP-11 SKIMP) does not have some of the facilities of full IMP which it is thought could be implemented such as RECORDS. If this shortcoming were corrected, then IMP and BCPL would be about equivalent. Neither of them however, have the static storage assignment capability as used by CORAL, which is important as a means of controlling speed overheads where timing is critical, as in Interrupt-handling.

Chapter 14

CONCLUSIONS

14.1 Introduction

The work described in this report relates to two distinct areas of computing. One is the area of telecommunications between computers, the uses to which this can be put and the techniques needed to make the connection effective. The other area is the one of techniques for a more sound approach to the construction of complex systems by the development of standardized modules applicable in a wide range of environments. The use of such techniques is a step towards the building of complete systems from 'off-the-shelf' components, instead of building each new system from scratch as at present.

This chapter attempts to summarize the main points of the work described which are relevant to these two areas. As such, it complements the remarks made in the introductory chapter listing the areas of work which would be studied. Also, this chapter makes some predictions about possible future developments in these two areas by extrapolating from the results and conclusions actually obtained.

14.2 Uses of Computer Telecommunications

The direct outcome of the work described in this report has been the establishment of several successful computer communications links involving a number of different small computers. The list given in Chapter 10 (Table 10.1) describes those connections completed as at October 1973, and there were also many other potential connections at that date. The system developed has thus achieved one of the stated objectives - that of a wide range of applicability.

Of the connections listed, the majority of the small computers are used to a large extent for local processing and only use the communication link to supplement their own resources and facilities. The two computers are thus working 'in tandem' to solve problems that would not be suitable for either computer on its own. The link therefore supports a genuine co-operative operation between the two computers, with each doing that part of the overall task best suited to it. The small computer is not merely acting as the dumb satellite of the large computer but has its own particular contribution to make.

This method of use is considered to be the way in which computer communication systems will develop in the future to provide significantly increased facilities to all computer users who can participate in such developments. Whenever the topic of computer networks is discussed, the subject of 'load-shedding' seems to be a favourite application for such networks. In the opinion of the author, the concept of simple load-shedding is considerably over-emphasised and is not nearly so simple to achieve in practice except in special situations. The idea of a computer dynamically off-loading work to another computer which is considered to be under-loaded is rather naive.

First of all, most computers tend to have their peak loads during the same periods of the days for obvious reasons related to the normal working hours of the computer users. Secondly, there are not many jobs capable of running on more than one machine without change. This applies even to machines of the same type, which support the same compilers and library facilities. There are still likely to be differences in Job Control Language, which may use optional facilities peculiar to one installation.

Furthermore, any job which accesses permanent files can only be sensibly run on the computer on which the files reside. Any alternative will only become feasible when data communication speeds become comparable with those for standard file residence devices, i.e. upwards of 1 Megabit per second. Even assuming that it is possible to detect those jobs which do not refer to permanent files by examining the Job Control Language, on many systems it is possible for one job to generate a second job, whose file requirements could be quite different from the original job.

The objective of using computer networks for automatic load-shedding is therefore difficult to achieve in practice and could only be applied to a restricted set of jobs.

The full potential of computer networks will only be achieved when the user who is submitting the work is involved in the process of deciding the best place to run it. In other words, transfer of work in a computer network should only be done under explicit instruction or advice from the user. The real purpose of a network should be to provide a wider range of facilities than could economically be provided at one installation and then make it easy for the user to access them. The user will quickly decide on the best way of getting his work done, using the machine that is most suitable for any particular task.

This is one of the important conclusions that has come out of the work on linked computers. If proper note is taken of the capabilities present at both ends of the link, then a more satisfactory facility can be achieved than if one end just acts as a dumb satellite.

14.3 Technical Aspects of Data Communications

The other relevant aspect of the telecommunications work relates to the communication techniques used to obtain an effective connection. It was concluded in Chapter 4 that the synchronous method of communication was more appropriate to computer-computer operation because of the requirement for fully automatic detection and recovery from error conditions and the higher speeds possible than for human-oriented communications. The asynchronous method of communication has advantages of cheapness and simplicity when applied to low data-rate systems, but these advantages do not carry over when applied to higher-speed computer-computer communications.

