ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

PROCEEDINGS OF THE
1978 CERN SCHOOL OF COMPUTING

Jadwisin, Poland, 28 May-10 June 1978
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

Copyright CERN, Genève, 1978

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may not be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use. If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
ABSTRACT

Every other year the CERN School of Computing brings together young workers in physics and data processing. The main theme of the 1978 School "Large-scale data acquisition and processing in high-energy physics" is covered by lecture notes on data transfer in on-line systems, microcircuits, software for microcircuits and minicomputers, on-line filtering, design of experiments, optical computing, all relating to high-energy physics experiments. The proceedings also include notes on computer-oriented subjects such as communication systems, large computer systems and new architectures, performance optimization and computer-aided software development.
PREFACE

The Fifth CERN School of Computing took place in Jadwisin, north-east of Warsaw, from 28 May to 10 June 1978. A total of 81 students participated, 24 of them coming from Poland, 47 from CERN Member States, and 10 from Non-Member States of CERN. The majority of the students were engaged in particle physics or in the development of data-processing methods. The theme of the School was "large-scale data acquisition and processing in high-energy physics" and most lectures were devoted to different aspects of the on-line and off-line treatment of physics data. Other lectures treated large-scale computers and their performance. In addition, seven one-hour seminars were held, covering special topics or case histories.

The School was opened by the Under-Secretary of State in the Polish Ministry of Energy and Atomic Energy, Dr. Jan Felicki. It benefited from grants from the Physics Committee of the Polish Academy of Sciences and from the hospitality of the Regional Computing Centre CYFRONET of the Institute of Nuclear Research in Swierk.

On behalf of the Advisory Committee and all the participants, we express our gratitude to the Local Organizing Group, formed by staff members of CYFRONET, for the excellent local arrangements. In particular, we thank Mrs. J. Olszewska, who assisted in the running of the school and cheerfully helped in solving many small practical problems.

We wish to express our thanks to the CERN Scientific Conference Secretariat, and in particular to Mrs. Ingrid Barnett for her efficiency in the organization and the running of the School. We are also indebted to the CERN Text Processing Unit and Document Reproduction Services for their competent work in producing these Proceedings.

The Proceedings would not have seen the light of day without the efforts of the lecturers who have provided their camera-ready texts. It is a pleasure to thank them and the seminar speakers for their very good work.

Finally we wish to thank all those others who assisted in so many ways with the School: the members of the Advisory Committee and staff from CERN, INR and Drom Pracy Tworzej Inzynierow i Technikow Budownictwa in Jadwisin.

Last but not least, the enthusiasm and the active participation of the students contributed largely to making this School a very enjoyable and absorbing one. The 1978 School of Computing will be remembered by all participants for its friendly atmosphere, for a large part created by the outstanding social programme arranged by the Local Organizing Group.

R. Zelazny, Director of the School
C. Verkerk, Editor
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFACE</td>
<td>vii</td>
</tr>
<tr>
<td>ADVISORY COMMITTEE AND LOCAL ORGANIZING GROUP</td>
<td>viii</td>
</tr>
<tr>
<td>INTRODUCTORY LECTURE</td>
<td>1</td>
</tr>
<tr>
<td>R. Želazny</td>
<td></td>
</tr>
<tr>
<td>DATA TRANSFER IN ON-LINE SYSTEMS</td>
<td>8</td>
</tr>
<tr>
<td>V. Zacharov</td>
<td></td>
</tr>
<tr>
<td>MICRO-CIRCUITS FOR HIGH-ENERGY PHYSICS</td>
<td>25</td>
</tr>
<tr>
<td>P.F. Kuns</td>
<td></td>
</tr>
<tr>
<td>SOFTWARE FOR MICROCIRCUIT SYSTEMS</td>
<td>50</td>
</tr>
<tr>
<td>P.F. Kuns</td>
<td></td>
</tr>
<tr>
<td>ON-LINE FILTERING</td>
<td>65</td>
</tr>
<tr>
<td>C. Verkerk</td>
<td></td>
</tr>
<tr>
<td>MINICOMPUTER SOFTWARE FOR HIGH-ENERGY PHYSICS EXPERIMENTS</td>
<td>88</td>
</tr>
<tr>
<td>D.M. Sendall</td>
<td></td>
</tr>
<tr>
<td>COMMUNICATION SYSTEMS</td>
<td>105</td>
</tr>
<tr>
<td>D. Belenees and T. Noodt</td>
<td></td>
</tr>
<tr>
<td>DESIGN OF EXPERIMENTS</td>
<td>116</td>
</tr>
<tr>
<td>D. Drijard</td>
<td></td>
</tr>
<tr>
<td>LARGE COMPUTER SYSTEMS AND NEW ARCHITECTURES</td>
<td>124</td>
</tr>
<tr>
<td>T. Bloch</td>
<td></td>
</tr>
<tr>
<td>SYSTEM PERFORMANCE OPTIMIZATION</td>
<td>144</td>
</tr>
<tr>
<td>R.J. Bednars</td>
<td></td>
</tr>
<tr>
<td>COMPUTER-AIDED SOFTWARE DEVELOPMENT</td>
<td>189</td>
</tr>
<tr>
<td>D. Teichroew, E.A. Hershey III and Y. Yamamoto</td>
<td></td>
</tr>
<tr>
<td>INTRODUCTION TO OPTICAL COMPUTING APPLIED TO HIGH-ENERGY PHYSICS EXPERIMENTS</td>
<td>222</td>
</tr>
<tr>
<td>G. Metzger</td>
<td></td>
</tr>
<tr>
<td>LIST OF PARTICIPANTS</td>
<td>235</td>
</tr>
</tbody>
</table>
INTRODUCTORY LECTURE

R. Żelazny
Institute of Nuclear Research, Świerk, Poland

The world alters as we walk in it, so that the years of man's life measure not some small growth of rearrangements or moderation of what was learned in childhood, but a great upheaval.

Robert Oppenheimer

It is wise to attend CERN Computing Schools, particularly the 1978 School in Jadwisin.

Roman Żelazny

It is my great pleasure and privilege to welcome all of you by these introductory remarks. The 1978 CERN Computing School has been organized this time in collaboration with the Institute of Nuclear Research in Świerk, Poland. The Regional Computing Centre of this Institute, CYFRONET, celebrates this year the 5th anniversary of its foundation. To provide the effort to organize the CERN School was first of all motivated by the educational impact of the School, secondly by our traditional and close cooperation with CERN -- which we all wish to further and develop in a most friendly way -- but it was also to some extent a part of our jubilee activities. It seems to us that there is no better way to celebrate this anniversary than to take the initiative to perform such a pleasant duty. One should be aware that the organization of schools and conferences as well as other educational efforts in computer science is a statutory responsibility of the CYFRONET Centre.

I am very pleased to inform you officially that the authorities of CERN are represented here by Dr. E. Gabathuler, Leader of the Experimental Physics Division. I welcome him most heartily and on behalf of both host institutions I welcome all eminent guests of our School; the Undersecretary of State in the Ministry of Energetics and Atomic Energy Dr. Jan Felicki, all the prominent lecturers and invited seminar speakers and all the participants to the School. I wish all of you an enjoyable and profitable stay at Jadwisin and Warsaw. I do hope that this visit to Poland will be for all of you the beginning or continuation of long lasting friendly relations with our people and our country.

There is no doubt that the use of computers in every day practice has changed the image and the spirit of physics, particularly of high-energy physics. Conversely, the computational methods and the development of computers was affected in an intimately deep way by the requirements and demands of physics. (At this moment I leave aside another important factor as solid-state physics which also has contributed to the architecture and performance of today's computers but which I consider as marginal in the scope of our considerations.) The two introductory lectures are intended to display this interaction between physics and computing, to elucidate the motivation of the 1978 CERN Computing School and to discuss the programme and content of the lectures from the point of view of those mutual transformations, which physics, computers and computing undergo while influencing each other.

My primary objective is to show how the implementation of specific tasks in physics, which can be described generally as large volume experimental data processing and interpretation -- mostly in high-energy physics -- can be performed using contemporary and possibly future possibilities of computer techniques and how the programme of this school has been organized to show you the basic ideas of this process. I will be very happy indeed if from my talk you will be able to deduce that our school is intended to help you to grasp how with different hardware and software elements one can construct a system able to perform our basic task and related problems in a reasonable way. To do this one must find a relationship between the set of lectures and the formulated objective. Let us build such a relationship from the point of view of the process which we are going to implement and which we consider as our basic activity.

Let us imagine we have a large scale high-energy physics (HEP) experiment. What does it mean from our point of view? Let it be an experiment in a beam. Some of these experiments are described in more detail in seminar lectures selected for our school. You will listen to details of hypernuclear gamma spectroscopy experiments, in which during the burst of CERN proton synchrotron (500 msec) K⁻-mesons are produced and irradiate the target of Li-nuclei.
You will get analogous information from two other seminars describing particular physics cases: one at SLAC, the other at Fermilab. You will listen to lectures describing typical experiments with a beam of particles incident on a fixed target in contrast to the experiments performed with storage rings, where you have two beams or bunches of particles colliding with each other. You will hear of peculiarities due to such different arrangements.

Of course, I will not go into details now -- they are at the moment irrelevant for us. The common factor of all beam experiments is that during the consecutive bursts of the beam or the collisions of two beams of particles in a storage ring a very large amount of particles, of the order $10^6 \cdots 10^{12}$, and sometimes even more, hits the target particles (collides) during one second. The duration of the pulses may change from a small fraction of a second to a few seconds. The interaction between two colliding beams in a storage ring is practically continuous, without a microstructure.

As the result of such encounters the interesting events take place, which we observe and of which we register the data. From these data we draw physics conclusions about the forces or the existence of particles, resonances, kinds of decay, etc. How much information is provided by one event in HEP? Usually, at sufficiently high energy multiproduction of particles takes place. Every case is characterized by approximately $10^4$ particles. Those particles are registered by a sophisticated system of detectors, surrounding the volume, in which the interaction of colliding particles occurs. We would wish to cover the whole spherical angle $4\pi$ and register or monitor possibly all ingoing and outgoing particles. There is a big variety of contemporary, more or less sophisticated detectors. Let me remind you of detectors as wire spark chambers, drift chambers, multiwire proportional chambers, anticoincidence counters, shower counters, gas Čerenkov counters, threshold Čerenkov detectors, scintillation proportional counters and many others. As all of them have their associated electronic equipment, converters and so on, this is the place where the CAMAC modules enter our picture. Individual experiments using electronic detectors consist of 100 and sometimes even more of such devices. Depending on the kind of detector one needs of the order of $10^2$ bits (10 words $\times$ 10 bits) per detector to register reasonably accurate information. Thus for one event one needs $10^2 \times 10^2 = 10^4$ bits of data to be registered. Having $10^8 \cdots 10^{12}$ interactions per second one obtains large amounts of data which are not of the kind one is interested in at this specific experiment. If we try to register all available data the volume of those data exceeds the recording possibilities of any one of the existing computing facilities. This is the reason why we -- using microcircuits and microprocessors -- trigger the action of the experimental facilities only on those events which are of the desired character. Series of lectures on hardware and software for microcircuits, microprocessors and microprogramming let you get acquainted with existing commercial possibilities and with the technology used in achieving specific goals in triggering HEP experiments. One should realize at this stage that microcircuits and microprocessors are used as part of experimental facilities not only to trigger on the desired events but also to satisfy specific processing needs. Through the nowadays so fashionable and practical technique of "distributed processing" or "distributed intelligence" we delegate to specific detectors, to specific subassemblies of the experimental set-up and to individual microprocessors the task of tackling specific processing needs connected with their functions and with their mutual communication.

It is worth while to notice that today's microprocessors are in fact becoming more and more full microcomputers (minicomputers). You will listen to the review of progress in this field in the last few years. I hope that it will be clear enough that a knowledge of how to use these elements is necessary not only to pick out from the "mess" of a huge number of different events those which pass certain strict physics selection criteria -- mostly tightly connected with the geometrical and physical set-up of the experiment -- but also to reduce in an intelligent way the amount of registered data and their distribution to the appropriate recording devices.

The experiment is going on. The sought for events trigger the registration of the experimental data by a minicomputer system equipped with disks and magnetic tape drives. This minicomputer system may be of several sizes, tailored to our needs, depending on the size of the experiment and on the amount of data to be recorded.

Let me quote here other figures which might be useful in evaluating the volume of data. One reel of magnetic tape contains of the order of a few $\times 10^8$ bits of data. For one experiment it is considered that from 10 up to 500 tapes may be generated, in average 200 tapes. On a PDP 11/45 one needs from
1 to 5 hours/tape to analyse the data. An average experiment needs then about 500 h on a PDP 11/45 for data analysis.

As you probably know people quarrel bitterly on the issue of the definition of a minicomputer. It is considered that a minicomputer is a "beast" which is cheap, easily expandable due to its architecture based on the bus concept and -- what is even more important -- which is suited to solve your individual problems. An extremely important role in fitting the minicomputer system to specific needs is played by user microprogramming. We shall listen to the lectures on these topics. You will also learn that in some cases one experiment is backed up by a few dedicated mini- and micro-computer systems.

Is everything going correct? To answer this question one must consider the following subquestions: are the detectors placed in an optimal way around the place where the collisions take place; are they designed in a proper way to enable the reconstruction of the trajectories of all incident and outgoing particles; is the number of events adapted to the performance of the detectors and their electronic back-up; does the trigger act in a proper way without introducing false signals; is the minicomputer system fast enough to record all data which we wish to store as the result of the data gathering phase in running the experiment? There are still more relevant technical questions which should be answered and -- what is important -- checked before and during the experiment run.

All these questions are answered during the process of simulation and -- I would call it -- tuning of the experiment set-up. The philosophy and technicalities involved will be given in a lecture here at the school. What I would like to stress is the following:

1. Simulation of experiments is at present performed with the help of relatively large modular packages of programs on relatively large computers (main frames).
2. It helps to plan and optimize the geometrical, kinematical and electronic elements of the, at present, very complicated experimental set-ups.
3. It is very important from the point of view of the discussion of errors and reliability of the final physics results.

To say it half jokingly, a large part of the experiment is done on the computer without any real particle. "Unfortunately" or "fortunately" -- depending on the point of view -- if we wish to get new information about physics laws we must change the randomly generated quantities into real ones from events using the accelerators or cosmic rays.

Let us assume that we have done everything possible to secure the proper set-up of the experiment. On a large computer we have run specific programme modules, for example GEANT at CERN's 7600 or another from the wealth of analogous ones and before starting now to assemble any experimental facilities we were able to design them as well as possible. Afterwards we order specific detectors, CAMAC modules, other equipment to have everything ready when we obtain beam from our accelerator.

We have got the place, we have mounted everything, checked everything possible under "beam off" condition and then comes the most crucial time -- we have got the beam for a specified number of hours. Usually less than we have asked for and expected. How to use this time efficiently and be sure that we perform the experiment properly and get surely the data which can be processed afterwards in a relatively less tense atmosphere?

Extremely important is to know after a few hours of the run whether everything goes as we have planned it. If not, we must take corrective action. Of course corrections will not be applied to the accelerator, although even such cases could not be completely excluded. So what do you do? "Today's" experiments are conducted in the following way. Interesting events which trigger our apparatus enter the minicomputer system dedicated to our experiment. There is quite a large amount of data. There is then a necessity to check them or to filter them. The filtering process is mostly based on checking the consistency of data with global conservation laws, and other relations which obviously must be true. Only those data which satisfy the checks pass through the filter and can be recorded. Now comes the problem of the volume of data, time of their arrival, and the duration of the break between two consecutive bursts of particles from the detectors. All those elements play an important role in the design and implementation of filtering through the microprogrammed elements of our system. In the majority of cases it is being done by hardware arrangements. In special circumstances you may solve the problem implementing purely software methods. You will listen to all these concepts and examples of implementation in one of our lectures.
When the amount and rate of data go up we speed up the system, buffer the data on disks, look after huge mass memories with fast access. We subdivide the data to be recorded into subfiles and dedicate to them separate recording systems. Another solution might be to be connected to a large computing system via very fast telecommunication lines. Large systems have much faster mass memories, much more tape drives, and are better able to perform efficiently filtering of your data. This filtering might become more sophisticated, more demanding. But you are not any more the only user of the system. The system demands from you observation of certain rules, demands patience and queueing, queueing, ... This way we have encountered a new, interesting quality, a new situation. This situation gives you the chance:

1. to run an experiment in a completely different place from the place of your computing back-up;
2. to split your team into two subteams: one working with the beam and the facilities at the beam, the second one occupied with the computer and the recorded data;
3. to check whether the experiment is going the right and expected way -- by performing not only filtering but also preliminary processing for checking and to correct -- if necessary -- the course of the experiment.

This is networking, with all its implications and possibilities, and questions. What type of networking to use? One "monstercomputer" connected to many "intelligent terminals"? The "intelligent terminal" being the minicomputer system dedicated to our experiment.

Or, we build up back-up minicomputer systems to such a size and complexity, that they be able to support the experiments. Opinions differ. The decision depends on cost effectiveness, volume of experimental data and size of the process of their interpretation and manipulation. In high-energy physics usually in the last resort people like and need to have "number crunching monsters".

Let me mention here one example of existing networks: Integrated Computer Network (ICN) at Los Alamos Scientific Laboratory. There are a large number of "number crunching monsters" in this network, three "Cray-1" and quite a few CDC 7600's. Another example of a network created specifically to serve the purposes of HEP experiments at SPS is the CERNET system (CERN Data Communications Network) allowing to connect minicomputer systems dedicated to individual experiments to an IBM 370/168 and a CDC 7600 computer via fast links.

Please remember that the volume of data per experiment is of the order of a few hundred tapes, let us say 200. If you have 30 experiments going on during a year you fill 6000 to 15000 tapes per year. The time necessary to analyse one tape is of the order of a few hours on a PDP 11/45. The CDC 6600 can analyse about 10^7 bits/sec. This way you may roughly estimate the required size and speed of your computer system.

To achieve the network goals one must be cognizant of the intricacies of telecommunications, of the notions of protocols, communication equipment, interfacing, error control, present possibilities and limitations, future outlooks, performance of the elements of your system.

There is a little doubt that finally we all will be caught in a "monsternet" created by giant computers and giant accelerators. The only question to be answered is where the "homo- spiders" shall sit in this web system: at the accelerators from where they send everything to computers which will act automatically, remotely controlled, or conversely at the computers, controlling everything remotely from there and setting up the accelerator exploitation and the experiments as part of a computer controlled activity?

Well, this is what should be understood by "distributed intelligence". But this way we have come to the computer monsters. In a particular lecture we will learn about their architecture, present trends in building them larger and faster. Particular attention to the idea of "parallelism" as the way of achieving very high computing speed will be given, with a mention of the difficulties encountered. We shall be interested in the components of such large systems and particularly in mass memories. We shall look upon these giants as elements of the networks and consider their software possibilities and trends of developments.

Concentration of computing power in the nodes of the computing network requires careful tuning of all components, hardware- and software-wise, to get the high effectiveness of such a system. Performance problems, the measurement, objectivization and tuning of performance is the subject of another lecture which will allow us to look at the giants and networks from a different but very important point of view.
Looking for new ways of solving computing problems we should not omit one of them which may completely change the concepts and methods of our work. I think of optical computing. As you know one revolution in physics was connected with light. Who knows what light is able to do in computers and through computers again in physics. About holographic memories we have heard something; the famous Illiac "ribbons" do exist. But who knows about parallel transformation properties of a laser beam, allowing to do the "tse computing". Picture-o-matic computing in an almost Chinese way. This seems a kind of political heresy but I am afraid in about 5-7 years we might all compute "à la chinoise".

Let us nevertheless return to our data. Along the lines I presented before those data have arrived at our large computer. This has been achieved with the help of data communications lines or — in simple cases — trivially, by carrying the data by a messenger or the interested physicist himself. We can start data processing. In which way? How to prepare a package of programs for such a computer? You know about different schools of thought and ways of organizing such a project. Big volumes have been written on this problem and many interesting computing schools have treated it. We shall not recall many of the existing approaches. We simply shall focus our attention on methods which might change programming methods to a large extent. Let me mention one of the seminars devoted to the special automatized language enabling the user to solve systems of differential equations in an easy and effective way. Another approach is even more modern and promising: why not use computers to develop large, complicated systems of programs. In other fields such an effort has been undertaken. Can they be transplanted to physics? Could we learn something from this approach? This would be a very important new quality in our activity in the future. It is beyond comprehension that such an idea has not been proposed and implemented earlier. Interesting information on this inspiring new approach to our old problems will be presented in one of our lectures.

I have opened this School and this lecture by mentioning the Regional Computing Centre CYFRONET. It seems appropriate to conclude my talk with some remarks concerning its present status and lines of development. The Computing Centre delivers computing power to the Institute of Nuclear Research and other Academic and Research Institutions of the Warsaw Region, such as Warsaw University, Warsaw Polytechnic University, the Research Institutes of the Polish Academy of Sciences, etc. This basic goal is achieved through the activity of the CYFRONET NETWORK, with 16 remote terminals (6 Low Speed Batch Terminals, 4 intelligent terminals, each equipped with a PDP 11/45 and 6 TTYs). The configuration of the computer system, a CDC CYBER 73, with communication lines and terminals is shown in Figs. 1 and 2. On the photos a short excursion to the Centre is presented to give you an impression of how it looks like. At this moment 40 MSS (million system seconds) are being delivered to the users per year. We wish to double it in 1980 and arrange for 20–25 intelligent terminals. TTYs are to be connected only via intelligent terminals.

In this way we have completed the presentation of the basic concepts of the programme of the School in the framework adopted from the very beginning as the guide line of our activity. It seems to me that the intimate relationship which does exist between physics and computing in the field of large volume experimental data processing and interpretation has been demonstrated and will be demonstrated every single day of our school.
**Fig. 1** Configuration of CYBER-73 at Institute of Nuclear Research

**Fig. 2** CYFRONET terminal system
DATA TRANSFER IN ON-LINE SYSTEMS
V. Zacharov
University of London Computer Centre

Abstract
The problem of transfer of data in both directions between experimental equipment and process systems on the one hand, and hardware processors on the other, is an important one. This fundamental question is discussed in the context of contemporary practice, where the principal processing element is the minicomputer. Although several interface conventions will be considered, practice is dominated by the CAMAC system, and the main emphasis will be to review recent developments in that system, particularly in the area of distributed configurations. The impact of new microcircuit technology on the way in which data transfers are performed is only beginning. The present discussion will try to assess this impact and to identify the main changes that are expected to occur.

1. Introduction
In the overwhelming number of cases, the digital data generated by instruments, detectors and transducers in experimental science, in engineering and in process-control applications, is, in raw form, useless. It is usually of considerable volume, it is generated at rates much too high for direct human absorption and, inevitably, some of it will be contaminated or destroyed by some form of noise or spurious data. In short, such data is neither useful information for those human beings involved nor is it of the right form to be used as input to some kind of automatic control system.

Nowhere is this situation more evident than in experimental particle physics where, for example, many modern spectrometer experiments may generate data at rates in excess of 10^6 bits/sec over running periods of months, but where the final result can usually be summarized, if not actually by one number, at any rate by a very small number of graphs or histograms.

Thus, before data can be of any value in all these cases, it has to be processed in some way or other. Usually the processing will result in a reduction in volume of source data by several orders of magnitude, and this can only be achieved by an automatic computational process; but in other cases the problem may only be one or sorting or formatting, in which case some kind of computational process will again be required. Even if all that is required is temporary buffering or more permanent storage of data, there will still arise the question of some sort of data transfer and processing.

So we can see that, in the majority of real cases, the data generated by various detectors and other data sources has to be transferred into some other device or system, and this device must be capable of digital processing or computation. But this problem is further increased by two other realities: first, the data is not usually generated in the right place and so will have to be transferred in any case; second, it is usually generated on time scales that require the data to be removed very quickly indeed, either because the data source has to be restored to a form appropriate to record more data, or because some result (or feedback) is required from the data already generated. This second consideration is, in fact, one which imposes one of the most stringent demands on the data transfer system; the data transfer problem becomes indeed a "real-time" one.

Corresponding to the all-pervasive real-time data acquisition problem just described, there is also the problem of data output. Here also, even in the event that the data is required for visual inspection by some human being, the requirements are often very demanding. The human being not only may wish to see a presentation in graphical form, often dynamic and with some dimension (like colour) artificially introduced, but he will certainly require the information on a quite different time scale from that of input data generation. But, for process-control applications, the output data required for transducers and forth will generally be of totally different form from the input data. And because output data is frequently required, both by humans and by control equipment, on time scales dictated by the need to ensure that the course of the experiment or process is a correct one, the output data-transfer requirement becomes like that for data input, a real-time one.

In summary, for so many problems associated with data generation and output, the first problem that arises is that of data transfer. And because of the nature of the data and the associated process, this problem is generally one of real-time data transfer. The purpose of these talks will be to examine this problem and to see what success there has been in finding relevant solutions. Since the problem is of such universality and pervasiveness, the question of finding conventional or standard solutions becomes of the greatest importance, and particular emphasis will be placed here on this aspect.

2. The System Problem
The first question that we need to examine is between what device or system component the data should be transferred. There are very many different kinds of data source and also a wide range of sinks or acceptors. Data can be processed by very many different processors or computers. So an important question is whether there are many different types of data-transfer problem or not.

Actually the number of different categories of receptors of data is not very great, so let us examine each of them in turn to see if there are any common features in solving the problems of data transfer to or from each category. As we shall see, we can reduce the complexity of the problem considerably.
2.1 Data Links

For the sake of completeness in our survey of data transfer we need to include data links, because one thing data is just to transfer it somewhere else. Of course, such links do not in themselves help us much except in removing data from one physical location to another, more convenient, one and sometimes in providing some short-term buffer storage. But, even if one could readily transfer data from detectors or other sources into data links, the problem is only translated to that of data transfer at the other end of the link.

The range of data links is enormous, and there is certainly no difficulty in transferring data along appropriate links at the highest rates of data generation encountered. They have been extensively described elsewhere and are only mentioned here to emphasize one point, namely that the use of links for data transfer introduces additional problems. It is not just that noise and digital errors are introduced by data links, but that their use inevitably leads to problems of increased complexity both in hardware and in logic. Even in the simplest case, the use of digital data links will introduce the necessity of an error-recovery strategy and a communications protocol, no matter how primitive these both may be. In more complex situations, there will also be the problem of commutation and multiplexing.

Thus the existence of data links does not really help us solve the fundamental problem of data transfer, namely, that we have to get the data processed. Indeed, data links introduce additional problems which further enhance the need for a data-processing capability in the system.

2.2 Storage

There are two categories of storage media that we need consider: short-term buffer storage and more permanent mass-storage. In all practical situations the former always exists, even if only in the form of cables. But nowadays it is rare indeed to find any system generating digital data without some form of active storage, and the main purpose will be for so-called "de-randomizing" and for obtaining from short bursts of data at high peak rates longer data blocks at more modest transfer rates.

So, like data links, buffer storage, however useful, does not really solve the basic requirement of data transfer which is to move data into or out of an "intelligent" processing device. And also, like data links, the use of buffer storage introduces additional requirements to process the data. Modern buffer storage is nearly always available in some conventional form, into which data can only be inserted provided it is properly formatted both physically and logically. And so the raw data from detectors and other data storage has often to be processed in some way even to be inserted into buffer storage. It is very uncommon indeed in modern systems for buffer storage to be unassociated with some form of processing element, and the most usual arrangement by far is for the buffer storage actually to be the memory of a minicomputer. Once again, the problem which remains to be solved is how to transfer data into intelligent devices.

In the case of mass storage the argument is different, since any computer system can have such storage attached, for example, magnetic tape units. So, if data can be transferred on-line, say to magnetic tape, then the tape can be subsequently transferred off-line onto a processing system. But the problem of data transfer onto magnetic tape, or for that matter onto any other mass-storage device or medium, remains. Here we are faced with two problems: first, that of formatting the data to the form required by the storage medium; second, that of matching the data rates. For both these problems we require together both a buffer and a processing device, and by far the best way of solving these associated problems is to use a minicomputer. Once again we see that the problem of data transfer, even in the case of intermediate storage, reduces to that of transfer to a processing device, usually a minicomputer.

2.3 Large Computer Systems

If we set aside for the time being the pre-processing of experimental data, associated for example with event selection in elementary particle interactions, the overwhelming bulk of data processing tasks could be handled in principle at least by large computer systems. Such systems have adequate processing power, they have a large range of peripheral devices, such as mass storage, and they are quite capable of the formatting and display required for visual presentation to human beings. So we should ask why cannot the data transfer problem be solved by moving the data into and out of large computer systems.

This was indeed one of the early approaches to on-line data acquisition and control in a number of large centres engaged in both high-energy and nuclear-structure physics, where experimental equipment was coupled to large (at that time) computer systems by means of special hardware directly coupled to an I/O channel. Later, special hardware was partly replaced by hard-wired couplers or data adapters provided by the manufacturers of the large computer mainframes, a typical example of which was the IBM 2701. It was quickly realized by many, however, that this approach was not the best, for a whole range of different reasons.

The operating systems of large computer systems are not designed to handle real-time problems of the kind encountered in experimental science. On the contrary, such systems are intended to utilise system resources (particularly central processors) very efficiently, or to give a rapid response to multi-access terminals operating at human speeds, or sometimes both. In addition the architecture of large systems was developed so that input/output was mainly along channels designed to match the requirements of conventional peripherals such as tape and disk drives. All these properties of large systems taken together made them well suited for transfers of large blocks of data in a highly scheduled and ordered manner, that is to say, kiloword blocks transferred at 1-10 millisecond intervals, but very badly matched to real-time requirements of data transfers in small blocks at microsecond time scales. Moreover, large computer systems are almost incapable of operating in a demand-handling regime; at the very best, the processing of external interrupts on a large system only results in a tremendous degradation of performance.

There are other aspects of large computer systems we should take into account, such as the fact that they are inefficiently used for the short-word integer data processing characteristic of data transfer tasks; but, in summary, they are very ill-suited to the environment we are discussing.

- 9 -
The way out of the problem of transferring data to and from large systems is to introduce an intermediate computer system between the data sink or source and the large computer, to make an impedance match, so to say. Such an intermediate computer system is termed a "front-end" computer, and invariably comprises one or more minicomputers. The functions of the front-end system can vary, but usually include buffering, multiplexing, formatting and sometimes even a certain amount of pre-processing, mainly involving integer arithmetic. The front-end processor will also handle all those computer tasks associated with communications and with servicing interrupts.

Thus, yet again, we see that the problem of data transfer between large computer systems and experimental or process-control equipment reduces to that of transferring the data to or from a front-end computer; that is to say, a minicomputer. In actual practice there may well be more than one minicomputer, either to separate certain functions or to provide a back-up capability for enhancing reliability, but this is not of importance for the present discussion. Also we need not be concerned here with the nature of the connection between any large computer and its front-end, since this is a problem that is faced very seldom, usually only once in the life of the large system, which is typically five years or even more.

2.4 Special Purpose Processors

In a growing number of cases, particularly in experiments in high-energy physics, there is an intermediate stage in the chain of data-transfer system, from data source to data acquisition buffer and processor. This intermediate stage is concerned with enhancing the quality of the data by pre-processing or filtering; the aim is to reduce the data as much as possible in quantity and in average transfer rate, and eliminating as much spurious data as possible. In practice the intermediate pre-processor is a special-purpose system, generally made (and indeed designed) just for the requirements of a particular data source or experiment.

A discussion of pre-processors is outside the scope of the present talks, and is extensively discussed by others, but the data-transfer problem is not greatly affected by the existence of such systems. Data transfer into pre-processors is a problem that in general has to be solved in a way that differs for each new system. It is very rare indeed for special-purpose processors to be capable of complete reduction of data, and so the problem still arises in any case of further data transfer from the pre-processors to the next component in the data-transfer chain, nearly always a minicomputer.

For the purposes of the present discussion, we shall regard special-purpose processors as being a part of an experimental system. In high-energy physics, for example, we shall think of pre-processors as part of the event triggering system, yielding a more refined "signature" for wanted events than would otherwise be possible. So our problem here is, once again, how to transfer data to a minicomputer.

In one particular respect, the discussion of pre-processors is different from all the other components considered so far, namely that pre-processing can represent one aspect of the introduction of "intelligence" into the data-transfer chain between source and main data-acquisition processor. This is something that we shall come to briefly when we consider the present status of modular systems and the question of introducing "distributed intelligence".

2.5 Microprocessors

The whole issue of microprocessors is extensively discussed elsewhere [7], but is one which is raised here in the context of data transfer to ask if the question of data transfer to minicomputers is any different from that of transfer to microprocessors. The question is a complex one, which we will mention again later, but at this stage we can say the following. Firstly, if the microprocessor is just one component of a system whose architecture serves the function of a minicomputer, then the problem of data transfer here is identical to that of a minicomputer. The fact that certain manufacturers (such as DEC) have chosen different I/O structures in the microprocessors they intend to replace their own minicomputers is an irrelevancy, since it does not change the qualitative nature of the data-transfer problem.

Secondly, if one or more microprocessors are components of a special-purpose system, such as a pre-processor, then the problem of data transfer into such systems is, as stated before, a special one, outside the present discussion. The problem of transfer from the special system to any subsequent minicomputer remains however.

But one aspect of microprocessors still remains, namely, the case where a complex microprocessor system may be all that is required for the complete reduction of data from some experiment or instrument. In that case, the problem of data transfer into a system synthesized of microcircuits is raised. Unfortunately, the disparity of different microcircuit components is so great, and the rate of evolution of these components so rapid, that it is impossible at this stage to make any general recommendations. There are not even any de facto industry I/O standards for microcircuits, far less any interconnection conventions, and we must wait rather longer for the emergence of general solutions.

3. Minicomputer Data Transfers

We have seen from the foregoing discussion that the most important problem introduced by the need for data transfer is that of getting data in and out of minicomputers, whether it be for the purposes of control or because of space requirements. This has been the situation in most experimental science (and certainly for high-energy physics) over the last ten years or so, and it is one which is likely to continue for a considerable number of years more.

Thus the first question to ask concerns the input-output (I/O) structure of ordinary minicomputers in widespread use, and to see to what extent this structure is appropriate for data transfer. Necessarily, the environment we are considering is a real-time one with on-line data transfer. To answer our question we need to examine the principal components of the I/O
structure, which are:

1) Program-controlled I/O
2) Direct memory access
3) Demand handling
4) I/O commands and software

We shall only review these briefly, for a more comprehensive survey will be treated elsewhere, and in particular we shall be concerned here only with the main options most generally met in typical and popular minicomputers.

3.1 Program-controlled I/O

This is the facility whereby data may be transferred between certain registers of a minicomputer processing unit (CPU) and an external device by means of execution of certain instructions in the minicomputer command repertoire. Nearly always the external device is not coupled directly to the registers, but couples by means of an intermediate unit, generally termed an interface unit or device controller (Fig. 1). The registers are usually special ones used only for I/O, but sometimes they are general-purpose CPU accumulators. In certain cases the special registers are chosen to be selected (and fixed) memory locations.

![Fig.1 Program-controlled I/O configuration](image)

Data transfers under program control occur when certain commands are encountered during a running program. Generally more than one instruction is necessary in order to transfer a single data word, several different commands being necessary usually for the purposes of device status checking, actual word transfer and waiting for device readiness. Even with very modern minicomputer architecture, program-controlled I/O is a technique that is generally slow, particularly for transfers of blocks of data words, since such transfers usually require even further instructions for counting and testing.

3.2 Direct Memory Access (DMA)

To overcome the limitations of program-controlled I/O for block transfers, many different techniques and configurations have been used, but by far the most usual is to transfer words to or from memory directly, with no program execution except possibly for setting up or termination of the transfer (Fig. 2).

![Fig.2 Direct Memory Access (DMA)](image)

There are two essential aspects of most DMA systems. The first is that the operations that need to occur each time a word is transferred, such as counting the number of words, sequentially incrementing the memory address, and checking whether the required number of words in the block has already been transferred, are performed other than by CPU instructions and usually by hardware. The second is that transfers can occur at any time in synchronism with a memory cycle, and so have only to wait for the end of execution of a current CPU or memory reference instruction. This latter facility, known as "cycle stealing" or "data-break", implies that the CPU is generally only temporarily halted for a single memory cycle for each transfer of a word along DMA channel. Indeed, in certain minicomputers, the memory may be partitioned into segments with several access paths. In this case DMA transfers may actually occur into one memory segment, while the CPU can continue execution without interruption if it only requires access to other segments.

Because of DMA hardware the data block transfer speed can occur at a rate limited only by the memory cycle time, typically nowadays 300-500 nanoseconds per word. And because of the data-break mode, the time necessary to initiate block transfer can be very short indeed. In DMA transfers, data words in a block are stored in contiguous memory locations, so the steps necessary to initiate such transfers are to set up in appropriate registers a word count (data block length) and starting address in memory of data-block storage. There will also generally be the need to signal to the supervisor (hardware or software) that a transfer is about to commence (Fig. 3).

Clearly, at the end of any block transfer, there will usually need to be an appropriate termination sequence of action, in order to restore the relevant minicomputer states back to those which obtained before block transfer. In many cases, this is done by generating an interrupt to the CPU with an appropriate transfer to the supervisory program to signal that DMA has been completed.

It should be emphasized that Fig. 3 represents only one of many different variations of DMA procedure. For example, when a DMA sequence is proceeding, but no data is available, instead of initiation of a recovery sequence as in Fig. 3, there might simply be a loop to wait for more data when
ready. It is important to note, however, that in nearly all DMA variants there is no effect on the status of any running program: the current address (CA) referred to in Fig. 3 is the address of memory and has nothing to do with any program counter or program address register.

![DMA flow-chart](image)

Fig. 3 DMA flow-chart

3.3 Universal Bus

The path between memory and CPU in a minicomputer is generally termed a Bus, and it comprises data, address and certain control lines. On the other hand, both for program-controlled I/O and DMA, the intercoupling paths are not so simple; they are called channels. In this case, there is a fairly well-defined difference between the two in the way in which data is transferred. For the memory bus, transfers are extremely straightforward, being achieved in a single cycle simply by the presentation of the relevant memory address and a definition of the mode (read, write, data width, etc.). Data transfers along I/O channels require, however, a definite sequence of distinct actions or steps (Fig. 3 shows one example of such a sequence); indeed, what is required here is a "protocol", albeit a rather elementary one.

In certain minicomputer architecture the memory bus and I/O channels are combined so that all elements or modules of the whole system communicate and transfer data along a single common highway or bus (Fig. 4). The best known of such examples is the DEC Unibus.

![Universal minicomputer bus](image)

Fig. 4 Universal minicomputer bus

Such common buses have several advantages over the other forms of architecture we have considered. For example, the addresses of memory locations, CPU registers and also registers in peripheral and external devices can be regarded as homogeneous, filling together the minicomputer address space. Here, contents of all locations in the address space can then be used as operands in any of the CPU instructions, including peripheral register contents. In common highways, the distinction between program control and DMA vanishes, since both become merely options of the general transfers along the bus. Also it becomes possible to transfer data between two devices directly without any intermediate step involving the CPU. There is, however, one important disadvantage generally, namely, that only one single transfer can occur at any given cycle along a common bus.

Since all data, address and control lines are shared in common by all units coupled to the common bus, it is clear there cannot be more than one paid of addresses present or more than a single data word at any one time. This problem has been tackled in many different ways, for example, by only allowing a single unit coupled to the bus (perhaps the CPU) to supervise all transfers; in other words, there will only be a single bus "master". More commonly however, the problem is resolved by having a special unit (or special software) to select which pair of modules coupled to the bus is allowed to be in communication at any given instant. Such a unit is called a bus-arbitration unit as shown in Fig. 4, but it must be remembered that the problem is a complex one, which requires for its resolution a knowledge of the states of all devices and units coupled to the bus as well as the ability to respond to every control condition that might arise. As we shall see, this problem is related to that of demand handling, particularly in an environment of inter-rupts with differing priorities, and it cannot generally be resolved by hardware alone.

So we see that, whatever the disadvantage, at least the common bus has the important advantage of more or less common data-transfer protocol for all external (and, for that matter, internal) device controllers coupled, irrespective of their speed or application. And in addition, as seen in Fig. 4, the interface (I) for coupling to the bus is the same for all units. However, before we can say whether or not the common bus of a minicomputer meets all
the requirements imposed by data transfers, we must examine the remaining aspects of demand handling and of software.

3.4 Demand Handling

Under program control and possibly even for DMA it would be possible to accomplish data transfer under the complete supervision of the CPU. However, this would certainly necessitate the frequent polling of external devices to see whether they required any attention, but also to discover when any particular transfer was complete. At the very best, polling can be a very inefficient process, but in a real-time environment it is generally impossible for polling to satisfy the data-transfer requirements. In practice it is essential to respond very rapidly indeed to any demands of external devices, and this is done by introducing external interrupts.

The form of interrupt scheme implemented in different minicomputer architecture varies enormously. In certain systems there is only a single external interrupt line which signals to the CPU that some external demand exists, and it is then up to the supervisor to identify which of the external devices requires servicing and what action needs to be initiated. In other schemes there is both an external interrupt line and a separate set of lines used to signal to the CPU (or the supervisor program) that some action related to DMA transfers is required. Such “service” interrupts can be used, for example, to meet the supervisor requirements of Fig.3, and they can result in very rapid responses to external conditions.

Whatever the precise form of demand handling, interrupt schemes usually have an associated means of introducing a priority in the response. Clearly this is of great importance in real-time applications, since a higher priority real-time process may need to be initiated even while a lower priority demand is being serviced as a result of some previous interrupt.

Here again, there have been many different ways in which a priority interrupt system has been implemented. In some minicomputers it has been done simply by physical position of a device as coupled along a data channel; the further away from the processor, the lower the priority. In other minicomputers, the priority is determined by having several different interrupt lines each with a different priority. Sometimes the priority is determined by a “status” code on a set of interrupt lines and, yet in other systems, sometimes different priorities may be determined by some combination of all the techniques mentioned.

In general, in any interrupt involving the CPU, there will have to be a break in the running program to service the interrupt if it is of higher priority. In such cases it is normal to save the contents of all the relevant registers in the CPU and elsewhere and then branch to the appropriate interrupt service routine, either a general one which then has to identify the interrupt source or one special to the already identified interrupt source.

Clearly, with a common bus architecture, where every data transfer as well as transfers of control information have to contend for processor time along the common path, the question of demand handling assumes the greatest importance. In particular, the time necessary for handling any demand involving transfer of bus mastership must be minimised. One way in which this has been done is to introduce so-called vectored interrupts, so as to avoid the time-consuming task of identifying the source of an interrupt. In vectored interrupt systems, the interrupt source itself provides (either directly or indirectly) the address in memory of the location of a routine specific to service that particular source. Usually the interrupt “vector” which contains the service program address also contains the “status” of the interrupting device. In certain cases of vectored interrupts using a common bus system, such status information can be useful in establishing that a transfer may be initiated between two coupled devices without interrupting the operation of a third.

We have seen that the priority interrupt structure of a minicomputer can be quite complicated and that the range of different implementations is very great indeed. Modern techniques using register stacks and micro-processing, while helping to reduce the interrupt handling time, also serve further to increase the complexity. There is a constant introduction of new features in minicomputer systems, designed to speed up interrupt handling, but many of them introduce dangerous compromises. One example of this is the scope of CPU register contents by hardware, leading to an actual loss of time in certain cases when the CPU is not required to service the interrupt.