Given the requirement for fully automatic error detection and recovery, some kind of communications protocol is necessary. The one implemented was the simplest sort with the intention of being compatible with different main computers (see Chapter 5). This was achieved in the limited Edinburgh environment and it seems likely that it could also be achieved in a wider environment if ever the need arose, especially if the modifications to the communication software suggested in section 9.7 were carried out.

Although the implementation of only one type of communications protocol was described in detail in this report, a study of other, more complicated protocols was carried out. A comparison between them was given in Chapter 12. As far as is known, this represents the first attempt at a critical comparison between these different types of protocol in judging their suitability for different applications.

In view of the varying levels of complexity associated with these different protocols for different requirements, it seems unlikely that any future industry-wide protocol can be uniformly applied to all situations. Such a protocol would have to be sufficiently powerful to handle the more sophisticated systems, and this would render it unnecessarily complex for systems with simple requirements. Any universal protocol, therefore, would have to be defined at a number of levels of complexity, from the simple one-way-at-a-time half-duplex system to the powerful multi-stream full-duplex system.

All such levels should be compatible in respect of the unit of physical transfer over the communications line, i.e. the block. The block-framing characters and the type of redundancy check used should be the same at all levels so that any particular terminal or computer can equip itself with hardware which will be able to autonomously assemble a block of data from the line. The interpretation of the contents of the block may then be dependent on the particular level of protocol in use and is analyzed by software. A terminal or computer can then upgrade the level of protocol it supports by software changes without new hardware being necessary.

This degree of flexibility can only be achieved at present either by having minimal hardware which merely assembles characters from the line and leaves the rest to software or by having complex hardware that has a large number of software-controlled options, typically implemented with a microprogrammed controller, which therefore becomes a very expensive solution. If everyone could agree as to what constituted a data block, then it would be much easier to handle communication between dissimilar machines.

A significant amount of effort (described in Chapters 6 and 7) was directed towards the design of a small computer synchronous communications controller capable of being used to communicate with any type of main machine. Because of the differences between the communications connections of these various main machines, this objective could only be achieved economically by making the hardware very simple and imposing a considerable burden on the software in respect of making sense of the sequences of characters that were assembled off the line. The hardware was therefore capable of handling anything synchronous as long as it was in 8-bit format.

Since the software has to interpret the characters at interrupt level, this can present considerable timing problems for the software, especially at higher speeds, e.g. 9600 baud. It renders communication at really high speeds, e.g. 48 kbaud, impossible except on fast processors which are not doing anything else of higher priority.

As was suggested above, this software burden could be removed if there was a standardized block format acceptable to all equipment. Such a thing is defined in the ISO HDLC protocol recommendations(28) and is designed so as to be easily implementable at the serial-to-parallel conversion level. Generation and recognition of this type of block format would be quite simple in hardware. It would thus be feasible economically to develop an autonomous block transfer controller for this type of block format for small computers, in the knowledge that it would be applicable to all synchronous communications systems.

This would relieve the software of all time-critical interrupt responses in the milli-second region. It would just be necessary for the software to generate a response to the whole block, and the critical time for this is of the order of seconds rather than milli-seconds.

One thing this report has shown is that communication with main computers is more complicated than it needs to be. The different standards in use involve the small computer software in some fairly intricate manoeuvres in order to handle the different message sequences used by the main computers. Some agreement on a standardized unit of physical transfer would make computer-computer communication much easier. An analogous situation occurs in respect of so-called 'industry-standard' magnetic tape. This can be used as a means of transferring data between dissimilar machines even though special utilities may be needed to unscramble the data once it has been read in. The important thing is that standard hardware is available on different machines which will read the same block from the same magnetic tape and the problem of data transfer then becomes amenable to a software solution.

14.4 System Construction Techniques

The other main area, outside of the specialist area of telecommunications, in which work has been done is that of software engineering or system construction techniques. The impetus for this was the requirement to develop a system that would run on different small computers in a wide range of environments.