Certainly there is very little homogeneity in the interrupt handling schemes offered by different minicomputer manufacturers, and there is very little stability even from one single manufacturer. In the case of micro-computers the situation is particularly bad, and we have seen certain manufacturers actually introducing new common bus structures on the micro-computer systems they offer to replace their own minicomputers. But, in any case, before we can complete our conception of the usefulness of minicomputer I/O and demand-handling systems for data transfers, we also have to consider the associated software.

3.5 I/O Software

It is not the purpose here to review all the programming techniques developed for handling input and output to minicomputers, but rather to illustrate the scope of the problem. The first point to make is that the repertoire of I/O functions in any given modern minicomputer is usually quite small, but it differs greatly from one manufacturer to another. Moreover, even within one single minicomputer, there may be different I/O commands for the different types of I/O; for example, there could be one group of commands for DMA, another group for program-controlled I/O and so forth.

But of course, if the range of commands necessary for I/O differs from one minicomputer type to another, the spectrum of techniques used for demand handling is even broader since, as we have seen, these techniques depend not only upon the I/O commands available, but also the hardware associated with the data channels or highway as well as the nature of system supervisor.

In recent times we have seen a certain simplification of I/O software due to the growing use of common bus structures. Thus, not only is the command repertoire in these cases relatively simple, because I/O data transfers are just special cases of general register-register transfers, and because arithmetic and logical operations on both
full words and bytes can apply equally to data as to CPU operations, but the vectored interrupt scheme allows the introduction of much more modularity into the handling of I/O demands. Nevertheless, in spite of all the recent developments in minicomputer architecture, and in spite of the long experience now in dealing with real-time problems, it is fair to say that the problem of software driver for any new on-line device is one that generally has to be solved anew.

3.6 Is the Minicomputer Suitable for Data Transfers?

Having reviewed briefly the I/O structures of modern minicomputers, we come back to our basic question as to their suitability for data transfers in a real-time environment. Well, the first thing to say, in all fairness, is that one probably could choose one range of minicomputers, and it would only be quite adequate at the technical level to solve most I/O problems. Indeed, this is exactly what has happened in an enormous number of cases throughout the world. Unfortunately, the solution to our problem is not quite so simple as just defining a certain range of minicomputers as a standard.

The first point is that different minicomputers have different properties, and some may be more suitable for certain tasks than others, even with the same external devices coupled on-line. The second point is that coupling of an external device to a minicomputer may depend upon the system configuration and upon the software or operating system available, so that even the same device coupler or software driver may not function correctly when moved from one minicomputer to another, identical in components but different in configuration. In yet other cases there is no separate device driver or module (either hardware or software) to transport from one system to another, because they are embedded in a more complex unit concerned with coupling many devices together. This question of lack of either "upward" or "downward" compatibility, as it is termed, is an enormous and frequently encountered problem, where all the work necessary (sometimes many man-years) to couple some device to a minicomputer has been performed, but the result is not in a form that can be transported, even between minicomputer systems of the same manufacturer.

So the problem reduces to that of compatibility and modularity, which is the only way to minimize the enormous task of data transfer. Manufacturers themselves have realized this by trying to offer "standard options" for I/O coupling, for example, by providing hardware modules with well-known electrical conventions such as TTL, and also by providing complete packages, both hardware and software, for coupling certain peripherals. Unfortunately this gets us nowhere, not only because we would be constrained to one manufacturer of minicomputer, but also to that range of external devices which that manufacturer felt to be of interest. In reality we need to be concerned with many different types of minicomputer, and, in any case, with a vast and ever-growing number of external devices.

The oft-quoted example that, if there are N different minicomputer types and M different kinds of peripheral, then there need to be N x M different coupling adaptors or interfaces (to say nothing of software) is actually a considerable underestimate, since even the packaging or power requirements or some other factor may prevent the transportability of a given implementation.

The conclusion here is that to rely upon the I/O system of a single manufacturer would not effectively solve the data transfer problem, because it would be too restrictive; and to use many different I/O systems directly would be too expensive in terms both of costs and of manpower. The only alternative is to define some other widely accepted intermediate standard for coupling external devices, and we shall now review what has been done so far to introduce such a standard.

4. Standard Options for I/O Systems

There has been an enormous number of different attempts to introduce some kind of standard for I/O, but they fall mainly into one of two classes.

The normal way of coupling devices is shown in Fig.5, where it can be seen that there are N device couplers for N devices on a single minicomputer. They all need to be developed separately for each new device and for each new minicomputer, and there may even have to be two different controllers or couplers for program control and for DMA. The first type of standard I/O solution is shown in Fig.6 where an intermediate standard interface is introduced, or rather a set of standard I/O ports. In this way, although it is still necessary to have N different device couplers (1), there is only one type of system coupler (IP) between the standard interface and minicomputer I/O highway. Thus the device couplers do not change in moving from one minicomputer to another, and the system coupler unit need only be developed once for each minicomputer (although there still have to be N of them for N devices).

The second type of option to implement a standard is not to have a standard set of ports, but rather to have a new standard I/O highway and to translate to this highway from the minicomputer data channel or I/O Bus by means of a single system converter or adapter.
This solution is shown in Fig.7, where (IM) are interface modules which need to be implemented once only in the life of the device (D), and the system converter needs to be developed only once for each minicomputer. As we shall see, there can be several different configurations developed from the basic idea shown in Fig.7.

Let us now look at some actual examples of standard solutions.

### 4.1 British Standard Interface

One of the best known implementations of the structure shown in Fig.6 is the BS 4421 system, originally devised in the UK by the National Physical Laboratory but later elaborated and defined as a British Standard in 1969. This system is based on the notion of data sources and acceptors which couple to each other across a "standard" interface, which defines the following:

1. **Physical format**
2. **Electrical signal specifications**
3. **Logic levels and line allocations**
4. **Data transfer mode and control**

As can be seen from Fig.6, there will need to be an adapter unit to match the real (but different) devices to a given minicomputer, only N of them will be identical. At each of the unions across the interface, at ports, there is complete standardization of the data transfer at the level of 8-bit byte.

BS 4421 defines a physical connector with 18 or 34 contacts depending upon whether single or twisted-pair lines are to be used. The lines are allocated for data, error detection and control, the data being transferred in 8-bit, byte-serial mode with a single parity bit. The signal levels are defined in a way appropriate to discrete components (such as transistors) and are different for data and for control.

There are seven control lines, with two used to signal the ready states of source or acceptor, one to signal if the byte parity in transmission is valid, and another one to define whether no or not parity is being used. The other three control lines are used to control data flow, two for "hand shaking" and one to signal the last byte of a sequence in block transmission. The way in which hand shaking is achieved using the two "Acceptor Control" and "Source Control" lines is shown in Fig.8.

There is very little to say about BS 4421 except that it has not come into widespread use and is unlikely to do so. First of all, the data width and signal specifications are wholly inappropriate to modern technology and to most modern requirements. Secondly, the hand shaking mode together with the narrow data path makes high speed transfer of long blocks extremely difficult to achieve. Finally, BS 4421 says nothing about demand handling, multi-source addressing or priority arbitration. In summary, it is no more than was originally intended, namely, an interface standard — and not a highway or data-bus convention. Even for data acquisition BS 4421 is not particularly useful, and it is really best suited to certain communications applications.

### 4.2 The MEDIA System

Quite different in concept from BS 4421 is MEDIA, an acronym for Modular Electronics Digital Instrumentation Assemblies, which is a system marketed by GEC Ltd under licence from ICI. It is of comparatively recent availability, having been released commercially only in 1975. The system is fairly simple in conception but rather restricted in implementation. The design of MEDIA owes much to the original intention, which was to implement a more or less self-contained system for industrial process control.

The MEDIA system adopts a wide range of different conventions, of which the principal ones are as follows:

1. Physical formats for modules, crates
2. Two types of highway, and associated interconnection plugs
3. Internal logic levels, addressing schemes and control sequences

- 15 -
All these conventions are not stipulated rigorously in the manner of an international standard, but appear rather to be defined by the actual implementations of different units and system components by the single manufacturer involved.

Fig. 9 shows a representative MEDIA configuration. It can be seen that two highways are involved, termed passive and active respectively, and that the two are intercoupled. The number of lines, the logic levels, the addressing structure and control are implemented separately for both highways, and both highways are essential to synthesize any system other than a very rudimentary one.

MEDIA modules, a typical one of which is shown in Fig. 10, fall into one of two categories and plug into a bin or crate (Fig. 11). Passive modules, with only a single address, can only send or receive a single 16-bit word on receipt of an appropriate command, and no passive module can genera implicitly, although a dataword can clearly be interpreted eventually as an indirect address or implicit command. Active modules, which can only plug into an active bin, can generate commands, but have to contend for access to the active highway. Resolution of contention is by means of a hard-wired priority unit and by a rather inflexible time allocation scheme. The introduction of program control and processor capability is by modules on the active highway which either contain "intelligence" or which interface to a remote minicomputer.

In assessing MEDIA as a contender for standard I/O Highway in place of the I/O Bus of some minicomputer, it must be said that the system certainly offers something not available from most minicomputer manufacturers, namely, the means to couple signal sources and sinks (both digital and analogue) without bothering about many of the details. Also MEDIA is certainly modular and also not slower than most minicomputer I/O data channels. But that is about all we can say on the positive side. It is not that MEDIA is not useful, for manifestly it can help solve some problems of visibility very effectively. However, MEDIA is not a standard and, in its present form, unlikely ever to become one.

The plain facts are that there is no complete and explicit specification of the MEDIA Highways, nor of the associated protocols, and there is no definition of such things as the arbitration scheme on the active highway. These all exist in one particular implementation by a particular firm, and no doubt work exceedingly well, but there is nothing to tell an independent designer what rules he should follow if he wishes to develop either modules or interfaces. The question of demand handling is entirely unspecified in MEDIA except at the active highway level, to which the overwhelming majority of applications modules can only couple. With the exception of certain special-purpose process controllers, there appears to be only one MEDIA interface to a minicomputer, namely, a set of modules to enable a PDP-11 to be coupled.

Thus we must disregard MEDIA any further as a candidate for standard I/O system. Indeed, even if someone wished to use MEDIA and to construct alternative modules or controllers, he would presumably first have to obtain a licence from the two firms concerned!

We have to look further, to see if there is a more generally specified I/O standard. Such a standard is the IEEE 488 specification.

4.3 The IEEE 488 Bus

This specification 8), published in mid-1975, is the outcome of work by the Hewlett-Packard Company to develop an interface standard for laboratory instruments. The result of the work was an implementation (called the HP-IB) which is now a standard accepted not only by the IEEE but also by ANSI and the IEC.

The IEEE 488 Standard defines completely a data bus and, in electrical, mechanical and logical aspects, any port on the bus. Not only are these specifications complete and unambiguous, unlike the MEDIA system, but also they are defined in a way totally independent of any device attached to a port and, particularly, independent of any processor or minicomputer. Considerable attention is given to the signal specifications, and the highway protocol is also defined.

Fig. 12 shows the bus structure and different categories of devices coupled to ports on the IEEE 488 Standard. There are 16 lines in all in the highway, of which eight are for data, transferred in bit-parallel, byte-serial mode. Three lines are used to regulate the transfer of each data byte in a "hand shake" manner, while the remaining five lines are for control purposes.
The hand shake principle here is similar to that used in the 80 Interface, only three lines are used, one (DAV) by the sender and two (NRFD and NDAC) by the acceptor. The three lines are used to signal as follows:

DAV - Data valid
NRFD - Not Ready for Data
NDAC - Not Data Accepted

Fig.13 shows a flow diagram for a typical hand shake procedure, where it should be noted that there can be more than one Acceptor but only a single Sender (or "Talker"), as it is termed in the vernacular of IEEE 488. With more than one Acceptor, since the data transfer is asynchronous, the cycle time is governed by the speed of the slowest device.

The other control lines serve different functions, but most important is the ability to define any given coupled device as one of three types: a Talker, which sends data over the bus, a Listener, which receives data, or a Controller, which directs flow of data and defines which devices may "talk" and which should "listen" at any given time. To do this, there is an addressing scheme which uses seven of the eight data lines, together with an appropriate signal on one of the control lines. The eight data lines are bi-directional.

The actual bus is a multi-core cable to which are coupled defined connectors (Fig.14) allowing both chain and radial configurations. Signals along the cable are also defined with TTL logic levels. Although the address space allows 31 addresses (or more if two successive bytes are used), in practice only 15 devices are recommended because of electrical limitations.

Any device can be coupled to a port of the Bus, provided it has a mating plug and can respond to the relevant signals. The address of the device is usually determined by a set of switches manually, but there are no other restrictions on the physical format of device. There is the facility for any device, including a Listener, to generate what is called a "Service Request", that is to say, a demand, and a Controller has functions which respond to such requests. Moreover, there can be transfers of "status byte" between devices.

Controllers can be programmable or otherwise, and there can be more than one, although clearly only one controller may be active at any one time. There is no explicitly defined arbitration scheme for passing control from one bus master to another, although certain primitive commands are available which would serve the purpose. The cycle time of the Bus can be as high as 1 Megabyte/sec, so manifestly there is no difficulty in using a minicomputer as a bus controller.

How can we assess the IEEE 488 Bus as a solution to our problem of standard for data transfer to minicomputers? Well, there can be no doubt that it certainly serves the purpose for which it was intended, namely, to inter-couple laboratory instruments and to control their data flow by means of some "intelligent" controller. Unfortunately, that is about as far as the standard does go, although it is fair to say that a number of firms in addition to Hewlett-Packard are beginning to make their instruments with an IEEE 488 port coupler, so that they may either be used manually or on-line.

So far as experimental data acquisition is concerned, or for that matter industrial process control on any large scale, the IEEE 488 Bus is not very appropriate for a number of different reasons.
First of all, the number of devices that can be coupled is very small and, in any case, the address space is also small and without any sub-address or multi-register option within a given device. Secondly, the data width is too small; 16 bits would have been much better. The demand handling facilities are rather weak and block transfer is more or less overlooked. Finally, although the system is, in a certain loose sense, modular, the modules are totally undefined.

The merits of the IEEE 488 Bus are its low cost and device independence. For the purpose of a general data transfer standard, alternative to a minicomputer I/O Highway, these qualities are unfortunately inadequate. We still need a standard which has a wider data path, which can handle block transfers with ease, and which has defined modules. There appears, so far, to be only one candidate to fulfil this need — CAMAC.

4.4 CAMAC

CAMAC started off as a convention for modular electronics agreed by ESONE, a committee of European national laboratories mainly concerned with nuclear and elementary-particle physics and with nuclear energy. The genesis of this convention was in about 1968, but many of the features evolved from the earlier NIKS standard and from the Harwell 7000 system developed by the AEHR. By 1969 this convention was published to define the mechanical, electrical and logical characteristics of a crate and data highway, and this was later extended to become a full European standard in about 1972. At this time also it was adopted as a US standard of the AEC, but it has since become the standard IEEE 488-1973.

Up to the present time CAMAC has continued to have additional features defined, some of which have also become accepted as IEEE and IEC standards, but it is true to say that the original definition of crate with dataway has undergone little change. The present status of CAMAC is that a very wide range of different features have been formally defined in a manufacturer and indeed device independent way. There is a vast body of literature devoted to the standard itself and to its applications.

It is not the purpose here to review all the aspects of CAMAC, for which purpose readers are referred to an excellent bibliography 77, but instead to view it in the context of our present discussion of data transfer. Therefore, let us summarize the principal attributes of CAMAC, which is concerned with the coupling of devices to an "intelligent" controller.

All devices in CAMAC are coupled by means of modules of standard format. These modules, which may be the devices themselves in certain cases, can occupy any one or more of 24 stations in a standard crate, at the rear of which is an interconnection highway called a Dataway. The Dataway is an 86-line highway with two 24-bit data paths, and each crate has a Controller, coupled to the two end stations 24 and 25 or a crate. The CAMAC system defines two such Crate Controllers (Type A and Type L), but there may be others as well. Thus, for example, the CAMAC system may stop there for certain applications, since a Crate Controller may be "intelligent". Fig.15 shows a Crate and typical module. Along the Dataway the addressing and command structures are completely defined, and there is also a comprehensive range of features to allow demand handling. Important aspects of the Dataway are that each module may have many sub-addresses, and that the status of each may readily be determined.

Fig.15 CAMAC Crate with single-width module

Fig.16 CAMAC Branch
Crate may be used singly, or they may be configured and intercoupled in a number of ways. The most common interconnection is the chain configuration shown in Fig. 16, in which crates are coupled by means of a parallel branch or highway. One such highway is defined within the CAMAC standard, which allows up to seven crates to be coupled by means of the Type A controller. In the CAMAC Branch, the Branch Highway is completely defined, as is the port to which there must be connected some intelligent controller or branch driver. Another interconnection configuration is by means of a serial highway (Fig. 17), and the CAMAC standard defines such a highway, operating in either bit-serial or byte-serial mode, with the corresponding serial crate controllers (in place of Type A). Clearly, there may be other methods of crate interconnection also, and these include data-link modules for CAMAC-CAMAC coupling, but only the Branch and Serial Highways are defined standards.

Thus CAMAC defines the electrical, mechanical, signal and logical specifications for all the principal interfaces between external equipment and an intelligent controller, and even defines standards for external analogue signals. Detailed information is given in the specifications about such aspects as timing and multi-address operation.

Particular attention has been paid, in formulating the CAMAC standard, to the question of block transfer of data words. Thus already the command repertoire of CAMAC and the Dataway organization make block transfers of any general kind readily facilitated. However, in a separate specification, CAMAC does illustrate several conventional methods of block transfer which make use of one of the dataway status lines, the Q-line. In this way, conventions are defined both for sequential data transfers from a single sub-address and for transfers from contiguous sub-addresses.

Unlike all the other conventions we have considered, CAMAC has evolved to a higher level in an extremely important respect, namely, that there exists such a thing as "CAMAC" software. As a result of the very extensive installation of CAMAC hardware systems, and the ease of transportability of modules and indeed of complete sub-systems, there has been a very strong motivation to develop transportable software. This has mainly taken the form of defined languages, but also there have been a number of high-level software packages. Thus, for example, there is a version of BASIC with additions of CAMAC Macro-commands, and there is even a certain move to use FORTRAN with conventional CAMAC-communicating sub-routines.

There can be no doubt that CAMAC can and indeed does perform the function we were seeking, namely, to be the intermediate standard highway system between minicomputer and terminal equipment. CAMAC is of course not restricted to minicomputers, nor indeed to the sole application of data transfer, but it is evident that it serves eminently well for the purpose of data transfers to and from minicomputers. This situation is attested well by the fact that CAMAC is used now in at least 20 countries and there are more than 70 manufacturers who supply CAMAC components. It is also probably true to say that there is no minicomputer of any general availability for which there is no CAMAC interface or driver available.

Thus, as a standard alternative to any particular minicomputer highway, CAMAC has served extremely well and has so far not imposed any particular limitation on either system configuration or on data throughput. We now have to examine the future, to see to what extent CAMAC will continue in its present form and how it might evolve.

5. CAMAC: The Present Status

It is clear that CAMAC can satisfy most data transfer requirements into minicomputers, and hence into larger systems also. And for this reason, it is not surprising that it has been used on a very large scale, not just in nuclear and particle physics but also in industry, medicine and in space research. We can see that this situation has come about because, on the one hand, CAMAC makes the exploitation of minicomputers in data-transfer activities very much more effective and, on the other hand, CAMAC does not impose on minicomputers any great limitation. Indeed, it can be said that, without CAMAC in elementary-particle physics, we just could not have achieved in any other way the results that have been attained, because we would have required many more man-years of effort (plus cost) than was available.

So, using CAMAC, up to the present at any rate, there has been a good match between the external system requirements of applications involving data transfer and the capabilities of minicomputers. To see whether this happy situation can continue, let us review the principal attributes of such total systems.

The first question is whether the number of modules available is sufficient and whether they are sufficiently "fine-grained". Here there are no difficulties, since one can have an almost unlimited number of crates, either by using many parallel branches or by crate-crate coupling of non-standard kinds, for example, by direct coupling.
between the I/O highway of a minicomputer and a special (Type U) crate controller. Similarly, it is not module size which imposes any limitation, but rather what one can put in it. Indeed, modern technology has allowed us to produce certain quadruple-wide control units in single-wide modules, where previously we could only have one unit. The address space of CAMAC also appears so far to impose no great restriction on modularity.

So far as data width is concerned, there appears to be no difficulty, since the 24-bit CAMAC word is nearly always more than is required. Indeed, for the overwhelming number of applications (and minicomputers) 16 bits would have been sufficient. In the coming period, we will have more and more 32-bit machines, but data from external sources will probably not require more than 16 bits for a long time yet.

The question of data-transfer rate is important, since there is a Datasync limitation both in peak and average rate to about 1 Megaword per second. This is, however, an enormous average data rate, which cannot be sustained for very long by any minicomputer. Indeed, this is a fact very frequently overlooked by certain of those who complain that CAMAC does not operate "fast enough" for their application. In point of fact there is nothing to stop CAMAC handling very much faster peak data rates into modules from experimental equipment, and indeed there are many buffer modules which do precisely that, handling peak rates of maybe 10 Megawords per second, but for relatively short blocks of data. What CAMAC cannot do (in standard form) is to remove data from buffer modules at faster than 1 Megaword/sec. So, for any application that really requires very much higher average data rates, CAMAC is not of much use (but then, neither are minicomputers!).

Concerning block transfer, as has already been mentioned, CAMAC does provide both for non-standard and conventional modes, so that data transfer under DMA into most minicomputers can be readily achieved. Actually, CAMAC, although possibly somewhat slower in block transfer than some minicomputer data channels, is considerably more powerful in the number of modes available, particularly in respect of being able to partition blocks in many data streams (for example, from contiguous modules or sub-addresses). But there is a limitation both in speed and operation, which we will come to later; this limitation has more to do with applications than with minicomputers.

Demand handling is an aspect that is extensively treated within CAMAC, and certainly imposes no limitation on any attached minicomputer. On the contrary, it is difficult for most minicomputers to handle all the interrupts that can come from a complex CAMAC system, and there usually has to be additional hardware to help, often in the form of a "grader". Having said that, it is clear that the original CAMAC specifications were formulated assuming a single intelligent controller only. This is fast becoming not the case, and already in the Serial Highway definition there has been a recognition of the problem by the introduction of a "demand message" rather than a unique signal.

The final aspect of CAMAC we need to include is cost, since there is certainly an additional expense involved in coupling. Here there can be little doubt that, above a certain very low threshold, the cost of coupling equipment to a minicomputer can be very much less using CAMAC than without, provided that all costs are properly taken into account.

6. Trends in Data-Transfer Systems

We have seen that the data transfer problem has been handled fairly well up to now, and that CAMAC has played a very important role indeed. Can this continue?

Well, there are many applications areas where CAMAC, even in its present form, can continue to give an excellent service and indeed to grow considerably. For example, in process control, in laboratory automation and in communications, there can be little doubt that CAMAC has much to offer. However, there are two important trends which strongly motivate an improvement in available data-transfer systems, including CAMAC. The first of these is new technology, which makes possible certain developments previously impracticable; the second is that experimental requirements are becoming more demanding. Let us examine these briefly to see what they suggest for future organisations of data-transfer systems.

6.1 New Technology

The main trend which we need to note is the availability at low cost of monolithic circuitry with a moderate degree of integration (LSI). Particularly of importance is the appearance of memory, both ROM and RAM, which is both of small physical size and of high-speed (500 ns cycle or even faster), and also of processors. Included in the latter are not only micro-processors of all kinds, both word and byte processors, but also bit-slice and programmable logic arrays (PLA).

The main impact that the new technology is having in that it has become possible not just to place "intelligence" (that is to say, processor + memory) physically close to the hard-wired electronic circuitry that needs to be controlled, but also to have as many separate intelligent devices as are required. In the terminology of today, the new technology and associated reduction in cost has made possible systems with "distributed intelligence". We have also to note that the reduction in cost of memory is so great that this will also have an impact on the way processing is done, replacing hard-wired functions more and more by microcode.

6.2 New Requirements

It is of course impossible to review all the new requirements of data-transfer, but perhaps we can identify the main trends, all of which depend upon the possibilities of the new technology. Possibly the most important difference from the past that is emerging is the necessity to distribute the flow of data. Whereas it used to be the case that data was more or less funnelled into a single processor (and into a single memory), it is becoming more and more important to be able to partition data in a co-ordinated way among different intelligent devices and buffers. To coin a slogan, "distributed processing implies distributed data".

Stemming from this trend, we can see that a number of dosisdata emerge in any future data-transfer system. For example, we need the following:

1) Ability to accommodate several processor and memory-containing devices in a modular way
2) High-speed (200 ns per word) data transfer between such modules
A defined means for recognition and servicing of demands between modules, some of which are intelligent

Incorporation of arbitration between different but inter-coupled intelligent devices

It may well be that, in order to satisfy the first requirement, we may have to choose wider data and address paths, particularly if there is a frequent need for transferring large data blocks from one memory module to another. We have to remember that modern technology allows us to put as much as 64k words of store in the size of a CAMAC module, and therefore we will need 16–20 bits of address at the module (that is, sub-address) level. But we should note also that we can always indirectly address or page.

Added to the above general requirements, we can also identify a number of more special ones which arise mainly in areas such as readout from certain spatial detectors of elementary particles. The problem here, which is a special case of encoding the results of pattern recognition, is to be able to transfer only significant data from very large arrays of data sources. For example, we need not know the addresses of only “triggered” wires in large proportional chambers, without having sequentially to scan all the separate wires.

The question of pattern recognition and pre-processing is a very important one. However, it is not yet clear whether this should be included within a data transfer system or not. Certainly, if this component could be included, there would be an increase of modularity and integration into the subsequent data-transfer chain. However, there could also be great difficulties, because there are usually several orders of magnitude difference in the data rates and volume before and after pre-processing (indeed, that is the whole idea!). So, to combine both levels of data flow into one single system could be very difficult and, maybe, should not be attempted. In CAMAC, for example, this problem has been avoided by separating data signals between the front panels of modules and the Dataway, with an order of magnitude (or greater) capability in the peak speed of handling data between the two.

Let us see therefore to what extent the new technology and the new requirements can be accommodated in existing data-transfer systems and, in particular, let us review the changes that are being made in CAMAC.

7. Current CAMAC Developments

Not surprisingly, all the considerations discussed in the last section have been well known to the CAMAC community, and many attempts have been made to incorporate features into CAMAC that, while retaining downwards compatibility, makes possible certain of the new requirements. There has also been a very active discussion about how to modify the CAMAC standard perhaps to meet the growing range of applications. Let us examine some of these developments.

7.1 New Technology in CAMAC

Monolithic circuitry has been used in CAMAC modules for a long time now, but the advent of programmable devices and LSI memory has manifested itself in several different ways. First of all, we have seen the appearance of “intelligent” modules, in which programmable devices (usually driven by ROM or PROM units) have been used to replace hard-wired electronic logic. So, for example, there are available intelligent display drivers and also peripheral couplers. Also there are intelligent communication modules, particularly modem drivers. Thus, either singly or in local clusters, we have seen the introduction of new technology at the module level. And this has had a deep influence on the pattern of data transfer in CAMAC, and may indeed have prolonged the life of CAMAC rather than shortened it. The point here is that the introduction of module processing capability has actually reduced some of the Dataway data-flow both in speed and volume. It is indeed an aspect of the introduction of pre-processing.

Another development has been the introduction of LSI units in the Crate Controller itself. In some cases here, the crate controller has been of adequate power to control completely all the crate modules without any external minicomputer at all, the functions of the minicomputer being taken over by the “intelligent” crate controller. Once again, the CAMAC system has been actually improved by the introduction of LSI technology, since the crate with intelligent controller is, for certain applications, a very economical solution.

7.2 Distributed Intelligence

Of course, many of the developments just mentioned have only been possible because certain modules within crates have had an interconnection through lines other than the Dataway. For example, there have been memory modules and other units separate from processing modules, and these have been connected by means of an auxiliary data and address path. And so we observe once again an analogous situation to that which obtained in the case of NIMs where, prior to CAMAC, we saw the emergence of numerous ad hoc data highways coupled to NIMs modules. There are now many different CAMAC systems in which modules, as well as coupling to the Dataway, are interconnected using a private bus (Fig.18). Sometimes even there has been a mixture of bus interconnections, with a minicomputer bus being used in DMA mode, coupled directly to a module which drives a local private bus.

In one particular area, there has been a move to define an auxiliary bus for CAMAC, and that is for the purposes of Auxiliary Crate Control (ACC). The problem here is the introduction of one or more controlling devices in a Crate in addition to the Crate Controller (CC), for which there clearly needs to be access, external to the Dataway, to the address and interrupt lines. This problem has been resolved by the definition of a supplementary CAMAC standard which deals with multiple controllers in a CAMAC Crate, interconnected by an auxiliary bus (Fig.19). In this supplement, not only is the auxiliary bus defined, but also is the bus protocol, the interconnection standard to the bus and a revision in the definition of crate controller which still permits downward compatibility.
It is interesting to note also that some attempt has been made to provide the means for resolving Dataway and Auxiliary Bus contention by the provision of arbitration lines. In this way a bus master can always be defined at any given time in accordance with an appropriate priority structure.

8. The Future

As always the future is difficult to predict. Particularly so in this case, since we are at a period, on the one hand, of enormous technological developments and, on the other, at a time when the investment in both software and hardware for CAMAC has been very great also. Thus there are good technical reasons for changing CAMAC, but good practical and economic reasons for not. At the same time, there still remains an immense range of problems, still unsolved, for which CAMAC can provide a most efficient and economical solution.

It is already clear that the CAMAC Dataway and associated Crate Controller impose certain speed and configuration constraints on the kind of data-transfer systems we should like to build. In any case, it is also evident that minicomputer I/O highways are also becoming much faster and, in some respects, more powerful (the Megabus of Honeywell, for example, is 72 lines wide and can have a 385 ns bus cycle). Thus, the Dataway must be replaced eventually, even if the mechanical format remains.

I believe that two quite separate developments will occur, probably concurrently. To start with, CAMAC will itself develop in two respects: firstly, there will be more and more standardization of an auxiliary bus, possibly with the incorporation of an interrupt sub-bus instead of separate lines from crate stations; secondly, there will be no agreement on compatible uses of the Dataway, probably by interleaving fast data and command transfers between Dataway cycles. The second development will be to use a CAMAC-like format, possibly with a similar Dataway, but with entirely new organization and cycle time. This second development could readily be achieved already, since we have all the technological means and we know what the main parameters should be. For example, a typical specification of data highway could include:

i) Cycle time of 100 ns
ii) 32-bit bi-directional data path (with optional 16 + 16 bits)
iii) 24-bit address bus
iv) 8-bit interrupt bus
v) Control lines to permit:
   a) bus arbitration
   b) asynchronous transfers
   c) status signals
   d) block transfers in various modes

All this would be quite straightforward to do, and it would also be fairly easy to define appropriate Crate-Crate coupling, which would probably be at the level of Module-Module with a defined communications protocol for transmission over long or short distances in block-transfer mode.

If the developments just mentioned were to proceed together, then the next phase of evolution would occur in a natural way, since it would become evident fairly soon which of the two systems would dominate at any given time, or, more exactly, what would be the relative balance between them. Indeed we could see a certain convergence between the two approaches, namely, the new data highway becoming identical in properties to the auxiliary bus of the old CAMAC system. In that case the transition from one system to another would occur automatically as the Dataway gradually faded from use. Such a smooth transition could be immensely beneficial since there should also be an accompanying smooth migration and evolution of software. Indeed, this could be a determining factor in the future evolution of CAMAC, since software is already the component which requires the greatest manpower effort in most modern systems.
Of course, the evolution just envisaged may not happen. And indeed there are already several different alternative suggestions to implement a completely new data acquisition scheme, such as FASTBUS[10], which has been proposed by a consortium of US Laboratory representatives. It is much too early to say which direction developments will go, but it is clear that the choice will depend finally not just on technical considerations but also on the interest and support obtained from industrial manufacturers. Let us hope that, whatever the choice, the next system developed for data transfer will be as successful as has been CAMAC.

Acknowledgment

The permission is gratefully acknowledged of GEC-Elliott Ltd. to reproduce Figs. 9, 10 and 11, and of the Hewlett-Packard Company to reproduce Fig. 14.

References

1. See, for example:

2. See, for example:
P. Kunz. "Micro-circuits as components in high-energy physics". ibid.

3. See, for example:
L. Monsen-Krohn. "The minicomputer spectrum today and tomorrow". (Presented at this School.)


ABSTRACT

Microprogramming is an inherently elegant method for implementing many digital systems. It is a mixture of hardware and software techniques with the logic subsystems controlled by "instructions" stored in a memory. In the past, designing microprogrammed systems was difficult, tedious, and expensive because the available components were capable of only limited number of functions. Today, however, large blocks of microprogrammed systems have been incorporated into a single I.C., thus microprogramming has become a simple, practical method.

1. INTRODUCTION

1.1 BRIEF HISTORY OF MICROCIRCUITS

The first question which arises when one talks about microcircuits is: What is a microcircuit? The answer is simple: a complete circuit within a single integrated-circuit (I.C.) package or chip. The next question one might ask is: What circuits are available? The answer to this question is also simple: it depends. It depends on the economics of the circuit for the semiconductor manufacturer, which depends on the technology he uses, which in turn changes as a function of time. Thus to understand what microcircuits are available today and what makes them different from those of yesterday it is interesting to look into the economics of producing microcircuits.

The basic element in a logic circuit is a gate, which is a circuit with a number of inputs and one output and it performs a basic logical function such as AND, OR, or NOT. Figure 1 shows the basic gate used in the popular TTL technology. It performs the NAND function, that is only when both inputs are TRUE does the output become FALSE. The truth table which describes the operation of the gate would then look like that shown in figure 2. From this basic gate one can form other logical functions. For example, the NOT function can be generated by tying the two inputs together as shown on the left of figure 3. It is usually represented by the INVERTER symbol as shown on the right of the figure. Another example is the OR function which may be generated from the NAND gates as shown on the left of figure 4 and usually represented by the symbol shown on the right of this figure. It can be shown that all the Boolean operations can be generated with combinations of the basic NAND gate.

The cost of a integrated circuit depends on the number of gates required to perform the desired function, but the cost of a gate depends on the number of gates in the chip. Figure 5 is a plot of

* Work supported by U.S. Department of Energy
the cost per gate versus the number of gates per chip. There are three distinct regions on this curve. The labor intensive region is where the labor of assembly, testing, and processing the order, as well as the fixed company overhead dominate the costs of the chip. In this region the manufacturer can double the number of gates on the circuit without changing its cost, thus the cost per gate would drop a factor of two. The silicon intensive region is the technically difficult region, where the manufacturer produces a small percentage of functioning circuits for his effort and hence the cost per circuit begins to rise rapidly. The flat central region is the region, where the cost of the circuit is proportional to the number of gates on the circuit. It is the optimal region for producing circuits.

As the technology of producing circuits improved, what was technically difficult at one time became standard practice at a later time. Figure 6 shows the cost curve for three periods of time. These periods correspond roughly to three generations of microcircuit manufacturing. The optimal region in the first generation, Small Scale Integration (SSI), had three to six gates per circuit. The circuits that were produced were simple logic functions and the technically difficult was a flip-flop. An example of an SSI integrated circuit package is the 7400 as shown in Figure 7. It is simply four independent NAND gates requiring 3 pins each. With the supply voltage and ground pins it makes the standard 14 pin package still in use today.

![Figure 6: I.C. Cost Curve versus Time.](image)

![Figure 7: 7400 Integrated Circuit Package](image)

When the optimal region for manufacture became 20 to 50 gates per circuit, the second generation of microcircuits was born: Medium Scale Integration (MSI). The semiconductor manufacturers faced a problem as to what circuits to produce, since the simple extrapolation of more simple logic functions per circuit runs into some problems such as too many pins per package. The problem was solved by producing larger blocks of digital systems such as counters, multiplexers, decoders, registers, etc., which were of general enough use that the manufacturer could sell them in large enough quantities to make a profit. An example of an MSI integrated circuit package is the 74157 as shown in Figure 8 (a). It is called a Quad 2-Input Multiplexer since it multiplexes one of two inputs to one output four times over. A single Select input controls all four channels. With one pin left over to make it an even number, the manufacturers have added a Gate to force the outputs to zero and one has a standard 16 pin package. The conventional symbol for this circuit is also shown in Figure 8 (b).

![Figure 8: Example of MSI Integrated Circuit Package, (a) Circuit, (b) Symbol.](image)

A few years ago, the optimal region of manufacture became 200 to 500 gates per circuit, Large Scale Integration (LSI), and the semiconductor manufacturers were again faced with the problem of what circuits to provide with these many gates. The problem was solved by producing an even larger block of digital systems so that we now find that microcircuits are arithmetic/logical processor elements, microprogram sequencers, direct memory access controllers, etc.

The LSI microcircuits will be the topic of these lectures. They offer the best economy because large subsystems of digital circuits are available on a single I.C. package. Within a given type of technology (e.g. TTL, ECL, MOS, etc.) they often produce faster systems because there is less loss of speed with interconnection between packages. They also reduce the amount of circuit board real estate required for a given logic system and large systems are less expensive to make.
With LSI microcircuits, the semiconductor manufacturers have made available large digital subsystems within a single I.C. But they still had to provide a means by which the circuit was flexible in its use in order to be able to sell enough of them to make a profit. The flexibility of these circuits was obtained in part by designing them to be used in a microprogrammed type of architecture. This is to say, that the function a circuit performs is controlled by an number of input signals which form an instruction word. The instruction word is assumed to come from the microprogram memory. The manufacturer also is making circuits for use where there is potentially the largest volume of users, which for digital systems is probably the computer and computer peripheral manufacturers. In this market, the microprogrammed technique of logic design offers many advantages as we will see in these lectures.

High Energy Physics is not a high volume user for semiconductor manufacturers. If we are to make use of LSI, we must, in general, bend our needs to those circuits which are already commercially available. In addition, in order to profit from the LSI microcircuits, we must learn the microprogram method of implementing digital systems, and we must be able to understand the digital subsystems that are available as a single I.C. In the following sections, we will study the basics of microprogramming from a point of view which is biased by the microcircuits that are commercially available. Then we will study in some detail a microprogrammed controller with a High Energy Physics application. Finally, we will study 2 of the most important LSI circuits which have become available.

2. BASICS OF MICROPROGRAMMING

2.1 COFFEE VENDING MACHINE

To understand the basics of microprogramming let us take a simple example: an automatic coffee vending machine. Figure 9 is a block diagram of such a machine which has two basic parts: the machine hardware and the sequential control logic. The coffee machine hardware is the system to be controlled. It contains the valves and solenoids that release the water, coffee, sugar, etc. which are needed to produce the desired result: a cup of coffee. The sequential control logic is the system controller. It sends signals to the hardware in the correct order and timing. It starts the hardware into operation when it receives a signal from the coin detection logic and alters the sequence according to what kind of coffee has been requested via the front panel push buttons.

The sequence control can be imagined as a series of steps, each lasting a fixed length of time, say 1/2 second. The list of steps might be as shown in figure 10. The coffee machine sequence controller could be implemented using combinations of flip-flops and one-shots as shown in figure 11. This approach is commonly called hard wired or random logic, and is typical of how designs have been done in the past. The advantage of this approach is that it uses the minimum number of logic gates and it is relatively simple for a given sequence.

The coffee machine sequence control may also be implemented with a binary counter and a read only memory (ROM) as shown in figure 12. In this figure, only one of the sequences has been implemented. The binary counter serves to count the steps and the ROM serves as a programmable decoder to produce the required signals at each step. Note that the input address of the memory is the output of the counter and that each bit of the memory's output is used directly as one of the signals for the hardware under control. In order to do the black coffee sequence, one would want the contents of the memory to be as shown in figure 13. A binary '1' corresponds to sending a signal, while a binary '0' corresponds to not sending a signal. A coffee machine sequence controller implemented in this way is said to be microprogrammed.

In order to include the other kinds of coffee one could increase the size of the counter from 5 bits to 7 bits and the size of the memory from 32 locations to 128 locations as shown in figure 14. The encoder circuit generates a binary code from 0 to 3 depending
on which of the push buttons was activated. This
code is then used as the two high order bits to the
counter when it is loaded. The loading of the
counter is under control by one additional bit of
output from the memory. Thus, for example, at memory
address 1 the load bit may be turned on so that the
next address of the sequence will be either 2, 34,
66, or 98 depending on the output of the encoder.

Figure 13: Memory Contents of Coffee Vending Machine
for Black Only.
2.2 GENERAL MICROSEQUENCERS

The coffee machine sequence controller is an example of a microprogrammed processor. The processor's memory contains two fields; the load control bit and the other bits to control the hardware signals. A generalized version of this processor is shown in figure 15, where the encoder has been replaced by an instruction register. The OP-CODE field of the instruction register contains the high order bits of the starting address of a sequence. The push buttons of the coffee machine have been replaced by the machine instruction.

In a more general processor, one may have sequences of widely different length and the circuit shown in figure 15 will lead to large areas of unused memory. The introduction of another memory, the MAPPING ROM, between the OP-CODE and the program counter will allow the flexibility of starting a sequence at any arbitrary address. The OP-CODE is used as the address of the MAPPING ROM and the output of the MAPPING ROM becomes the starting address for the program counter. This MAPPING ROM is shown in figure 16 and it is another example of using memory as a programmable decoder.

With the microsequencer shown in figures 15 or 16, the flow of the program can only be the next sequential address until another sequence is started when the LOAD bit is present. This sort of flow is show schematically in figure 17. At instruction 50 of the figure, for example, the next instruction can only be instruction 51. In this sequential flow the processor is said to execute the CONTINUE (CONT) instruction.

One useful way to add flexibility to the microprocessor would be to allow the program the jump to an address which is contained in the microprogram. A method of doing this is shown in figure 18. A multiplexer has been added between the output of the
A very important feature to add to this basic processor would be the ability to alter the flow of the program depending on the results of a previous operation. This is called CONDITIONAL BRANCHING and it can be implemented as shown in figure 20. The LOAD input to the program counter is now taken from the output of multiplexer which is called the CONDITION CODE MULTIPLEXER. One of its inputs is selected by part of the output of the microprogram called the CONDITION CODE field. Note that one of the inputs to the multiplexer is a logic '0'. When this input is selected, the LOAD input to the counter is always '0' so that the counter goes to the next sequential address. Another input to the multiplexer is a logic '1'. When this input is selected, the counter will always be loaded. These two inputs are necessary in order that this processor can execute the CONTINUE and JUMP instructions, respectively. When the third input to the CONDITION CODE MULTIPLEXER is selected, the counter will either go to the next sequential instruction if the conditional input is '0' or be loaded if the conditional input is '1'. Thus we have added the CONDITIONAL BRANCH instruction to the processor. An example of this flow is shown in figure 21 at instruction 53.

Figure 18: Microsequencer with JUMP logic.

Figure 19: JUMP instruction.