This objective has certainly been achieved in practice, on the basis of the number and variety of systems now operational (see Table

10.1). A directly useful outcome of the work is that we now have an "off-the-shelf" component of system software. This can be taken by anyone and plugged into a new system without modification. The detailed information contained in Chapters 9, 10 and 11 should be sufficient to enable anyone to install this communications component into a new system or new computer without understanding how it works. All that is necessary is to produce the layer of interfacing routines between the communications software and the real environment, and the requirements for these routines are well-defined.

Also specified are test procedures for ensuring that the interfaces perform their functions correctly before connecting everything together into a working package. Also, if the communications software itself has to be re-translated for a new computer, a standard mechanism is given for testing out the logic of it in a controlled way before applying it in the real-time environment. Therefore, by using a certain amount of intelligence in implementing the interface routines and following mechanically a prescribed series of steps, it is possible to add an important new functional capability to a system - that of communicating with another computer.

14.5 Transferable System Components

This capability for building up a set of basic system software from components developed on other computers is not really possible at the present time because the system components have not been designed to allow this to be easily done. In order for this to be possible, components must be designed in a machine-independent way with clearly defined interfaces wherever they interact with the rest of the system.

There are some examples of software outside the basic system level which have been designed in this way. One of these is the BCPL compiler(25), which compiles BCPL to a hypothetical machine code, OCODE, which must then be interpreted or translated for a real target machine. The compilation of BCPL to OCODE is strictly machine-independent and OCODE is clearly defined, so that there is a well-defined interface between the machine-independent part and the real machine. A similar system is used in the Edinburgh SKIMP compiler.

Both these compilers have been applied to a number of different target machines by re-writing the interface between the virtual machine code and the real machine. However, there is no evidence of similar techniques having been applied to basic system software, i.e. device drivers, interrupt handlers, schedulers, etc.

The idea of having hardware-driving software that is machine-independent is novel, but has been shown to be applicable in the work described. There is no reason why similar ideas should not be applied to other parts of the system, e.g. disc handler, drum handler, scheduler. They can be coded once in a high-level language in terms of an idealized hardware interface and then mapped onto the real hardware in the way described. There is no reason why a disc handler coded in this way for one system would not be applicable to another system. The characteristics of all moving-head discs are broadly similar. Problems in arm scheduling, error recovery, rotational scheduling, etc. can all be solved once in a particular way, and the same solution would be applicable to any other moving-head disc. Currently, these problems are being solved, coded and tested many times over on different systems, and the solutions being used are probably very similar. However, because of the structure of current executives, there is no possibility of plugging in modules developed elsewhere.

Therefore, although the existence of the transferable communications software is useful in itself, it is hoped that the detailed description of its conception and its development will serve as a useful guide for the production of other transferable system components. It is difficult to give a specific set of rules to be followed but the following is a summary of the important points.

14.6 Interfaces

Probably the most important aspect is the careful definition of interfaces to everything that the component is going to interact with. The existence of clear interfaces is what makes it possible for someone to use the component without understanding how it works. It also facilitates testing, since simple test programs can be devised to exercise the interfaces. An interface should be defined in as general a way as possible and should be as simple as is consistent with the function required.

For each component, there exists what can best be described as a set of 'natural' interfaces to the environment. This rather inprecise notion can best be illustrated by some examples.

For the communications system described, the natural user interface was a block-oriented interface. This would also be a natural interface for any physical device whose transfers are blocked, e.g. disc, tape. This is, of course, only one level of interface. Another level of user interface can be defined at the logical-record level, but this is not an alternative to the block interface, rather an addition to it. Software to support a logical interface would sensibly make use of a block interface provided by another component.

Similarly, the interrupt interface to the communications package, consisting of RECEIVE, TRANSMIT and ANALYZESTATUS, is a natural interface for a two-way single-character communications channel. Also, the four functions used for software control of the hardware, namely READDATA, WRITEDATA, READSTATUS and WRITECONTROL, from a natural interface for peripheral control and should be applicable to any peripherals.

A natural interface to a component is not always immediately apperent and it may require a number of design iterations before the right one is chosen.

It is only through the wide acceptance of and adherence to the use of universally applicable interfaces for separately identifiable functions that systems can be easily constructed from ready-made components. The development of software engineering as a useful discipline depends to a large extent on the use of such standard interfaces.

14.7 Real-time components

The use of a finite-state machine representation of components that have to respond to real-time events, e.g. interrupts, eases the testing problems considerably since components can be thoroughly tested in a controlled environment before being used. Assuming that the rest of the system has been properly structured in respect of communication between asynchronous processes and that system loading permits adequate response times, then the whole system should perform correctly without odd hangups sometimes associated with real-time systems.

14.8 Use of high-level languages

The use of a high-level language is a very important aspect of the production of transferable software components. Even if the language cannot be directly compiled for a particular target machine, the high-level language coding serves as a system description and as an authoritative definition of the implementation. The use of a high-level language also avoids the temptation to take advantage of special hardware features available on one machine, which tends to produce a specific rather than a general solution.

If a high-level language is to be used, then a careful choice has to be made since all languages or all compilers are not suitable. Due attention must be given to efficiency considerations and run-time environment, as well as language features. These considerations were discussed in chapter 13.

14.9 Transferable Hardware

Although most of this discussion has been concerned with the software part of the communications system, it is worth noting that a specification was also given for a communications hardware component that was intended to be easily transferable between machines. This was again achieved by the careful choice of an interface which was capable of being implemented in a compatible way on a number of small computers (see Chapter 6). The general form of the interface defined would be applicable also to other peripherals using a single-character transfer. The use of a universal interface such as the one proposed would make it easier to interface new peripherals to a wide range of small computers. It is also possible that this interface could be extended to handle autonomous transfer devices in a uniform way.

14.10 System Construction in the Future

It is envisaged that in the future it should be possible to construct a complete set of system software from off-the-shelf system components instead of having to write a new system from the bottom up as at present.

Currently, the software for a computer can be divided roughly into three levels - applications software, system support software such as language processors, basic executive to control the use of the hardware. Of these three levels, only the first currently has a large body of software that is transferable between machines, through the widespread use of languages like FORTRAN for the implementation of applications packages. Software for the second level has mainly been written in assembler language and is normally very system-dependent. Projects such as the SIM system(4) have attempted to make this level less system dependent by the use of a high-level language and the careful structuring of interfaces. With SIM, this level has been made easily transferable between different machines, provided the basic system executive provides certain standard facilities, such as block-oriented peripheral transfers.

At the third level, there has been nothing that has been demonstrated to be transferable between different machines. Apart from certain research efforts, e.g. the EMAS(36) development, this level is written in assembler and is very machine-dependent.

However, the work described in this report has produced transferable software at this level, albeit on small-scale systems, thus demonstrating the feasibility of making even this level machine-independent to a large extent. The fact that the work has been done on small-scale systems is an advantage in that the work involved has been of manageable proportions for one person.

There seems to be no reason why other components of basic system software should not also be written in a machine-independent way, and thereby made available for different machines.

These basic components would obviously have to be very well defined, with well-specified interfaces so that they could be properly incorporated into an overall system. An 'off-the shelf' software component would therefore have to be provided with a complete 'product description' in terms of how to use it and also in terms of its likely performance characteristics on particular machines.

One could foresee a situation where a range of different software modules was available for the same function. They would differ in respect of the way the facilities were implemented in order to lay emphasis on different things, e.g. efficiency in space or efficiency in speed, different scheduling algorithms, different techniques for disc space allocation, degree of fail-safe security in failure situations, and so on. All these various alternatives could be provided by different software packages.

Someone constructing a system then has to decide what particular features are important to him and choose the appropriate package. The task of system construction then becomes one of choosing a set of off-the-shelf components for the required functions, selecting those that exhibit the desired characteristics, which, of course, are likely to be different for different systems. These system components then have to be interfaced together according to the defined interfaces, and this may involve a certain amount of programming in order to ensure that interfaces match properly, e.g. interfaces to the actual peripherals.