MAPPING ROM and the input of the counter. One input of the multiplexer is the MAPPING ROM while the other comes from a part of the output of the microprogram memory. The latter is called the BRANCH ADDRESS field of the microprogram memory. One additional bit from the microprogram memory is routed to the SELECT input of the multiplexer so that when the bit is in one level the output of the Mapping ROM is routed to the input of the counter and when the bit is in the other level the BRANCH ADDRESS field of the microprogram memory is routed to the input of the counter. This bit is called the ADDRESS SELECT (ADR-SEL) field of the microprogram memory. The flow of the microprogram can now be altered as shown in figure 19. After execution of instruction 53, the next instruction is 90. The processor is said to execute a JUMP (JMP) instruction at location 53.

Figure 20: Microsequencer with Conditional Branching.

Figure 21: Conditional Branch Instruction.
At this point it is appropriate to take a look at the timing of the processor. Figure 22 shows the time sequence of the signals within the processor. Each microinstruction starts with the leading edge (i.e., the '0' to '1' transition) of the clock signal. When this signal is received by the microprogram counter, it increments its contents by one. The change of its output does not occur instantaneously, however. Each logic gate within the counter circuit has a response time called its "propagation delay". Thus it is only some time after the counter receives the clock signal that its output switches to the next address. For example, with a standard Schottky TTL counter the delay from clock to output is 13 nsec.

The microprogram memory also has a delay between the time an address is presented to its input and valid data is available at its output. This delay is called the "access time" of the memory and for Schottky TTL memories it is on the order of 50 nsec. The period of time from the generation of a new address until the output of a memory is steady is called the "fetch" time. Note that for the microprocessor we are studying, the total fetch time is equal to the sum of the propagation time of the counter and the access time of the microprogram memory.

After the fetch time, the process under control of the microprogram memory starts its execution. Again this process is only finished after a delay called the "execute time" which may be on the order of 100 nsec depending on what is being done. At the end of this period we have the results which may now be saved at the leading edge of the next clock signal in say an accumulator. Thus the minimum cycle time of the microprocessor is determined by the sum of the fetch and execute times. With the next edge of the clock signal the processor starts the next instruction.

Let us consider the microprocessor timing when a conditional branch instruction is executed. In the timing shown in figure 23 microinstruction 1 generates a result upon which we wish to conditionally branch. The result of this instruction is available during execution of microinstruction 1+1, thus we should make microinstruction 1+1 the conditional branch instruction. At the time of the third microcycle, we can start microinstruction 1+2 or the instruction of the branch address depending on which path the result has taken us.

![Sequential Timing with Program Counter](image_url)

**Figure 22: Sequential Timing with Program Counter.**

![Conditional Branch with Program Counter](image_url)

**Figure 23: Conditional Branch with Program Counter.**

One of the frequent requirements of logic systems in High Energy Physics is speed. With speed in mind, one could ask why do we use up a whole microcycle to do a branch instruction? So let us consider for a moment how the timing would change if we attempted to do the conditional branch in the same microcycle as the execute. The condition upon which we want to branch would not be ready until the end of the execute time. It would be at that time that the condition would begin to propagate through the CONDITION CODE MULTIPLEXER and be presented to the LOAD input of the counter. Before the clock signal...
can be asserted at the counter, we must wait a period of time called the "set-up" time so that the counter can do the LOAD or COUNT function correctly. Using standard Schottky TTL circuits, the sum of the multiplexer propagation delay and the counter set-up time would add another 35 nsec to the minimum microcycle time. If the microcycle time is constant, then this additional time would be added to all microinstructions whether they contained a branch or not. Thus depending on the number of branches in a program and the execute time, program execution time may be faster with the separate branch and execute instructions.

A much more important improvement can be made in program execution speed by using the technique of "pipelining". Note that in figure 22 that during the FETCH time the processor under control is effectively idle since it is waiting for the output of the microprogram memory to become steady. Also during the execute time, the microprogram memory is effectively idle since it is merely holding its output steady for the execution. By inserting a register at the output of the microprogram memory as shown in figure 24, one can overlap or "pipeline" the fetch and execute times. One can see how this works by looking at the timing in figure 25. With the leading edge of the first clock signal, the microprogram counter advances to microinstruction i and after the FETCH time the output of the microprogram memory is presented to the input of the PIPELINE REGISTER. As with the counter, one must wait a set-up time before the clock can be asserted to the register. When the clock does arrive, the microprogram memory output is stored in the PIPELINE REGISTER and after its propagation delay, the microinstruction i is presented at the output of the register so that the execution of that instruction can begin. With this same clock edge, the microprogram counter advances to microinstruction i+1 and the FETCH of this instruction begins. Thus the FETCH of one instruction is done simultaneously with the EXECUTE of the previous instruction. The minimum microcycle time is now determined by the longer of the FETCH or EXECUTE times rather than the sum of them.

Figure 24: Microsequencer with Pipeline Register.

Figure 25: Sequential Control with Program Counter and Pipeline Register.

The circuit shown in figure 24 leads to faster program execution. It has one difficulty with conditional branch instructions however. Consider a conditional branch on the results of instruction i as shown in the timing diagram in figure 26. With the first microcycle, we have the FETCH of microinstruction i and with the second, we have the FETCH of microinstruction i+1 and the EXECUTE of microinstruction i. The results of this instruction are ready to be tested in the third microcycle so clearly microinstruction i+1 should be the conditional branch instruction. At the end of the third cycle we begin the FETCH of the next microinstruction which is either microinstruction i+3 or the microinstruction located at the BRANCH ADDRESS. The problem is: what can the execution unit do during the fourth microcycle? The answer is that it can only do something which does not depend on which path the program has taken after the branch instruction. In most cases, nothing useful can be done by the execution unit during this cycle so that the microinstruction after the branch instruction (microinstruction i+2 in this case) becomes a NO OPERATION (NOP), which is a waste of execution time.
The circuit of the processor can be changed to fix this branching problem without slowing down the program execution as is shown in figure 27. The microprogram counter has been replaced by an incrementer and a register, whose output is routed to an additional input to the address multiplexer, and the CONDITION CODE MULTIPLEXER has been replaced by some combinational logic. The address multiplexer is now routed directly to the address inputs of the microprogram memory. This multiplexer is called the NEXT ADDRESS MULTIPLEXER. An incrementer is a circuit whose output is equal to its input plus one. The new register is called the MICROPROGRAM COUNTER even though it is no longer a counter. In the circuit shown the current microprogram address plus one is stored into the MICROPROGRAM COUNTER with each clock edge. Thus when the MICROPROGRAM COUNTER is selected as the output of the NEXT ADDRESS MULTIPLEXER, one has the CONTINUE instruction in effectively the same way as when we forced a COUNT of the counter in figures 15, 16, 18, 28, and 24. A jump instruction is accomplished by selecting the BRANCH ADDRESS as the output of the NEXT ADDRESS MULTIPLEXER and a conditional branch instruction is accomplished by selecting either the BRANCH ADDRESS or the MICROPROGRAM COUNTER depending on the state of the CONDITION input.

The timing of this circuit for non branching instructions appears to be the same as the previous one as is shown in figure 28. The minimum microcycle time is still determined by the longer of the FETCH or EXECUTE times. But the conditional branch timing is now improved as is shown in figure 29. With the first microcycle in figure 29 we have the FETCH of microinstruction i and with the second we have the FETCH of microinstruction i+1 and the EXECUTE of microinstruction i as before. With the third microcycle we begin to test the results of microinstruction i so clearly microinstruction i+1 is the branch instruction. The condition input selects the next address during the third microcycle so that the FETCH of the next microinstruction begins and ends during this cycle. Thus with the leading edge of the clock of the fourth microcycle, the next microinstruction is stored into the PIPELINE REGISTER and the EXECUTE of this instruction can begin. Thus even with a conditional branch instruction, this circuit for a microprocessor makes efficient use of microcycle time. It is this type of circuit that we will find to be commercially available as a LS1 microcircuit when we study microsequences.
3. AN EXAMPLE: A SIMPLE SCANNER

With the next address instructions we have defined so far it is already possible to look at a practical example. Suppose one had a large set of devices which potentially contain a data point on a given event, but for a given event let us say that only a small fraction of the devices have any valid data. Let us design a controller which would scan the devices for data and store the data into a buffer memory along with the address of the devices with data. Figure 30 shows a block diagram of the proposed scanning set-up. For simplicity let each device be interrogated (or addressed) by a signal sent on a cable which we will call the NEXT signal. If the device had data it would send back the data on a bus along with a response signal which we will call DATA-VALID. If the device did not have data it would send a '0' on the DATA-VALID bus. The NEXT signal would be daisy chained from device to device so that after each device received one NEXT signal it would pass the next NEXT signal to the next device until the system was reset for another scan. The NEXT signal from the last device on the chain is routed back to the scanner where it is called the DONE signal so the scanner knows when to stop.
For the moment we shall ignore all details on how the data is read into the devices, read out of the buffer memory, etc. The scanning processor should have two internal counters, a DEVICE COUNTER (D.C.) to keep track of which device has valid data, if any, and an ADDRESS COUNTER (A.C.) to point to the next buffer memory address to be filled. The sequence of events the scanning processor should follow is shown in the flow chart given in figure 31. When the processor starts, it should first reset the DEVICE COUNTER, reset the ADDRESS COUNTER, and send the first NEXT signal as shown in step 0. In the next step, it can test the DATA VALID signal to see if any data was found in the first device. If it is false, it should then test the DONE signal (step 2). If this signal is also false, it can proceed to the next device by incrementing the DEVICE COUNTER and sending another NEXT signal. The processor continues by going back to step 1, in order to see if any data is found in that device. If data is found, it can write the data along with the contents of the DEVICE COUNTER into the buffer memory (step 4). Then it may increment the ADDRESS COUNTER (step 5) and try the next device by going to step 3. The sequence continues until the DONE signal is detected at step 2, in which case the process goes into a STOP state at step 6.

Figure 32 shows a possible implementation of a microprogrammed processor to perform this task. The microprogram memory contains five fields as shown in figure 33. Let us examine each field in order to understand how the processor works. The first two

fields control the next address of the processor. The first field is the BRANCH ADDRESS field (bits 0-3), which is four bits wide. These four bits are routed from the output of the PIPtLINE REGISTER to the '1' input of the NEXT ADDRESS MUXIPLEXER as shown in figure 32. Whenever the SELECT input of this multiplexer is a logic '1', the BRANCH ADDRESS becomes the address input to the microprogram memory. The next field is the ADDRESS SELECT field which is two bits wide and it controls the CONDITION CODE MUXIPLEXER. The conditions '00' and '11' are inputs to this multiplexer in order that CONTINUE and JUMP
the fields have the following meaning:

- Bits 0-3: BRANCH ADDRESS for Branch or Jump instruction.
- Bits 4-5: CONDITION CODE MULTIPLEXER Control
- Bits 6-8: BUFFER MEMORY Control
- Bits 9-11: DEVICE MEMORY Control
- Bit 12: FP CONTROL

<table>
<thead>
<tr>
<th>Bits</th>
<th>Condition Code</th>
<th>Branch Address</th>
<th>Buffer Memory Control</th>
<th>Device Memory Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9-11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 33: Definition of Simple Scanner Microinstructions.

The next 3 bits (bits 6-8) is the BUFFER MEMORY CONTROL field. Each bit of the field is wired to a control point of the buffer memory; if bit 6 is 1, the ADDRESS COUNTER is reset to zero; if bit 7 is 1, the ADDRESS COUNTER is incremented; and if bit 8 is 1, the contents of the DEVICE COUNTER and the DATA lines are written into the buffer memory.

The next 3 bits (bits 9-11) is the DEVICE CONTROL field. Again each bit of the field is wired to a control point dealing with the devices; if bit 9 is 1, a NEXT signal is sent out; if bit 10 is 1, the DEVICE COUNTER is incremented; and if bit 11 is 1, the DEVICE COUNTER is reset to zero. The last field contains only one bit (bit 12), when it is 1, the processor stops.

The program can now be written to perform the scan operation. Let us go through the flow chart shown in figure 31 again. At step 8, we want to reset the ADDRESS COUNTER so bit 6 should be 1. We don't want to increment this counter or write to the memory so bits 7 and 8 should be 0. Thus the BUFFER MEMORY CONTROL field should be set to '100'. The DEVICE COUNTER should be reset and a NEXT signal sent out, so the DEVICE CONTROL field should be set to '1101'; i.e. bits 9 and 11 should be set to '1' and bit 10 to '0'. The next microprogram address can be the next sequential instruction, so the ADDRESS SELECT field should be set to '00' in order to execute the CONTINUE next address instruction. It doesn't matter what the BRANCH ADDRESS field is set to since they are not used for the CONTINUE instruction, so we set it to '0000'. Thus the contents of the first microprogram location (address 0000) should be as shown in figure 34.

The next step tests the DATA VALID signal and leaves the buffer memory and device control alone. Thus both the BUFFER MEMORY CONTROL and DEVICE CONTROL fields both contain '000'. The ADDRESS SELECT field is set to '01' to route the state of the DATA VALID signal to the NEXT ADDRESS MULTIPLEXER. The BRANCH ADDRESS field is set to the next micro-program address if DATA VALID signal is present which we will set to '0101', i.e. step 4. Thus the contents of microprogram memory at location 0001 should be as shown in figure 34.

At microprogram location 0010, which will be the instruction if the DATA VALID tests fails, we should increment the address counter, send a NEXT signal, and go back to the DATA VALID test at location 1. Thus we set bits 9 and 10 to '1', the ADDRESS SELECT field to '11' and the BRANCH ADDRESS field to '0001' as shown in figure 34.

The remaining steps are programmed in a similar fashion. The whole program is summarized in figure 34. Only six microprogram instructions were needed to program this processor to control the simple scanning operation. Locations 7 through 15 of the microprogram memory are unused.

4. WHY MICROP'ROGRAM

Up to this point, we have studied the basics of microprogramming and a simple example. We are now ready to evaluate whether we should or should not build our logic subsystems using the microprogramming technique. Keep in mind that most logic subsystems have two parts: one is the part under control and the other is the part which generates the timing.
sequences. One can identify these parts in logic systems as small as the simple scanner or as large as the central processing unit of a major computer system.

An advantage in a microprogrammed system is its very clean and orderly structure. The simple scanner described in the previous section is an example. All control points within the processor are controlled by the microprogram memory. All registers, counters etc., can be synchronized by the same clock which makes it much easier to find faults in the circuit. The system is very easy to describe and document.

There are still other advantages, especially for larger, more complex logic systems. One advantage is the ease in which changes can be made. Suppose, for example, that one wanted to change the simple scanner so that it would stop scanning if the buffer memory became full. From the ADDRESS COUNTER one could obtain a signal indicating that the maximum buffer memory address had been attained. This signal could be routed to an additional input to the CONDITION CODE MULTIPLEXER and the program could be changed so that the buffer full condition would be tested after the ADDRESS COUNTER was incremented. Although this change would require changing some circuits such as the CONDITION CODE MULTIPLEXER, the change is quite simple to understand and implement compared to changing a hard wired or random logic design for the simple scanner. The more complex a logic subsystem, the easier it is to change a microprogrammed circuit compared to its random logic counterpart.

It is also much easier to add special features to a microprogrammed logic system. One desirable feature, for example, would be self diagnostic programs in the microprogram memory. These features probably require only additional memory space and a few extra circuits. But for a random logic design, the additional logic required to perform all the diagnostics may be as complex or difficult as the original logic itself.

There will be cases, however, where the random logic design is more suitable for a logic subsystem. The microprogrammed system may be slower, for example. The random logic design avoids the difficulty of programming the ROM. But if one is to use the LSI microcircuits, which we will be studying in the following sections, one must use the microprogramming technique.

5. LSI MICROCIRCUITS

There is an increasing number of LSI microcircuits becoming available. They all fit very well into a microprogrammed architecture. This is clearly becoming the "nouveaux vague" in bipolar technology. Most of these components are intended to be used in building computers and computer peripherals where the speed of bipolar circuits is necessary. In our field of High Energy Physics, we are not in the business of building computers but these microcircuits can nevertheless simplify how we build many of our logic systems.

The circuits can be classified by what part of a computer they are intended to be used. The following list shows some of the computer parts that currently available as LSI microcircuits:

<table>
<thead>
<tr>
<th>Arithmetic/Logical Elements</th>
<th>Microprogram Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bipolar Memory</td>
<td></td>
</tr>
</tbody>
</table>

Interrupt Control
Input/Output Elements
Direct Memory Access Control
Timing Control
Shifting Elements
Status Storage and Multiplexing
Memory Address Control
Program Logic Arrays

For the logic designer in High Energy Physics, one would layout a sketch of the control task he wants to build. Then scan through the semiconductor catalogs to find which circuits have the capabilities of parts of his logic diagram. Frequently, one would find that only a fraction of an LSI circuit is needed to implement his circuit and it would seem wasteful to use. On the other hand, the logic in traditional SSI and MSI may exceed the cost of this LSI circuit and the amount of space required on the circuit board may be reduced by using the LSI microcircuit.

There is clearly not enough time in these lectures to cover all of the above kinds of LSI microcircuits. We will therefore study only two of them, the microprocessor slice and the microsequencer. These two in some ways are the most interesting and the most different from the older SSI and MSI circuits.

5.1 THE BIPOLAR MICROPROCESSOR SLICE

The microprocessor slice is designed to be the principal arithmetic/logic element within the central processing unit of a computer or peripheral controller. The most widely used microcircuit of this type is the 2901A. It was introduced by Advanced Micro Devices in the summer of 1975 and has since been manufactured by most of the bipolar semiconductor manufacturers. Figure 35 shows a block diagram of this circuit. All the data paths shown are four bits wide. To form a processing unit with a larger number of data path bits, the circuit has appropriate inputs and outputs to allowed it to be cascaded with other slices. Thus four such slices can form a unit with 16 bits of data path and eight slices forms a unit with 32 bits of data path.

Let us start studying this microcircuit with its arithmetic logic unit (ALU). An ALU performs arithmetic and logical operations on two inputs, called R and S in figure 35, and produces an output which is called F. The 2901 can perform one of eight functions on the operands R and S. The function is selected by three input signals which form a three bit function code which is part of the microinstruction of the circuit. That is, the user provides a three bit binary number, from 0 through 7, on three input pins of the circuit to control which function is to be performed. There are three arithmetic functions:

- R PLUS S,
- S MINUS R, and
- R MINUS S;

and five logical functions:

- R OR S,
- R AND S,
- NOT R AND S,
- R EXCLUSIVE-OR S, and
- R EXCLUSIVE-NOR S.

The R and S operands are each outputs of separate multiplexers. The R MULTIPLEXER has as inputs the DIRECT DATA inputs (D), the output of the A-LATCH, or
a binary '0'. The D input allows the user to bring in data from outside the circuit, say from memory, so four pins of the circuit is used for this purpose. The significance of the A-LATCH and the usefulness of providing a '0' as one of the inputs will be seen in the following paragraphs. The S-MULTIPLEXER has for its input also a binary '0' and the A-LATCH as well as the output of a B-LATCH and the contents of a Q-REGISTER. With these inputs there are twelve combinations possible for the R and S operands. Twelve is not a nice number, so the manufacturer has chosen eight of these twelve that he feels are the most useful. That is, three bits of the micro-instruction are used to select which combination of the R and S inputs are used as operands to the ALU. These three bits is called the Source Code of the microinstruction.

The Source Codes can be best seen in a matrix of the Source Code versus the Function Code as is shown in figure 36. Note that the entries of this matrix, look very much like computer operations, i.e. A PLUS B, A OR B, D MINUS A, B PLUS 1, etc. It is essentially a sufficient set to allow the user to do any operation on two binary or logical quantities. Note also that some of the elements of this matrix have two entries. The difference between these entries is whether the CARRY-IN to the least significant bit of the ALU is '1' or '0'. For the addition of two numbers the CARRY-IN should be '0', but for subtraction in 2's complement form, the CARRY-IN to the least significant bit should be '1'. Thus we realize that in order to control this microcircuit we must provide every small detail to get what we want. We are truly programming at the microinstruction level.

Let us now return to the A- and B-LATCHES. Within the 2901A there are 16 registers which are organized as a "dual port memory". That is, two addresses can be read from the memory simultaneously. The 16 word memory is called a register file and the user provides a four bit address to select one of 16 words in the register file for each port of the memory. These are called the A and B addresses. The clock signal input to the circuit is used to hold the outputs from these two ports in a special kind of register called a latch. Thus the output of the A- and B-LATCHES are the contents of the register at the A and B addresses when the clock signal makes the transition from low to high.