One possible consequence of this method of building a system is that a set of software necessary to do a particular job can be chosen prior to choosing the hardware. The actual machine used can then be chosen on the basis of the most economic way of running the chosen software. Therefore, choice of hardware could become far more application-oriented. This differs from the situation which frequently pertains at present of choosing the hardware on the basis of apparently desirable features such as core speed or addressing properties, and then discovering how, if at all, particular applications can be run on it.

One further consequence of building a very modular system in this way is that, if the economics justifies it, it is possible to assign particular components to separate processors as a way of distributing the overall system load. Since each component has a well-defined interface, this should be easily accomplished without affecting the appearance of the component to the rest of the system. Then, major components such as control of communication network or a high-level filing system could be devolved to separate processors to relieve the load on the central part of the system.

14.11 Further Developments

Possible future work in the area of computer telecommunications suggests itself in relation to improving the communications techniques used to communicate with the large computers. The work described in this report was constrained to operate within the limited facilities presently supported by the large computers. The system developed therefore supported the simplest type of protocol for reasons of compatibility.

The indications are, however, that the large computer manufacturers are adopting a more flexible approach to communications through the use of programmable communications controllers. Also, there are moves to adopt a common standard which would permit compatibility at the level of block formats. It should be possible, therefore, to develop higher level protocol packages for small computers which overcome some of the limitations of the simple protocols but which are still compatible with different large computers, even if it involves some programming work on the large computers as well. There is certainly scope for improvement in this area.

In the area of system construction and development, there is scope for the development of further transferable system components e.g. multiprogramming monitors, disc filing systems, peripheral control systems, and the definition of system interfaces applicable in a wide range of applications and environments. This should help to promote the development of more modular systems with structures more easily understandable.

Further work is also needed in the development of suitable high-level languages for small computers. Although much work has been done in this area, there is certainly scope for further development in order to increase the facilities provided, such as those suggested in Chapter 13. Then, these developments need to be exploited and the compilers made more accessible by making them available on the commonly-used large computers. Also, compilers are needed which support the same facilities on a number of different small computers so that transferable systems are more easily implemented.

ACKNOWLEDGEMENTS

The work described was carried out in the Edinburgh Regional Computing Centre as part of the on-going communications network development. This work is under the general direction of Mr P.E. Williams, whose guidance is much appreciated.

Much of the practical work involved the active co-operation of the ERCC Engineering Support Group in handling the design and construction of communications hardware. Their involvement in the testing of various aspects of the overall communication system was also indispensable.

A more detailed list of items of work involving other people, for both hardware and software, is given in Appendix A.

The work done on the use of IMP as a vehicle for the communication system was dependent upon the willing co-operation of the respective compiler-writers, namely-

S.T. Hayes for PDP-8 IMP
N.H. Shelness for Modular One IMP
J.K. Yarwood for PDP-11 IMP.

The constructive criticism of Professor Michaelson during the production of the thesis was most valuable.

Thanks are due to Mrs J. Speed and Mrs L. Campbell for their patient endeavours in typing the thesis.

Finally, thanks are due to my wife, whose patience and encouragement were instrumental in the timely completion of the thesis.

REFERENCES

1. MILLS D.L., Topics in Computer Communication Systems. Concomp Technical Report 20 University of Michigan, 1969.

   MILLS D.L., The Data Concentrator. Concomp Technical Report 8 University of Michigan, 1968.

2. GOULD I.H., Communication between unrelated computers A Study of Software Design Problems. Institute of Computer Science, ICSP124, 1971.

3. YARWOOD J.K., Towards Machine-Independent Processors. Computer Bulletin, July 1970, Vol 14, No 7.

4. MILLARD G.E. and YARWOOD J.K., Bridging the Generation Gap Proceedingsof Conference - Software 72.

5. HAYES S.T., Real-Time Supervisor for Experiment Control by Computer. Ph.D. thesis, Dept. of Physics, University of Edinburgh (to be published).

6. JACKSON J.H., A Conversational System for the Graphical Specification of Markovian Queuing Networks. Concomp Technical Report 23 University of Michigan, 1969.

   JACKSON J.H. An Executive system for a DEC 339 Computer Display System. Concomp Technical Report 15 University of Michigan, 1968.

7. KRETZMER E.R., MOdern Techniques for Data Communication over Telephone Channels. Proceedings of IFIP Congress, Edinburgh, Aug. 1968 pp. D1-D5.

8. REES D.J. and WHITFIELD H., The IMP Language. Dept. of Computer Science, University of Edinburgh, 1970.

   BARRITT Mrs. M.M., BURNS J.G., MCKENDRICK A. and STEPHENS P.D., Reference Manual for the Edinburgh IMP Systems Edinburgh Regional Computing Centre, 1969.

9. MARTIN J., Chapter on 'Line Errors' in 'Telecommunications and the Computer' Prentice-Hall, 1969.

10. PETERSON W.W. and BROWN D.T., Cyclic Codes for Error Detection Proceedings IRE, January 1961.

11. Introduction to the IBM 3705 Communications Controller. IBM Form No. GA27-3051.

12. The ICL 7905 Communications Control System 7905 Introduction, Preliminary Edition TP 4364.

13. HIGGINSON P.L. and KIRSTEIN P.T., On the Computation of Cyclic Redundancy Checks by Program. Computer Journal, February 1973, Vol 16, No. 1.

14. CHISHOLM R.A.F., A Cyclic Redundancy Check Register. Edinburgh Regional Computing Centre, 1971.

15. BARRATT F.E.J., CHISHOLM R.A.F., FORDYCE J.G. and ROY A.N., Synchronous Communications Interface. Edinburgh Regional Computing Centre, 1971.

16. Specification of the 6310 Synchronous Communications Controller Data Dynamics Ltd., 1972

17. CCITT White Book Vol VIII, Recommendation V24

18. ICL 4100 NICE Executive Reference Manual.

19. ICL 4100 NEAT Assembler Reference Manual.

20. Modular One - 1.61 Communications System. Computer Technology Ltd., Ref. 361E.

21. Modular One E2 Executive. Computer Technology Ltd., Ref. 381/2/1U.

22. DP11-A Synchronous Line Interface Manual. Digital Equipment Corporation DEC-11-HDPB-D

23. PDP-11 Disk Operating System Monitor, Programmers Handbook Digital Equipment Corporation DEC-11-MWDA-D.

24. WOODWARD P.M., WETHERALL P.R., GORMAN B., Official Definition of CORAL 66. HMSO, 1970.

25. RICHARDS M., The BCPL Reference Manual. University of Cambridge Computing Laboratory, 1969.

RICHARDS M., A Description of the BCPL Compiler. University of Cambridge Computing Laboratory, 1971.

26. LADNER R.L. Verification of Transmission Algorithms. Ph.D. thesis, Dept. of Computing and Information Sciences Case Western Reserve University, 1970.

27. EISENBEIS J.L., Conventions for Digital Communication Link Desigg IBM Systems Journal 6, No. 4. 1967.

28. ISO Draft Recommendation on High-Level DAta Link Control (HDLC)
ISO/TC97/SC6 - 731 and 794

29. Post Office Engineering Department Specification TG2327A.

30. ASA Tutorial, Performance of Systems used for Data Transmission Transfer RAte of Information Bits CACM Vol. 8, No. 5 May 1965.

31. KUCERA J.J., Transfer Rate of Information Bits. Computer Design, June 1968.

32. General Information - Binary Synchronous Communications IBM Form No. GA27-3004.

33. Basic Mode Control Procedures for Data Communication Systems ISO Draft Recommendation No. 1745 ISO/TC97 (Secretariat-180) 278E, 1968.

34. Post Office Engineering Department Report TD5.3.2/RWB - CP5 'Datel 2400 Dial-up. Analysis of Results of Customer Field Trial' August 1972.

35. DAVIES J.I. and BARRY P., A Primer of HASP Multi-Leaving Edinburgh Regional Computer Centre, 1973.

36. WHITFIELD H., et al. The EMAS Operating System. Computer Journal Vol 16, No 4, November 1973.

Appendix A

Chronology of significant developments, indicating items of work involving other people.