The are two sets of results from the ALU. The first is called F in the figure 35 and it is the result of the function on the two operands. The other is a set of status conditions indicating if the operation resulted in a carry out of the most significant bit, a 2's complement arithmetic overflow, a zero result, or a negative result. In addition to these status signals the ALU incorporates the standard Carry-Look-Ahead techniques to speed up operations over many slices which requires the generation of the Carry Generate (G) and Carry Propagate (P) signals.
\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\text{Octal \( \text{is}_{4,3} \)} & C_0 = L & R \text{ Plus} & C_0 = H & A + Q & A + B & Q & B & A & D + A & D + Q & D \\
& C_0 = S \text{ Minus} & R & C_0 = H & Q + 1 & A + B + 1 & Q + 1 & B + 1 & A + 1 & D + A + 1 & D + Q + 1 & D + 1 \\
& C_0 = L & R \text{ Plus} & C_0 = H & Q - A & B - A - 1 & Q - 1 & B - 1 & A - 1 & A - D - 1 & Q - D - 1 & D - 1 \\
& C_0 = S \text{ Minus} & R & C_0 = H & A - Q & B - A 0 & Q - 1 & - B - 1 & - A - 1 & D - A - 1 & D - Q - 1 & D - 1 \\
& \text{R EXN, O} & \text{REG} & \text{o} & \text{AV} & \text{q} & \text{AV} & \text{q} & \text{B} & \text{A} & \text{DVA} & \text{DVA} & \text{DVA} \\
& \text{R EXN, R} & \text{REG} & \text{o} & \text{AV} & \text{q} & \text{AV} & \text{q} & \text{B} & \text{A} & \text{DVA} & \text{DVA} & \text{DVA} \\
\end{array} \]

\( + = \text{PLUS}; \quad - = \text{MINUS}; \quad \vee = \text{OR}; \quad \wedge = \text{AND}; \quad \psi = \text{EX-OR}. \)

### Figure 36: Source Operand and ALU Function Matrix.

<table>
<thead>
<tr>
<th>MICRO CODE</th>
<th>RAM FUNCTION</th>
<th>D-REG. FUNCTION</th>
<th>Y OUTPUT</th>
<th>RAM SHIFTER</th>
<th>Q SHIFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_7 )</td>
<td>( b_6 )</td>
<td>( b_5 )</td>
<td>( b_4 )</td>
<td>( b_3 )</td>
<td>( b_2 )</td>
</tr>
<tr>
<td>L L L</td>
<td>0</td>
<td>x</td>
<td>none</td>
<td>none</td>
<td>P = Q</td>
</tr>
<tr>
<td>L L L</td>
<td>1</td>
<td>x</td>
<td>none</td>
<td>none</td>
<td>P = B</td>
</tr>
<tr>
<td>L L L</td>
<td>2</td>
<td>none</td>
<td>P = B</td>
<td>x</td>
<td>none</td>
</tr>
<tr>
<td>L L L</td>
<td>3</td>
<td>none</td>
<td>P = B</td>
<td>x</td>
<td>none</td>
</tr>
<tr>
<td>L L L</td>
<td>4</td>
<td>down</td>
<td>P = B</td>
<td>x</td>
<td>none</td>
</tr>
<tr>
<td>L L L</td>
<td>5</td>
<td>down</td>
<td>P = B</td>
<td>none</td>
<td>x</td>
</tr>
<tr>
<td>L L L</td>
<td>6</td>
<td>up</td>
<td>P = B</td>
<td>x</td>
<td>none</td>
</tr>
<tr>
<td>L L L</td>
<td>7</td>
<td>up</td>
<td>P = B</td>
<td>none</td>
<td>x</td>
</tr>
</tbody>
</table>

### Figure 37: 2901A Destination Codes.

Generally, one would like to do something with the results of the ALU operation. The data paths within the 2901 provide us with many possibilities. First, the results may be written back into the register file. At this point we start running out of pins available on the circuit package, so the manufacturer has made a compromise in that when results are written to the register file they are written into the register of the B address. But before writing into the B address the 2901A has the capability of shifting the results to the right or to the left. It can also write the results to the Q-REGISTER. We can also output the results on the other pins of the package or we can use these output pins to output the contents of the A-REGISTER. Of all the possibilities the manufacturer has again used 3 pins on the package to allow us to select one of eight possibilities. These 3 bits are called the Destination Code of the microinstruction and these codes are shown in figure 37. Note that one of the Destination Codes doesn't write the results anywhere (Code 0). It is a NO-Operation code (NOP) and is useful when we want to do a COMPARE operation without destroying the contents of any register. Note also that there are Destination Codes where the Q-REGISTER is shifted while shifting the results written into the register file. The purpose of this code is to allow the user to program the circuit to do multiplication, division, and double length shifts and rotates.

In summary, a microprocessor slice can form the core of a central processor unit within a computer. It is controlled by providing it with a micro-instruction which it uses internally. In the case of the 2901A, the microinstruction must be at least 18 bits in length: 3 bits for the Source Code, 3 bits for the Function Code, 3 bits for the Destination Code, 1 bit for the CARRY-IN in the least significant bit and four bits each for the A and B addresses. Although the 2901A is the most widely used microprocessor slice today, there are others on the market which may be more suitable in certain applications. Table 1 lists all the microprocessor slices currently available with a few comments on their individual features. For more details, the reader is referred to the specification data sheets from the various manufacturers, or to some of the numerous articles in some of the journals [1,2].

### 5.2 An Example with the 2901A

An example of the use of the 2901A is a microprocessor called the 168/E which was developed by my colleagues and me at S.L.A.C. Figure 38 is a block diagram of the processor. As with other designs using it, the 2981 forms the core of the data processing and there are circuits around it to form a complete microprocessor. For the 168/E, the choice of the circuits around the 2981 was based on what would be easy to program. For this purpose some assembly code written for the IBM 370 computer was used as a model for the kind of operations that would
### Table 1
List of Microprocessor Slices Currently Available.

<table>
<thead>
<tr>
<th>Part no.</th>
<th>Originated by</th>
<th>Second Source</th>
<th>(bits) Tech.</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2901A</td>
<td>Advanced Micro Devices</td>
<td>Fairchild, Monolithic Memories, National, Raytheon, Motorola, and Signetics.</td>
<td>4 S-TTL</td>
<td>most popular slice, dual-port architecture</td>
</tr>
<tr>
<td>2903</td>
<td>Advanced Micro Devices</td>
<td>National.</td>
<td>4 S-TTL</td>
<td>Improved version on 2901A, expandable register file, built-in multiply, divide, etc.</td>
</tr>
<tr>
<td>6701</td>
<td>Monolithic Memories</td>
<td>ITT</td>
<td>4 S-TTL</td>
<td>similar to 2901A</td>
</tr>
<tr>
<td>3002</td>
<td>Intel</td>
<td>Signetics</td>
<td>2 S-TTL</td>
<td>accumulator orientated</td>
</tr>
<tr>
<td>9405</td>
<td>Fairchild</td>
<td>Signetics</td>
<td>4 S-TTL</td>
<td>smaller package</td>
</tr>
<tr>
<td>74S481</td>
<td>Texas Inst.</td>
<td>none</td>
<td>4 S-TTL</td>
<td>similar to 2903, but has no register file</td>
</tr>
<tr>
<td>10800</td>
<td>Motorola</td>
<td>Fairchild</td>
<td>4 ECL</td>
<td>has no register file</td>
</tr>
<tr>
<td>SBP0400</td>
<td>Texas Inst.</td>
<td>none</td>
<td>4 TTL</td>
<td>very slow</td>
</tr>
</tbody>
</table>

---

![Figure 38: Block Diagram of 168/E Microprocessor.](image-url)
be necessary to have a useful processor. An example of a typical DO LOOP found in many of the programs we wished the microprocessor to be able to handle is shown below.

```
LOOP C 0,ED(9,10)
BL GOTE
SR 9,1
BM LOOP
```

Let us consider how this piece of code is executed on the 168/E in order to understand the functions of the circuits around the 2901A microprocessor slices. The first IBM instruction is a memory to register comparison. The memory address is calculated as a sum of the contents of registers 9 and 10 plus the contents of a 12 bit displacement field (ED) which is part of the instruction. The IBM 370 has 16 registers and so does the 2901A. Thus a one to one identification of the registers in the 370 with those of the 168/E was made. We have already studied the 2901A well enough to see how one can add the contents of two registers together and output the sum. This would be the next step, the instruction at hand does not in order to follow the example of the IBM 370 program. That is, a 168/E microinstruction with the 2901A Source Code 1, Function Code 0, and Destination Code 1 as can be seen from figures 36 and 37.

The next step would be to add to this sum the 12 bits of the displacement field. The 168/E performs this operation in a separate microinstruction with an additional adder circuit. In this microinstruction, 12 bits of the microprogram memory are routed to one input to the adder. The other half of the adder is connected to the output of the 2901A. The sum of this add instruction is stored into a MEMORY ADDRESSE REGISTER whose output is connected to the address inputs of the data memory. The access time of the data memory is fast enough for the data memory output to be strobed into the D-REGISTER at the end of the microcycle.

The last step of the COMPARE instruction is to make the comparison between memory, which has now been strobed into the D-REGISTER of the 168/E, and the contents of register 0. This step is performed by another microinstruction in which the instruction for the 2901A uses its D input as one of the ALU source operands, i.e. Source Code 5, Function Code 1, and Destination Code 1.

The next IBM 370 instruction is a conditional branch in which the next instruction is to be taken from an address labeled 'GOTE', if the result of the comparison was negative. Thus to the 168/E micro-instructions we have already defined, we must add conditional branch microinstructions. The 168/E uses a binary counter to control the next microinstruction instruction address. As in figure 20, the flow of the program execution can be altered by asserting a signal on the LOAD pin of the counter circuit. To control the conditions with which one wants to branch, the IBM 370 is again used as a model. The conditional branch instruction of the IBM 370 has a 4 bit field called the mask which specifies what conditions are required to force a branch. The conditions after arithmetic operations are a zero result, a negative result, a positive result, and an arithmetic overflow result. The arithmetic instructions sets one of these conditions to be true after the operation. If the condition which was set matches with a set bit in the mask of the conditional branch instruction, then the branch is taken, otherwise the next sequential instruction is executed.

The status outputs of the IBM 370 do not exactly correspond to those of the IBM 370 but with a few logic circuits one can produce identical codes. Thus the 168/E conditional branch microinstruction has been set up with the same 4 bit mask as the IBM run. If there is a match between the 4 bit mask and the modified 2901A status bits then the program counter is put into the LOAD state. Fifteen bits from the microprogram memory contain the branch address which is routed to the parallel load inputs of the program counter.

The rest of the instructions of the DO LOOP can be emulated by the 168/E with the microinstructions already described. The structure of the 168/E is typical of structures in which the 2901A forms the core of data processing. We can identify the parts which are under control and the part which provides instructions. The IBM 370 instructions, one sees that an impressive number of the instructions can be exactly emulated by the 168/E. These instructions turn out to be about the same subset of the instructions that the IBM FORTRAN IV compiler generates when dealing with 2 or 4 byte logical or 4 byte integer variables. The Floating Point instructions are executed in a separate processing unit but not shown in figure 38. Thus the 168/E microprocessor can be programmed in FORTRAN by using the IBM FORTRAN compiler to generate machine instructions, then using a program which runs on a IBM computer to translate these machine instructions into the microinstructions of the 168/E. This is possible because the 2901A has the same number of registers as the IBM 368/370 and 168 can perform all the integer arithmetic and logical operations of the IBM 368/370. (In fact, it can do some operations that the IBM computer can not).

Also, the circuitry around the 2901A makes a processor with the same type of addressing and conditional branching. Thus either the 2901A with some circuitry around it forms a very powerful microprocessor or the IBM 368/370 is a very simple computer depending on your point of view.

We have already mentioned speed of execution as one of the frequent requirements for logic systems in High Energy Physics. So one can ask what is the speed of the 168/E? The 168/E was implemented with low power Schottky circuits and a typical gate propagation delay of 5 nsec, yet the speed of execution of a program is only between 1.3 and 1.8 times slower than the IBM 370/168. Compared with typical minicomputers, the 168/E speed is about 3 to 10 times faster. And what about the cost? The main cost of the 168/E processor is the eight 2901As which is about 150 US$. The other circuits, circuit board, sockets, and power supply add less than another 300 US$. Thus the user of these LSI microcircuits can build for himself very powerful and fast processors with standard "off the shelf" components at a price he can afford.

5.3 THE MICROPROGRAM CONTROLLER CIRCUIT

Let us now take a look at another LSI microcircuit. Another important circuit which has recently become available is the microprogram sequencer. These circuits are designed to serve as the next address control of the microprogram memory. They incorporate most of the circuitry we discussed in section 2.2. Essentially every bipolar semiconductor manufacturer has a circuit of this type.
### Table 2
Partial List of IBM 360/370 Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>REG</th>
<th>HW</th>
<th>FW</th>
<th>MUL</th>
<th>IMD</th>
<th>CHR</th>
<th>DEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>LR</td>
<td>LH</td>
<td>L</td>
<td>LM</td>
<td>LA</td>
<td>IC#</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>AR</td>
<td>STH</td>
<td>ST</td>
<td>STM</td>
<td>STC#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>LR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>CR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td>CLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>MR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>DIVIDE</td>
<td>DE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>OR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>ER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>SHIFTS</td>
<td>SRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SKR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Branching</td>
<td>BALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Floating</td>
<td>LER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LCR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LPR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>AER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ADR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>SER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td>SER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>MER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>DIVIDE</td>
<td>HER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>HALF</td>
<td>HER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

*Type of Second Data Operand:

- REG: Register (4 byte)
- HW: Half Word (2 Byte)
- FW: Full Word (4 Byte)
- MUL: Multiple Words (4 or more Bytes)
- IMD: Immediate (Operand from Instruction Word)
- CHR: Character (1 Byte)
- DEC: Decimal

# Not implemented in 168/E

* Implemented with optional Floating Point Processor

**REAL#6 rather than REAL#4

It is interesting to study one of them in detail in order to both have a basic understanding of their features and to illustrate some of the techniques used in microprogramming. A list of the currently available microsequencers is shown in Table 3 . Unlike the case of the microprocessor slice, none of these circuits seems to have taken a clear lead in popularity. As an example of a microsequencer to study, I have chosen the newest and probably the most interesting one: the 2918.

Figure 39 is a block diagram of the AM2918 made by Advanced Micro Devices. From this figure one recognizes the same basic structure that we used in the figure 27; i.e., the NEXT ADDRESS MULTIPLEXER, the incrementer, the MICROPROGRAM COUNTER register, and the condition code input (CC). To the basic structure of figure 27 some additional features have been added in order to make the circuit of more general utility. The NEXT ADDRESS MULTIPLEXER, for example, has two additional inputs: one from a 5 word last-in-first-out program counter STACK and the other from a register which can also be used as a counter.

The D input to the multiplexer come directly from the data inputs pins on the chip. They are intended to be used for both the BRANCH ADDRESS field of the PIPELINE REGISTER and the output of the MAPPING ROM as is shown in figure 40 . In order to choose between these two possibilities the 2918 provides two outputs, PL and MAP which enable the outputs of the PIPELINE REGISTER or the MAPPING ROM respectively. An additional OUTPUT-ENABLE signal is available for another register and/or ROM called VECT. The 2918 will generate a signal on only one of the OUTPUT-ENABLE (OE) signals at a time, thus the three sources for the D inputs are effectively multiplexed. In the 2918 the NEXT ADDRESS MULTIPLEXER is really a six input circuit, with 3 internal and 3 external sources.

Let us consider each of the NEXT ADDRESS MULTIPLEXER's inputs. First the MICROPROGRAM COUNTER input is identical to the one we studied in figure 27 . The Carry-In (CI) should be set to 'l' to make the incrementor add one to the current microprogram address.

A new feature is the R REGISTER/COUNTER. It has several uses which illustrate some of the other techniques one can use with the microprogramming. As a register, it is an auxiliary storage location to the BRANCH ADDRESS field of the program memory PIPELINE REGISTER. It is be loaded from the Direct
TABLE 3
List of Available Microprogram Sequencers.

<table>
<thead>
<tr>
<th>Part no.</th>
<th>Originated by</th>
<th>Second Source</th>
<th>Bits (Tech.)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2909</td>
<td>Advanced Micro Devices.</td>
<td>Raytheon, National.</td>
<td>4 S-TTL Slice</td>
<td></td>
</tr>
<tr>
<td>2911</td>
<td>Advanced Micro Devices.</td>
<td>Raytheon, National.</td>
<td>4 S-TTL Slice, similar to above with additional features</td>
<td></td>
</tr>
<tr>
<td>2910</td>
<td>Advanced Micro Devices.</td>
<td>none</td>
<td>11 S-TTL</td>
<td>With Condition Code logic</td>
</tr>
<tr>
<td>3001</td>
<td>Intel</td>
<td>Signetics</td>
<td>9 S-TTL</td>
<td>Complex but saves memory space</td>
</tr>
<tr>
<td>67110</td>
<td>Monolithic Memories</td>
<td>none</td>
<td>9 S-TTL With ALU shift matrix</td>
<td></td>
</tr>
<tr>
<td>74S482</td>
<td>Texas Inst.</td>
<td>none</td>
<td>4 S-TTL Slice, simple</td>
<td></td>
</tr>
<tr>
<td>8X02</td>
<td>Signetics.</td>
<td>none</td>
<td>10 S-TTL Very simple</td>
<td></td>
</tr>
<tr>
<td>9406</td>
<td>Fairchild</td>
<td>none</td>
<td>4 S-TTL Slice</td>
<td></td>
</tr>
<tr>
<td>9408</td>
<td>Fairchild</td>
<td>none</td>
<td>10 T1L Condition Code Register</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 39: Block Diagram of 2910 Microsequencer](image)

Data (D) inputs whenever the LOAD signal is received. At a later time in the program one could execute a conditional two way branch or subroutine CALL to either the D input or the R REGISTER.

The R REGISTER can also be used as a counter. This allows one to repeat an instruction or a series of instructions in the following way. The counter is initially loaded with a value. During certain microinstructions the counter is decremented by one. When the value of the counter reaches zero, a ZERO DETECT signal is generated. Other microinstructions can use this signal in place or in conjunction with the CC input of the circuit in order to select the next address. Thus when the counter has been decremented to zero one can have an instruction take an alternative branch to the normal one. This structure may be used for example in certain iterative instructions such as multiplication and division.

As with ordinary programming, there are advantages in using subroutines for certain sections of the program so that they need not repeated in the memory as many times as they are used. In order to make a subroutine CALL one needs to add two capabilities to the hardware structure we have already studied. First, when we make a BRANCH to the subroutine, we must have the capability of storing the address to which we should return after the subroutine execution is completed, and second, we must be able to return to that stored address.
The 5 word STACK in the 2910 with its connection to the MICROPROGRAM COUNTER and to the NEXT ADDRESS MULTIPLEXER gives us both capabilities. It is used in conjunction with the STACK POINTER which is a up/down counter that always points to the last data entered into the STACK file. When the counter is incremented it is called a PUSH and conversely when it is decremented it is called a POP. A subroutine CALL is executed in the following way. The subroutine address is selected as the next microprogram address from either the D input or the R REGISTER. The MICROPROGRAM COUNTER will thus be address of the subroutine CALL plus 1. The STACK POINTER is first pushed and then the MICROPROGRAM COUNTER is stored into the top of the STACK. The next cycle will be from the first location of the subroutine. The subroutine RETURN is executed by selecting with the NEXT ADDRESS MULTIPLEXER the output of the STACK. Thus the next instruction to be executed will be one instruction beyond the instruction which made the subroutine CALL. At the end of the RETURN cycle the STACK is popped to complete the linkage. Since the STACK contains 5 words, the subroutine CALLS can go to 5 deep; beyond that the highest level subroutine return address will be lost.

The circuit has a number of parts which need to be controlled. The output of the NEXT ADDRESS MULTIPLEXER must be selected from one of the four inputs or forced to zero; the STACK must be PUSHed, POPped, HELD, or ZEROed; the R REGISTER must be LOADED, DECREMENTED, or HELD; and one of the OUTPUT ENABLES may generated. Of all the combinations possible, the microprogram controller instruction. In most applications, four bits of output from the microprogram memory are used to provide this instruction in the same way that bits 4 and 5 were used in the simple scanner example. Some of the instructions use the CC input for conditional branching. In when the condition is 'true' it is called the PASS state and when it is 'false', it is called the FAIL state. In addition the CONDITION ENABLE (CCCON) input can be used to force the internal condition code (TEST) to PASS. Finally, the LOAD input of the R REGISTER can be independently controlled. Thus we have really 6 bits of instruction input, although some of the 64 combinations are redundant.

We will now go through all 16 of the 2910's microinstructions. This exercise will serve to illustrate the special techniques one can use in microprogramming. It is also rather interesting and fun. For each instruction we shall consider the state of the TEST input and the contents of the R REGISTER/ COUNTER since they may alter the resultant operation of the microinstruction. When their states do not affect the operation, it is called a "Don't Care" condition which is indicated by an "X" in the figures that are to follow. The microinstruction may affect the contents or status of the STACK, next address source, the R REGISTER/COUNTER, and/or the OUTPUT ENABLES. If the operation does not affect any one of these then it is called a "No Change" condition which is indicated by "NC" in the figures. As we study the 2910's microinstructions, one can try to imagine an analogy with FORTRAN statements that control the program flow.

5.3.1 Continue.

Instruction 14 is a Continue (CONT), which is the simplest instruction. The next address source is always the contents of the MICROPROGRAM COUNTER. One should recall that the MICROPROGRAM COUNTER is always the current address output of the 2910 plus one. As shown in figure 41, the status of the TEST input and R REGISTER don't influence the operation, the STACK and R REGISTER don't change their value and the PIPELINE REGISTER is enabled. The Continue instruction is probably the most frequently used instruction since it is used when a series of microinstructions are executed.

5.3.2 Jump Map.

Instruction 2 is an unconditional branch instruction in which the Mapping ROM OUTPUT ENABLE is turned on. It is called a JUMP MAP (JMAP). The
status of the TEST input and the R REGISTER are Don't Care. The next address source is always taken from the D input. In the example given in figure 42, microinstruction 53 has the JMP instruction. When it appears in the PIPELINE REGISTER, the MAPPING ROM is enabled and its output is routed through the 2910 to the address input of the microprogram memory. If the contents of the MAPPING ROM were 90, then the program flow would jump from 53 to 90 as shown. In FORTRAN the JMP instruction is analogous to the GO TO statement.

5.3.3 Conditional Jump Pipeline.

Instruction 3 is a conditional branch instruction in which the PIPELINE REGISTER OUTPUT-ENABLE is turned on. It is called Conditional Jump Pipeline (CJP) and it is illustrated in the example given in figure 43. If the status of the TEST input is Fail, then the next address source is taken from the MICROPROGRAM COUNTER. So in the example, the program flow would be from instruction 52 to instruction 53. On the other hand, if the status of the TEST input is Pass, then the next address source is the D inputs. Thus the program flow in the example goes from instruction 52 to instruction 30. The contents of the R REGISTER are Don't Care and the STACK and R REGISTER are unaffected. One should recall that one can use the Condition Code Enable (CCEN) input to force the internal condition to Pass. Doing this changes the CJP instruction into a unconditional jump to the contents of the PIPELINE REGISTER. The CJP instruction corresponds to the FORTRAN statement "IF(...) GO TO".