Summary

The contribution of the author to the hardware work was in the detailed functional specification of the two communications controllers described in Chapter 7. The author had no involvement in the actual implementation of these specifications, except in relation to testing. Where work on the communications software involved other people, this was done under the close direction of the author. Unless specifically mentioned, the author acted only in an advisory capacity in relation to the development of user programs.

The following list of abbreviations for names will be used:-

ERCC Staff -

RJ - the author

FB - F.E.J. Barratt, Engineering Support Group (ESG)

RC - R.A.F. Chisholm, ESG

JF - J.G. Fordyce, ESG

AR - A.H. Roy, ESG

JA - J.W. Allan, Communications Software Group (CSG)

JD - J.I. Davies, CSG

KF - K. Farvis, CSG (Vacation student)

EM - E.R. Mansion, CSG

GB - J.G. Burns, ERCC/Physics

SH - S.T. Hayes, ERCC/Physics

Other organisations contributing:-

NC - Napier College

DD - Data Dynamics Ltd.,

CL - Culham Laboratory of UKAEA

RL - SRC Rutherford High Energy Laboratory

SM - Dept. of Social Medicine, Edinburgh University

GU - Glasgow University

HW - Heriot-Watt University.

The list of developments is given on the following pages.

YEAR	MONTH	COMPUTER	Brief Description of Work Done	Persons Involved
1969	APRIL	PDP-8	Design specification for ERCC Synchronous Controller	RJ,RC,JF
	JUNE	PDP-8	Design specification for communications software	RJ
	NOV	PDP-8	Prototype of ERCC SCI completed	RC,JF
	DEC	PDP-8	Communications Software package completed (written in IMP)	RJ
	DEC	PDP-8	User program to support calcomp plotter completed	GB
1970	JAN	PDP-8	ERCC PDP-8/L operational as RJE terminal with plotter	RJ,GB,RC,JF
	APRIL	PDP-8	Communication software converted to assembler to reduce size	RJ
	MAY	PDP-8	User program extended to support paper-tape input/output	GB
	JUNE	PDP-8	Design for re-engineered Version of ERCC SCI completed	RC,JF,AR,FB
	AUG	PDP-7	Communications software package translated for PDP-7,9,15	KF
	OCT	PDP-8	User program extended to support line printer	RJ
	NOV	PDP-8	First production version of ERCC SCI completed	ESG
	DEC	PDP-8	User program extended to support DEC-tape	GB,SH
1971	JAN	PDP-8	Physics PDP-8 operational with new hardware and DEC-tape software	RJ,GB,RC,JF
	APRIL	Mod 1	Simple test programs written to test communications hardware	RJ,JD
	APRIL	PDP-8	User program extended to handle card reader	RJ,GB
	APRIL	PDP-8	Social Medicine PDP-8 operational as RJE terminal with paper-tape and card-reader software	RJ,GB,RC,JF
	MAY	PDP-8	Computer Science PDP-8 operational as RJE terminal with paper-tape software	RJ,GB,RC,JF

YEAR	MONTH	COMPUTER	Brief Description of Work Done	Persons Involved
1971	JUNE	Mod 1	IMP version of communications package transferred from PDP-8	RJ
	AUG	ICL 4120	ERCC SCI interfaced to 4120 at Napier College	RC,JF
	AUG	ICL 4120	IMP version of communications package and user program translated to NEAT to run under NICE	JD,NC
	AUG	Mod 1	User program to support paper-tape input/output completed (written in IMP)	RJ
	AUG	PDP-8	Design specification completed for Data Dynamics SCI	RJ,DD
	SEPT	Mod 1	ERCC Modular One operational as RJE terminal	RJ
	SEPT	4120	Napier College 4120 operational as RJE terminal	JD,NC
	OCT	Mod 1	User program extended to handle card reader and line printer	RJ
	OCT	Mod 1	MRC Modular One operational as RJE terminal	RJ
	DEC	PDP-8	Prototype Data Dynamics SCI completed and tested on ERCC PDP-8/E	DD,RJ
1972	JAN	PDP-8	Assembler version of communications software modified for Data Dynamics SCI	RJ,JD
	JAN	PDP-11	Simple test programs written to test DP11 controller	JA
1972	FEB	PDP-8	User program modified to suit RHEL PDP-8/E DEC-tape requirements	RL
	MAR	PDP-8	Production Version of Data Dynamics SCI completed	DD
	MAR	PDP-8	RHEL PDP-8/E operational as RJE terminal with Data Dynamics SCI	RL
	MAR	PDP-11	IMP communications package and user program translated to PAL11 to run with IOX	JA
	MAY	PDP-11	Medical Faculty PDP-11/20 operational as RJE terminal with card reader and line printer	JA

YEAR	MONTH	COMPUTER	Brief Description of Work Done	Persons Involved
1972 (cont)	JUNE	PDP-11	User program modified to run under DOS and use disk	SM
	SEPT	Mod 1	IMP communications package modified to communicate in ISO with System 4-75	EM
1973	JAN	PDP-8	User program modified to support disk on Glasgow University Physiology Dept. PDP-8/I	GU
	MAR	PDP-8	Glasgow Physiology Dept. operational as RJE terminal with ERCC SCI	GU
	APRIL	4130	User program modified to support card reader and line printer under DES1	JD,HW
	APRIL	Mod 1	IMP RJE system modified to run under MISER and with different peripherals	CL
	MAY	Mod 1	User program extended to support CIL graph plotter	GU
	JULY	PDP-11	IMP RJE system transferred from Mod 1 to run on IOX	JA

Appendix B

Other available documents on particular implementations.

A number of other documents have been produced during the course of the work which describe certain aspects of the development in more detail in relation to specific computers. These documents are listed here for the benefit of people interested in thos particular computers.

JOHN R.B., The Interfacing of a PDP-8 computer to HASP as a BSC RJE terminal as a means of supporting a graph plotter and paper-tape input/output. Edinburgh Regional Computing Centre, 1971.

JOHN R.B., Programming Specification for PDP-8 BSC package Edinburgh Regional Computing Centre, 1972.

FARVIS K., BSC Communications Package for PDP-7,8,9 and 15 Edinburgh Regional Computing Centre, 1970.

GOODWIN D., RJE Input/output Package for the PDP-8 Edinburgh Regional Computing Centre, 1971.

DAVIES J.I., RJE System for the ICL 4100, Edinburgh Regional Computing Centre, 1971.

JOHN, R.B., Specification of IBM Communications Package for Modular One Edinburgh Regional Computing Centre, 1973

FORSYTH J.B., and PENFORL J., A Remote Job Entry Terminal to the Rutherford Laboratory IBM 360/195 Computer Rutherford Laboratory Report RHEL/R 261.

ABSTRACT OF THESIS

Name of Candidate Robin B. John

Address 5 McLean Place, Lasswade, Midlothian.

Degree Doctor of Philosophy *Date* November 1973

Title of Thesis The Design of Systems for Telecommunications between Small and Large Computers.

This thesis describes the development of a data communication system for small computers to enable them to link to large computers. The particular advantages and additional facilities made available to computer users through the use of such a link are described. A detailed description is given of the hardware and software components needed to achieve this link, together with the reasons for choosing the particular techniques employed. The discussions given highlight the problems involved in this type of operation. Some of these problems, such as lack of standardization, are short-term and will be overcome with the natural evolution of computer systems, while others are of a more fundamental nature related to the use of data transmission over long distances.

The system was designed to be applicable to a number of different small computers. This has resulted in a system which is easily transferable between machines, through the careful choice of interfaces to other components. This is seen as a step towards a more flexible and more modular method of system construction whereby complete software systems for arbitrary configurations can be put together using 'off-the-shelf' components already well-developed and tested. This contrasts with the present situation in which whole new systems are developed for a new computer, frequently duplicating systems already developed on other hardware. A detailed description of the factors involved in producing machine-independent, easily-transferable system components is given as a guide to other developments in this direction. It is felt that there is need for a better-engineered approach to the construction of software systems and it is hoped that the work described makes some contribution towards this end.

Use other side if necessary.