5.3.4 Conditional Jump Vector.

An almost identical instruction is instruction 6 which is illustrated in figure 44. The only difference is that the VECT OUTPUT-ENABLE is turned on instead of the PIPELINE OUTPUT-ENABLE.

5.3.5 Jump Zero.

A very special instruction is instruction 0. In this instruction the output of to 2910 is forced to a binary zero, thus it is called the Jump Zero (JZ) instruction. In the same instruction the STACK is cleared and the PIPELINE REGISTER is enabled. The intention behind this instruction is to put the microsequencer into a well defined state when the power is first turned on. It is easy for the user to add circuits so that on power up the microinstruction 0 is issued to the 2910. Figure 45 illustrates this instruction.
5.3.6 Conditional Jump R/PL.

Instruction 7 is the first example which uses the R REGISTER. It is a Conditional Jump R or PIPELINE REGISTER (JRP) and is illustrated in figure 46. When the TEST input is PASS, the next address source is from the D inputs with the PIPELINE REGISTER enabled. When the TEST input is FAIL, the next address source is from the contents of the R REGISTER. One should recall that the R REGISTER may be loaded in any instruction by generating the LOAD signal. This instruction is effectively a two way Jump, since the next sequential address is never the next address source. In FORTRAN it would correspond to two statements: an "IF(...) GO TO" followed by "GO TO". In a microprogram with the 2910, the two way branch is only one instruction.

7 COND JUMP R/PL (JRP)

5.3.7 Conditional Jump Subroutine Pipeline.

Subroutine CALLs can be made with instruction 1. As shown in figure 47, the Conditional Jump Subroutine (CJS) instruction is actually a conditional subroutine CALL. If the TEST input is FAIL, the next address source is the contents of the MICROPROGRAM COUNTER which is the next sequential instruction. If TEST input is PASS, then the next address source is the D input with the PIPELINE REGISTER enabled. The STACK COUNTER is PUSHED and the current contents of the MICROPROGRAM COUNTER are stored in the STACK, thus saving the address to which the subroutine return should be made. The CJS instruction can be modified to a unconditional subroutine jump by using the CCEN input to force the

5.3.8 Conditional Return.

The return from subroutine is executed by instruction 10. As shown in figure 48, it is also a conditional instruction. If the TEST input is FAIL, the next address source is taken from the MICROPROGRAM COUNTER with no other change. If the TEST input is PASS, then the next address source is the contents of the top of the STACK and at the end of the microcycle the STACK POINTER is POOped. Again this instruction can be modified to a unconditional return by using the CCEN input. The FORTRAN equivalent would be "IF(...) RETURN".

5.3.9 Conditional Jump Subroutine R/PL.

Another method for making subroutine CALLs is the Conditional Jump Subroutine Register/Pipeline (JSP) as shown in figure 49. If the TEST input is FAIL, then the next address source is taken from the contents of the R REGISTER. If the TEST input is
PASS, then the next address source is taken from the D inputs with the PIPELINE REGISTER enabled. In either case the STACK POINTER is pushed and the contents of the MICROPROGRAM COUNTER is stored at the top of the STACK. Thus the TEST input determines which subroutine is called and not whether one CALLS a subroutine or not. The FORTRAN equivalent is somewhat more complex then the ones we have seen so far. It might be written as "if (...) CALL X" followed by "if (not (...)) CALL Y". In some other high level programming languages this microinstruction might be expressed as a "if (...) THEN CALL X ELSE CALL Y". Again we see that in the microprogram it is only one instruction.

5.3.10 Load Counter and Continue.

The "Load Counter and Continue" (LDCNT) instruction provides an alternate method of loading the R REGISTER. As shown in figure 50, the next address source is always the MICROPROGRAM COUNTER just like the Continue instruction. The R REGISTER is loaded from the D inputs with the PIPELINE REGISTER enabled. Many microprocessors could use this instruction as the only method of loading the R REGISTER thus eliminating the need to control separately the LDAD input to the 2910. In FORTRAN, this microinstruction might be equivalent to setting the end point of a DO LOOP as will be seen below.

5.3.11 Repeat Pipeline Counter Not Equal to Zero

The next instruction is the first example of using the R REGISTER as a counter. It is called "Repeat Pipeline Counter Not Equal to Zero" (RPCT). If the contents of the R REGISTER are not equal to zero then the next address source is taken from the D inputs with the PIPELINE REGISTER enabled. At the end of the cycle, the R REGISTER is also decremented. If the contents of the R REGISTER is zero, then the next address source is taken from the MICROPROGRAM COUNTER and the R REGISTER is left unchanged. As illustrated in figure 51, the RPCT instruction can be used to force execution of the same microinstruction many times by letting the contents of the PIPELINE REGISTER be equal to the address of the instruction. This may be used, for example, to do iterative multiplication or division microinstructions. The combination of 2910 microinstructions 12 and 9 look very much like the FORTRAN statements

\[
\text{DO 10 I=1,N}
\]

(One or more statements)

10 CONTINUE

5.3.12 Push/Conditional Load Counter.

Another instruction which loads the R REGISTER/COUNTER is shown in figure 52. It is in fact a conditional load of the counter and it is called "Push and Conditional Load Counter" (PUSH). If the TEST input is PASS, then the R REGISTER/COUNTER is not loaded while if it is PASS then it is loaded from the D inputs with the PIPELINE REGISTER enabled. In either case the next address source is from the MICROPROGRAM COUNTER, the STACK COUNTER is pushed and the MICROPROGRAM COUNTER is stored at the top of the STACK. The purpose of this instruction will not be clear until we study the next and last 4 microinstructions.
4 PUSH/COND LD CNTR (PUSH)

CONT 50
CONT 51
PUSH 52
CONT 53

STACK
REGISTER/ COUNTER

<table>
<thead>
<tr>
<th>TEST</th>
<th>REG/CNTR</th>
<th>STACK</th>
<th>ADDRESS SOURCE</th>
<th>REG/CNTR</th>
<th>OE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS</td>
<td>X</td>
<td>PUSH</td>
<td>PC</td>
<td>LOAD</td>
<td>NC PL</td>
</tr>
<tr>
<td>FAIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 52: 2910 Instruction 4.

5.3.13 Repeat Loop, Counter Not Equal 0.

The next instruction works with the PUSH to perform a microprogram DO-LOOP as is shown in figure 53. It is called "Repeat Loop for Counter not equal to Zero" (RFCT). The instruction is a conditional jump using the contents of the R REGISTER/COUNTER as the TEST input. If the contents are not equal to zero, then the next address source is taken from the top of the STACK and the R COUNTER is decremented. In other words, the program branches back to the beginning of the loop. When the contents of the COUNTER becomes zero, then the next address source is taken from the MICROPROGRAM COUNTER and the STACK is POPed while the COUNTER is left unchanged. In other words, the program drops through the bottom of the loop. Thus we see that the R REGISTER/COUNTER is used like the running index of the DO LOOP. The STACK is used in this case to save the beginning of the loop rather than for saving the subroutine return address. In fact the STACK can be used as a combination of both up to 5 levels of loops and subroutines. The combination of the PUSH and RFCT microinstructions looks very much like the FORTRAN statements:

DO 10 I=1,N
(one or more statements)
10 CONTINUE

8 REPEAT LOOP, CNTR#0 (RFCT)

PUSH 50
CONT 51
CONT 52
CONT 53
RFCT 54
CONT 55

STACK
REGISTER/ COUNTER

<table>
<thead>
<tr>
<th>TEST</th>
<th>REG/CNTR</th>
<th>STACK</th>
<th>ADDRESS SOURCE</th>
<th>REG/CNTR</th>
<th>OE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS</td>
<td>X</td>
<td>POP</td>
<td>PC</td>
<td>NC STACK</td>
<td>NC PL</td>
</tr>
<tr>
<td>FAIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 53: 2910 Instruction 8.

5.3.14 Test End of Loop.

Another example of looping is an instruction called "Test End of Loop" (LOOP). It operates the same way as the RFCT instruction except that the Condition Code input is used as the TEST input rather than the contents of the R REGISTER/COUNTER and the counter is not affected. Note that in the example shown in figure 54 if one never got a TEST input PASS status one would have an infinite loop. Note also that although the PUSH instruction was used at instruction 51 in order to save the beginning address of the loop, the R REGISTER/COUNTER is not used in the loop. In FORTRAN, the LOOP microinstruction looks like a simple "IF(...) GO TO".

13 TEST END LOOP (LOOP)

CONT 50
CONT 51
CONT 52
CONT 53
CONT 54
CONT 55
LOOP 56
CONT 57

STACK
REGISTER/ COUNTER

<table>
<thead>
<tr>
<th>TEST</th>
<th>REG/CNTR</th>
<th>STACK</th>
<th>ADDRESS SOURCE</th>
<th>REG/CNTR</th>
<th>OE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS</td>
<td>X</td>
<td>POP</td>
<td>PC</td>
<td>NC STACK</td>
<td>NC PL</td>
</tr>
<tr>
<td>FAIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 54: 2910 Instruction 13.

5.3.15 Conditional Jump PL and POP.

Each PUSH of the STACK must be followed somewhere by a POP in order to not to lose the subroutine linkage. Instruction 11 has been designed to enable one to conditionally jump out of a loop and restore the STACK at the same time. It is called the "Conditional Jump Pipeline and POP" (CJPP) and it is illustrated in figure 55. If the TEST input is FAIL then the next address source is the MICROPROGRAM COUNTER and the STACK is left unchanged. If the TEST input is PASS, then the next address source is taken from the D input with the PIPELINE REGISTER enabled and at the same time the STACK is POPed.

5.3.16 Three-Way Branch

The next and last instruction is the most complex of all. It uses both the TEST input and the contents of the R COUNTER to determine one of three next address sources. It is appropriately called "Three Way Branch" (TMB). It is also used with the PUSH instruction as shown in figure 56. As long as the TEST input is FAIL, the instruction operates like the RFCT, that is, it the microprogram branches back to the address contained at the top of the STACK as long as the R COUNTER is non-zero. When the R COUNTER reaches zero, however, the next address source is taken from the D inputs with the PIPELINE REGISTER enabled. If the TEST input is PASS, then the program drops out of the loop by taking the next address source from the MICROPROGRAM COUNTER and the STACK POINTER is POPed. In this case the R COUNTER is decremented or unchanged depending on its value.
This strange instruction turns out to be quite useful. If in a loop one were searching for a data point in memory, for example, then the loop could end when either the data point is found (TEST input becomes PASS) or by reaching a certain limit (REGISTER becomes zero). Note that the in the two ending conditions the program goes to two different locations. Thus when compared to a FORTRAN program, this instruction is like having an "IF(...)GO TO" statement as the last statement in a DO-LOOP.

5.3.17 Summary of 2910.

This completes the study of the microinstruction of the 2910. To the FORTRAN programmer these instructions should not seem too strange at all. There is a big difference, however, in the manner in which the instructions are executed. With a FORTRAN program running on a normal computer the compiler has generated various machine instructions to get the desired program flow. With the microprogram sequencer, the program flow is controlled with one microinstruction. Hence we see that microsequencers are designed to make microprogram fast and efficient is memory space by minimizing the number of instruction steps to control the program flow. One must remember that besides the PIPELINE REGISTER bits which control the microsequencer, there other bits which control that which is being controlled. The sequencer does not do useful data manipulation itself.

REFERENCES


INTRODUCTION

Modern Large Scale Integration (LSI) microcircuits are meant to be programmed in order to control the function that they perform. In reference[1], I have already discussed the basics of microprogramming and have studied in some detail two types of new microcircuits. In this course, I will explore the methods of developing software for these microcircuits. This generally requires a package of support software in order to assemble the microprogram, and also some amount of support software to test the microprograms and to test the microprogrammed circuit itself.

1. MICROPROGRAM ASSEMBLERS.

1.1 ASSEMBLERS IN GENERAL.

An assembler is a support software program which allows the programmer to write a program in a symbolic language. It does many tasks for the programmer which greatly relieves him from tedious work of writing programs. These tasks are illustrated in an example of assembly language program given in figure 1 which comes from the output of the IBM 368/370 Assembler. Of interest to us are the columns labeled 'LOC', 'OBJECT CODE', and 'SOURCE STATEMENT'. Under the column labeled 'LOC' is the relative address of an IBM 368/370 instruction which is represented in hexadecimal format under the column labeled 'OBJECT CODE'. The symbolic program is presented under the column labeled 'SOURCE STATEMENT'.

The first task of the assembler is to convert the symbolic operation codes into the machine binary code. For example, in figure 1 the operation code 'SR' was converted into the machine code '1B' and placed in the proper field of the machine instruction as shown at 1. A second task is to substitute for symbolic variable names the machine binary form. One can see at 1 in figure 1 that the symbols '3,2' have been converted to a binary form and placed in the proper fields of the machine instruction for the source and destination registers. Also, at point 2 of the figure the symbolic variable name 'ZAS(7)' has been substituted with the proper form of memory addressing. The third task is to substitute symbolic addresses with the binary addresses. In figure 1 at point 3, the instruction at location '17A' was given the symbolic label 'A22'. All references to this location made by the program used the symbolic name such as the one at location '19C'. The assembler substitutes for the symbolic name the actual address. This was also the case for the address of the symbolic variable given at point 2. The fourth task for the assembler is to supply after all the conversions and substitutions a complete binary program that can be loaded into the processor. In the case of the IBM assembler, this machine code is called the Object Code.

With the aid of the assembler, the programmer can write programs of great length and complexity which would be too difficult to write directly in machine code. Instructions can be inserted or moved without difficulty when all variables and branch addresses are referred to symbolically because the assembler will do the work of calculating the real addresses in generating the Object Code. The programmer is also almost completely relieved of having to know about the placement of the fields in the machine instruction and the details of addressing memory.

The assembler is generally given by or bought from the manufacturer of the computer when one receives the computer as part of a package of support software. It is written expressly for the computer. If one is to supply an assembler for a microprocessor of one's own design, then one has the problem of having an assembler written for that machine. Methods of doing this will be discussed in the next sections.

1.2 MICROPROCESSOR ASSEMBLERS IN GENERAL.

An assembler written expressly for a microprocessor should have the same general features as an assembler for a computer. One should realize, however, that there may be some major differences between the instruction set of a microprocessor and a that of a computer. Take for example the question of the operation codes. A computer typically has a set of about 100 to 400 instructions. Each instruction may have a few parameters, and these parameters are specified in a way which is common to many instructions. For example, the 'Subtract Register' (SR) instruction of the IBM 368/370 discussed above had two parameters, the source and destination registers. An 'Add Register' instruction has the same two parameters and they are specified in the same way.

A microprocessor might have an several orders of magnitude more operation codes if we tried to define them in the same way as computer operation codes. For example, a processor with a 2401A bit slice microprocessor requires that the source, function and destination fields be specified as was shown in figures 36 and 37 of reference [1]. Since each of these fields is 3 bits in length, there would be 512 combinations which could be considered operation codes. And since the CARRY-IN to the least significant bit also needs to be specified, we find that there are really 1024 combinations. The A and S addresses, of course, may be considered as operands, as were the source and destination registers with the IBM 368/370.

Besides the fields of the microinstruction defining the operation of the 2901A, we may find a microsequencer in the processor. The 2910 microsequencer, for example, has a 5 bit microinstruction

* Work supported by the U.S. Department of Energy
code if we include the condition code enable bit along with the next four address instruction bits. If the next address control circuit were as shown in figure 48 of reference [1], then we must also consider the four bits which control the condition code enable bit. All the combinations of next address control must be joined with the 2981 instruction which would lead to a total of 524,288 operation codes.

Thus, it is general practice for microprocessor assemblers to divide the microinstruction into several fields each with its own set of operation codes and operands. Whereas an ordinary computer assembler generates one machine instruction for one operation code, the microassembler may expect several operation codes to be concatenated into one microinstruction.

Another difference between ordinary computer instructions and microinstructions is the appearance of don't Care fields in the microinstruction. In the case of the simple scanner described in section 3 of reference [1], the branch address field was not used for instructions in which the next address was taken as the next sequential address (CONTINUE instructions). As we will see below, it is also convenient to have "Default" fields in the microinstruction. A microassembler must be able to handle these situations as well as the multiple operation codes.

2. GENERATION OF MICROPROGRAMS

There are many methods for generation of microprograms and we will study four of them. The choice of which method to use depends on many factors, such as the length of the expected programs, the execution efficiency required, the complexity of the instruction set, etc. And as with programming computers, a programming language that generally leads to efficient code such as assembly language may be rejected in favor of a programming language which is easier to use by a average programmer, such as FORTRAN.

2.1 HAND-CODED BINARY

Hand-Coded binary programming is a method in which the programmer writes directly in the binary bit pattern of the processor's microinstruction set. This method was used in programming the simple scanner in section 3 of reference [1]. Figure 33 of reference [1] gave the instruction set of the processor. The program to perform a scan of devices with data was written in only six instructions as was shown in figure 34 of that reference.

The Hand-Coded binary method was a perfectly viable method in the case of the simple scanner. Unlike the methods which will be discussed in the following sections, it requires no support software in the form of assembler programs. The only support
software that may be needed is a way of actually loading the microprogram memory and even there one could think ways to get around using software to do this.

If the program should become very long this method can be very tedious. It is also relatively difficult to read the program many months after it is written. Modifying the program may also be very time consuming. For example if one inserted a few new instructions in the middle of a program, then one might need to change many other instructions in order to correct the branch address field for those instructions which have changed their address. The hand-coded binary method should probably only be used for very short and simple microprograms which do not need to be modified often.

2.2 DEDICATED MICROASSEMBLER.

A dedicated microassembler is an assembler which has been written to assemble programs for one microprocessor. If the processor is simple, and one does not expect an assembler with many of the sophisticated features we normally associate with assemblers that come with computers, then one can write a dedicated microassembler relatively quickly.

As an example of a dedicated microassembler, let us study a microprocessor designed by Guzik for use at experiment at Fermilab[2]. Its purpose was to read data from a CAMAC crate and make a decision on whether the event should be read out by the host computer. Figure 2 is a simplified block diagram of the processor which is based on the 2901A bit slice microprocessor and a 2909 microsequencer.

As is shown in figure 3, the microinstructions are 24 bits in length with an 8 bit control field and a 16 bit operand field. The control field has bits that are routed directly to the control points within the processor. Two bits control the next address multiplexer of the 2909 (S0,S1). The SEN bit enables the stack file of the 2909. The CND bit allows the SIGN bit from the 2901A to be ORed with the least significant bit of the microinstruction address. This is the only form of conditional branching the processor can execute. The MPX bit was used for iterative multiplication. The DAT bit controls whether the operand field is clocked into the accumulator register or the CAMAC address register. The MPX bit controls the multiplexer at the input to the 2901A. It selects either the accumulator register which could be loaded from the 16 bit data operand field of the instruction or the data on the CAMAC READ lines. And finally, the MOD bit controls whether the 2901 or a register is clocked. The operand field may be used for one of four purposes: a 16 bit data word, a 16 bit microinstruction address, a instruction for the 2901, or a CAMAC command. Figure 4 shows the bit assignments in the operand field.

Figure 5 is an example of the processor's microassembly source code. The control field and operand fields are handled in different ways. For the control field the processor is divided into columns corresponding to the 6 control bits and the one 2 bit field. In these columns, the program writes a symbol to generate a 'b' or 'l'. These symbols are easily interpreted in terms of what the processor is controlling during that microinstruction. For some of the fields, the a blank means the Don't Care state of that subfield, while in others it means the Default value.

As discussed above, the operand field can have one of four different meanings. The first character of the operand field of the source code contains a symbol which tells the assembler which kind of operand follows. These symbols are

$ for data operand
* for microinstruction address
# for 2901A instruction field
# for CAMAC command

Within the rest of the operand field are further symbols to indicate parameters to put into the subfield, if any, of the operand. The microinstruction address may have a symbolic name or label and it is placed in the left most column of the instruction. The right most column is reserved for a comment field.

If we look in detail at a few instructions we should be able to see how the assembly language is used. The first instruction has the symbolic label 'START'. The MOD column has the symbol 'OPR' indicating that the 2901A will not function for this cycle. The DAT column has the symbol 'NAP' so that the operand field will be loaded into the CAMAC NAF register. The SEQ field has the symbol 'CNT' so that the next microinstruction will be taken from the microprogram counter register of the 2909. The operand field starts off with a 'l', thus the operand contains the CAMAC address and function. The symbol 'N(3)' means that the station number subfield should be filled with a '3'. Similarly, the symbols 'A(0)' and 'P(8)' cause the subaddress and functions subfields to be filled with '0'. The net effect of this instruction is thus a CAMAC Read at station 3 subaddress 0.

The next instruction is similar to the first except that the operand field is loaded into the accumulator as is indicated by the symbol 'DAT' in the DAT column. The operand field starts with the symbol '$' to show that what follows in a single data word and the symbol 'A' is used for this word. Thus this instruction loads a 16 bit data word from the
24 BIT MICROINSTRUCTION WORD

<table>
<thead>
<tr>
<th>OPERAND FIELD</th>
<th>CONTROL FIELD</th>
</tr>
</thead>
</table>

8 BIT CONTROL FIELD

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **S0** for 2909
- **SI** next address
- **S1** multiplexer
- **MPL** 0 = normal 2901A operation (default)
- **CND** 0 = SIGN flag disabled (default)
- **SEN** 0 = Stack disabled (default)
- **DAT** 0 = operand field to ACC, Register (DAT)
- **MPX** 0 = ACC, Register to 2901A D input (ACC)
- **MOD** 0 = 2901A operation (ALU)
- **DIR** 0 = Direct Data (DIR)
- **STK** 1 = Stack (STK)
- **REG** 1 = R-Register (REG)
- **CNT** 0 = Program Counter (CNT)
- **SEN** 1 = Stack enabled (SEN)
- **CND** 1 = SIGN flag enabled (CND)
- **MPL** 1 = conditional ADD in 2901A (MPL)
- **DAT** 1 = operand field to CAMAC Command Register (NAP)
- **MPX** 1 = CAMAC Read lines to 2901A D input (CAM)
- **MOD** 1 = Data or Address (OPR)

Figure 3: Control Field of Guzik's Microinstruction Word.

16 BIT OPERAND FIELD

DATA Operand

<table>
<thead>
<tr>
<th>DATA</th>
</tr>
</thead>
</table>

Microinstruction Address Operand

<table>
<thead>
<tr>
<th>ADDRESS</th>
</tr>
</thead>
</table>

2901 Instruction Operand

<table>
<thead>
<tr>
<th>D</th>
<th>F</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **CARRY-IN**
- **2901 Source code**
- **2901 Function code**
- **2901 Destination code**
- **A-Register address**
- **B-Register address**

CAMAC Command Operand

<table>
<thead>
<tr>
<th>E</th>
<th>L</th>
<th>N</th>
<th>A</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **CAMAC Function code**
- **CAMAC SubAddress**
- **CAMAC Station Number**
- **CAMAC LAN generation**

Figure 4: Operand Field of Guzik's Microinstruction Word.

The 2901A will function in the next instruction as indicated by the symbol 'ALU' in the MOD column. The operand field starts with the symbol 's' which means that the operand field source is be read as subfields of the 2901 instruction. In this instruction the 2901 source, function and destination codes are programmed to be 'D,s', 'OR', and 'F->Q'. The A and
B register address are not used which is indicated by the 'X' in the symbol 'A(X)(X)'. In the next instruction, however, the data on the CMAC Read lines are loaded into register 2 of the 2981. Note the symbol 'CAM' in the MPX column and the operand field symbol 'A(X)(2)'.

The next instruction illustrates how a subroutine call is programmed. The stack enable bit (SEN) is turned on with the next address multiplexer of the 2989 selecting the D inputs ('DIR' in column 58). The operand field must then contain a micro-instruction address so it starts with a 'R'. The operand field contains the symbol 'MULTI' which is also used further down in the program to label a microinstruction address. The microassembler will substitute the binary address of MULTI into the operand field of this instruction.

It is left as an exercise to the reader to read the rest of the program. One might accuse this microassembler of being rather primitive, but the author feels it fits well to the task it must do. The processor was designed to execute a simple program and one can even notice that not all the functions of the LSI microcircuits were implemented in the circuit. Likewise, the microassembler only is capable of doing what the programmer needs: to write the relative short programs that this processor will be used for. Programming this processor with the assembler is considerably easier than using the hand-coded binary method and yet the assembler is not so complex that it is difficult to write.

2.3 META ASSEMBLERS.

Assemblers for microprocessors or computers perform very similar tasks. The code that must be written to write an assembler is also very similar. It is possible to divide the task into those parts which are the same for all machines and those parts which depend on the processor's instruction set. Then, if an assembler were written to accept as input the definition of the processor, we could reuse this sort of assembler for many different processors. Such an assembler is called a 'microassembler' or a "meta-assembler".

A meta assembler operates in two phases, the definition phase and the assembly phase. The definition phase, which must be executed first, sets up tables with the programmer's defined set of instructions and their model format. That is to say, the programmer specifies the symbols which will be used in the assembly phase to produce the binary bit pattern of the microinstruction. The assembly phase then uses the output of the definition phase and the source program input and operates in the same way as an ordinary computer assembler.

In order to study the properties of a meta assembler, we will study in some detail a meta assembler called AMASAM which was written by Advance Micro Devices for users of their LSI microcircuits. We will only discuss only the some of the features of AMASAM in order to bring out the basic ideas. More details may be had in the reference manual which is included with the book on circuit specifications[3].

2.3.1 Definition phase example.

The definition phase of the AMASAM meta assembler has two basic statement types: the EQU statement and the DEF statement. An EQU statement is used to generate a symbolic name for a constant value or expression. An example would be:

```
S4: EQU B#100
```

which sets the value of the symbol 'S4' to a 3 bit binary number '100'. Once the EQU statement is made, any further reference to the bit pattern '100' may be made by using the symbol 'S4'. A choice of four number systems may be made. The programmer selects which one by the letter in front of the '#' symbol as follows:

- B for binary,
- Q for octal,
- D for decimal, and
- H for hexadecimal.

The purpose of the EQU statement is the same as in ordinary assemblers, it relieves the programmer of the tedious task of always coding the bit pattern. Instead he can code a symbol which makes the program not only easier to write but also much easier to read and understand at a later date. Also, if the constant bit pattern need be changed, it may be done only at the EQU statement rather then throughout the program where that pattern may be used. Consider for example setting up the function code field of the 2981. We may write the following EQU statements:

```
As an example of the DEF statement, consider the simple scanner example again. Figure 33 of reference [1] defines the fields associated with that processor. Using the AMDASM assembler we can define the following microinstructions to handle the branching portion of the microinstruction as follows:

```
CONT: DEF 4X,8#00,7X ; CONTINUE
BRDV: DEF 4VH#F,8#01,7X ; BR DATA VALID
BRDONE: DEF 4VH#F,8#10,7X ; BR DONE
JUMP: DEF 4VH#F,8#11,7X ; JUMP
```

In the first statement, the symbol 'CONT' is a microinstruction with 3 fields which are separated by a ',' . The first and last are Don't Care fields which is indicated by the 'X '. These fields are 4 and 7 bits in length respectively . The second field is 2 bits in length and contains the constant value '00' binary . The same number system specification as the EQU statements are used in the DEF statements . In the remaining statements the first field is a variable field four bits in length . The default value will have the value ' 0 ' hexadecimal . The second and third fields in these statements are like those of the first statement except the constant value is different . A complete input to the definition phase of the AMDASM assembler for the simple scanner is shown in figure 6 .

![Figure 6: Definition Phase Input for Simple Scanner.](image)

### 2.3.2 Assembly phase example

The assembly phase reads the source program statements, substitutes values for the constants and labels, and generates the bit pattern which is to be loaded into the microprogram memory. The symbols used in the source statements must be either those defined in the definition phase or from EQU statements given in the assembly phase. The meta assembler in this phase looks very much like an ordinary assembler except for its ability to overlay or concatenate several microinstructions into a single microinstruction word. This feature is made clear by considering statements for the program of the simple scanner.

The program flow of the simple scanner is given in figure 31 of reference [1]. In the first instruction, the processor must reset the DEVICE COUNTER and ADDRESS COUNTER and send the first NEXT signal. The next instruction will be the next sequential address. This instruction may be coded as

```
STOP: DEF 12X,1V#0 ; SET TO ' 1 ' TO STOP
```
shown in the first program statement of figure 7. Reading this statement from left to right, it is explained as follows. The 'START' is a label for this microinstruction address and it does not effect the instruction generated for this location in any way. The 'CONT' is a symbol from the definition phase which sets bits 4 and 5 to '00' and leaves all the other bits in the Don't Care condition. The symbol 's', means concatenation or overlay is to be performed by the assembler. The symbol 'ZMAC' is also from the definition phase and it sets bits 6 through 8 to '10'. Because the overlay symbol has been used the bits affected by both the 'CONT' and 'ZMAC' will be set by this instruction. Continuing to the right another 's' symbol follows which means that even more bits are to be set. The 'DEVICE' symbol comes next and from the definition phase we see that it sets three variable fields. Only two variable symbol names follow; 'NEXT' and 'ZDAC'. The value of the variable 'NEXT' is equal to 'l' because of the EQU statement in the definition phase. Thus the with combination 'DEVICE NEXT', the assembler will generate a '1' in bit 9 of the microinstruction. The next variable field of the 'DEVICE' instruction is not specified as we can see by the two commas occurring in a row. Thus the assembler will use the default value for this field and generates a '0' in bit 10. The last variable field is specified, and from the EQU statement for 'ZDAC', one can see that the assembler should generate a '1' in bit 11.

The next instruction is labeled 'DVCKR' and it overlays a 'BRD', 'MNOP', and 'DEVICE' microinstructions into this single instruction. The 'BRD' contains a variable field which is the first bits of the microinstruction. The variable to be used is 'DVON'. This symbol is used as the statement label for the fifth microinstruction in the program assembly. Thus the assembler takes the address of this instruction as the value of the variable DVON and puts this value in bits 0 through 3. The net effect of this code is for the program to branch to the instruction labeled 'DVON' if the 'DATA-VALID' signal is being received while doing no operation on either the buffer memory control or the device control signals. Step 1 on the flow diagram in figure 31 of reference [1] corresponds to this instruction.

One should now be able to follow the complete program for the simple scanner shown in figure 7. Note that for the fields of the microinstruction which control the buffer memory and the device counter, I have used two different methods of setting the bits in order to illustrate variable substitution. I could have used either method for both fields or combined them into one microinstruction model.

One would think that the program given in figure 7 would yield the same bit pattern for the microinstructions as shown in figure 34 of reference [1]. One mistake, however, has been made in the program as shown. It illustrates that microprogramming with a meta assembler is not as easy as it seems. The 'STOP' microinstruction model was used in the program statement in order to halt the processor. The use the 'STOP' symbol followed by the variable field '1' is correct in the last statement of the program shown in figure 7. In all the other instructions, however, this bit of the microinstruction is left in the Don't Care state because the symbol 'STOP' does not appear. The default value '1' which is desired for these instructions will not be invoked unless the symbol 'STOP' appears in the source statement. As will be seen in section 3, all the Don't Care fields must be translated to either '0' or '1' when they are moved into the microinstruction memory since obviously a memory can't store a Don't Care. Thus, unless all Don't Cares are translated to '0', the processor will not function pass the first instruction. Figure 8 shows the program corrected with the symbol 'STOP' appearing in each statement. In all but the last statement the default value '1' is generated because no variable is specified.
2.3.3 Other features in meta assemblers.

Even a meta assembler as simple as that part of the AMDSAM assembler described above can make programming a microprocessor much easier when compared to hand-coded binary. It may also be used for many different microprocessor projects and programs can be easily be modified when a processor is modified. There are other desirable features one would like to have in a meta assembler which would make programming even easier and a few of these features are described below.

First of all, one would like to remove the positional dependence of the variable substitutions. This may be accomplished if the assembler has what is called a 'keyword' ability. Consider the following source program statement which might come from a processor with a 2901A:

\[
\text{ALU} = (A(1), B(3), \text{OR}, \text{AB}, \text{FBP}), \text{BR} = (2, \text{LOOP})
\]

From the description of the 2901A, we can imagine that this statements calls for registers 1 and 3 to be selected for the A and B outputs respectively, an OR ALU function code with A and B as the two ALU operands, and the results loaded back into the register file. In the same microcycle the program should branch to the location labeled by the symbol 'LOOP' if the result is zero. The Keywords can be at least 'ALU=', and 'BR=' as well as 'A()' and 'B()'. A meta assembler which could correctly interpret the statement this way would be much easier to write programs for and the programs would be much easier to read. One could also imagine that symbols such as 'OR', 'AB', and 'FBP' could be defined in such a way that they would cause certain fields of the microinstruction to be generated independent of the position within the microprogram statement.

Another feature, which is sometimes very useful in ordinary assemblers, is the macro capability. It allows the programmer, in the assembly phase to define one or more instructions to be generated by a symbol defined as a macro. For example one could define a macro 'OR' which when written in a program statement thusly:

\[
\text{OR} (A(1), B(3)), \text{BR} = (2, \text{LOOP})
\]

would generate exactly the same microinstruction shown in the previous paragraph. Some ordinary assemblers with macro capabilities allow the user to test for the number of operands. With such a capability in a meta assembler, the statement

\[
\text{OR} (A(1), B(3)), \text{BR} = (2, \text{LOOP})
\]

could be assembled as

\[
\text{ALU} = (A(1), B(3), \text{OR}, D, \text{FBP}), \text{BR} = (2, \text{LOOP})
\]

where the use of the D inputs for one operand was understood by the assembler because there were three operands in the argument list.

Another use of the macro capability is the generation of multiple microinstructions with one macro. Consider a macro called 'LOAD' which when written in a statement like

\[
\text{LOAD} Q, X4
\]

would generate

\[
\text{MOP=}(X4, \text{READ}), \text{CONT}
\]

\[
\text{ALU} = (A(X), B(X), \text{OR}, D, \text{FBP}), \text{CONT}
\]

which might be a memory read cycle with address X4 followed by passing the memory contents through the 2901 ALU in order to load it into the 2901 Q register.

One might also want an automatic default feature to avoid the kind of error that was made with the simple scanner program as shown in figure 7. That is, with the automatic default feature, any model microinstruction not appearing in the microprogram statement would be set to its default field. Such a feature however, might lead to other kinds of programming errors if the programmer had to make a real choice of possible variables. On the other hand, in all the microprogram statement examples we have shown so far, one had to always specify a 'CONTINUE' for the next address selection. It would be easier for the programmer if the assembler defaulted to 'CONTINUE' unless a 'Branch' was explicitly stated.

2.3.4 Difficulties with meta assemblers.

It seems that with an assembler of the type described with the AMDSAM assembler that microprogramming can be quite easy. One way of analyzing the difficulty is the consider some of the errors one might make and when these errors can be detected. Table 1 gives a possible list of errors and also shows what kind of errors are unique to microprocessor assemblers as compared to ordinary assemblers.

2.3.5 Obtaining a meta assembler.

The meta assembler is a good, if not essential, starting point for developing the necessary microprograms. The question is: How does one obtain a meta assembler which has the capabilities required for microprocessors we would find in our laboratories. The decision is basically whether to buy one or to write one.

One can write a meta assembler in FORTRAN or assembly language. It is not as formidable a task as it might appear if one keeps the definition and assembly phases simple enough. Probably the most tedious part of the task is writing the code which recognizes the syntax of the character strings. One should consider a programming language which is good at this. The rest is simple.

Since the task of recognizing character strings is already done by the assembler one has with a computer, one might like to find a way to use this assembler to do most of the work. One way to use an existing assembler is to make use of its macro capabilities. This approach is highly desirable because many of the necessary capabilities of an assembler will be taken care of by the host assembler so the user will not need to rewrite them. These include the assignment of symbols to constant values, the handling of address labels, and the substitution of variable values into instructions. Also the writing of the assembler is made easier by the macro language capabilities for decoding of parameter fields and the general ability for manipulating character strings and bit patterns. There are also some higher level languages which have a macro capability, such as PL/1, which should be good for writing a meta assembler.

An example of this latter approach is the MIMIC assembler written at SLAC by Edward Frank.4 It
makes extensive use of the capabilities of the IBM 360/370 assembler. It consists of a set of macro definitions so that, except for EQU statements, every symbolic source statement calls a MIMIC macro. The general format rules of the IBM Assembler program are preserved, along with its error handling capabilities.

In the definition phase of the MIMIC meta assembler, the programmer writes the definition of his fields with a set of MIMIC macros. The macros generate a table of symbol names and may generate other macros. The table of generated macros are "Punched" to an output file which is in fact the output of the definition phase and they may be saved for future use.

In the assembly phase of the MIMIC meta assembler, the programmer codes the program using the symbols defined in the definition phase. The "Punched" output file is the set of macros necessary for this phase. These macros assemble the bit pattern of the desired microinstruction into assembler patterns. The binary output is generated by a "DC" (Define Constant) IBM assembler directive. It may then be saved using all the normal facilities of the operating system.

An alternative to writing a meta assembler is to rent one. There are several meta assemblers available from the time sharing computer services. In the U.S., for example, the Computer Sciences Corporation rents the AMDASM that we have already studied. Output is available on paper tape ready to program IBM's. The cost of such a service is about U.S. $100-$1000 per month depending on the amount of programming done. Despite the seeming high cost for the rental, this may be the best answer when one has a small project and does not expect to need the meta assembler for other projects.

The same meta assemblers that one can rent are generally available to buy in the form of either a FORTRAN source program or a Load module for a mini or micro computer. In addition, there are other meta assemblers that one can buy. Many of the semiconductor manufacturers, realizing that potential users of their LSI microcircuits need programming aids, are also in the business of selling meta assemblers. For example, Signetics sells a meta assembler written in FORTRAN, while Advanced Micro Devices sells an assembly program written for the 8008. There are also independent Software houses which have developed meta assemblers which are geared for the microprocessor logic engineer. Even IBM has a software product which is a meta assembler. It requires that the installation has APL. The cost of these meta assemblers is not available. A survey of commercially available meta assemblers, the reader is referred to a recent article by V. M. Powers and J. H. Hernandez [5].

It is interesting to note that the interest in meta assemblers has increased dramatically since the availability of the LSI microcircuits. The semiconductor manufacturers find themselves not only supplying the circuits but also some software to help their customers use them. In addition, many companies are publishing extensive application notes which give examples for the use of their circuits. Whereas microprogramming was the domain of a few academics and professional computer designers in the past, it is now the focus of a large public education campaign waged by semiconductor companies interested in bringing the technique to the largest body of people as possible.

Another method to obtain a meta assembler is to "steal" one. That is to say, get a copy of one from someone who is willing to give you a meta assembler he has written. Since the LSI microcircuits are becoming more and more in use in the High Energy Physics Laboratories, one will certainly find that some good meta assemblers will be developed. I have already mentioned the MIMIC meta assembler written at SLAC, for example, which may be used by anybody who has access to an IBM 360/370. Another one has been written by W. Wimmer at DESY [6].

### TABLE 1 Errors in Microprogramming

<table>
<thead>
<tr>
<th>PHASE</th>
<th>ERROR DESCRIPTION</th>
<th>OWNER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINITION</td>
<td>Illegal character strings</td>
<td>Ordinary</td>
</tr>
<tr>
<td></td>
<td>Undefined symbols</td>
<td>Microassemblers</td>
</tr>
<tr>
<td></td>
<td>Duplicate labels</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bad word length</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bad field length</td>
<td></td>
</tr>
<tr>
<td>ASSEMBLY</td>
<td>Illegal character strings</td>
<td>Ordinary</td>
</tr>
<tr>
<td></td>
<td>Undefined symbols or labels</td>
<td>Microassemblers</td>
</tr>
<tr>
<td></td>
<td>Duplicate labels</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Overlay error</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Misplaced variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Incorrect variable specification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Missing variable with no default</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Necessary field unspecified</td>
<td></td>
</tr>
<tr>
<td>RUN</td>
<td>Bad field position</td>
<td>Microassemblers</td>
</tr>
<tr>
<td></td>
<td>Bad field length</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wrong constants</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bad Timing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don't Cares are not</td>
<td></td>
</tr>
</tbody>
</table>

2.4 EMULATION OF EXISTING COMPUTER.

For a given project in which a microprocessor is to perform some task, one has a generally a great deal of design flexibility. That is to say there are many ways in which the circuit could be designed to perform the task. Given a processor with a reasonable instruction set, the programmer can accomplish almost any calculation he desires. One microprocessor on the market (the 8X386, made by Signetics) has only 8 machine instructions and yet one could imagine doing very complicated calculations with it. Obviously, if the main feature of a
microprocessor is speed of calculation for one particular problem, then the choice between many possible designs becomes smaller. An example is the M7 processor [7] which was designed to calculate the effective mass from two particles in the detector. It has the capability of doing two multiplications and one addition in a single cycle.

For a microprocessor which is designed for general purpose, there is generally a larger choice of design. However, there is one choice which can greatly reduce the programming difficulty. It is to emulate an existing computer which one has access to, so that software development can be done on it. An example comes from my own work with a microprocessor called the 16E/E which I have described in section 5.1 of this chapter. The design of this machine was the following. A general purpose microprocessor must have three basic features:

1. arithmetic and logical operations perhaps with a register file or accumulator,
2. a means of memory addressing with efficient indexing capabilities, and
3. a means of condition branching on previous results.

The design of the 16E/E chosen so that all these conditions was satisfied in a fashion which is so similar to the instruction set of the IBM 360/370 series of computers that one can easily translate IBM 360/370 programs into microprograms for the 16E/E.

The advantages of emulation can be seen if one follows the steps for writing and debugging a program. First, a program is written in a language which is available for the emulated computer such as FORTRAN or assembly. One can then use a compiler that exists for this machine to generate object code. Then the program is tested with real or simulated data on the emulated computer using all of that computer's resources for program debugging such as line printers, interactive terminals, graphics, and histogram and debug software packages. The working program can then be translated from the object code into the microprogram code of the microprocessor and when loaded into it, it should work the first time.

The advantage in overall effort can be quite substantial. For one thing, no software documentation for a microprocessor instruction set need be written or maintained. There would be no need to write an assembler or use a meta assembler. Simulators for a microprocessor are sometimes written to aid program testing, but no such simulators need to be written for a emulating microprocessor since the emulated computer serves this function. One should also realize that no retraining of programmers need be done which is important since most programmers only get very proficient with a particular machine language after they have much experience with it. The greatest advantage of all is that because of the emulation of another computer, the experimenters on the project can program the microprocessor themselves in a higher level language they understand, such as FORTRAN.

The emulation technique requires a restriction of the design of the microprocessor so that it best corresponds to the emulated computer, which is sometimes called the "target machine". The translator, which is the only special software which needs to be written, is a partner with the hardware design to produce the desired result. The net result is that one has a processor which is considerably easier to program yet is probably no slower in execution speed nor not much more costly than designs based on the same LSI microcircuits. They will be slower, however, than processors with a lot of dedicated high speed arithmetic units.

3. POST-PROCESSING MICROPROGRAM ASSEMBLY.

After assembling a program for a microprocessor on a host computer, one is ready to load it into the microprogram memory of the processor. This task is simple in concept but nevertheless requires some additional software which must be considered as part of the software for microcircuits.

The output from the assembly stage is an object code file containing the bit pattern to be loaded into the microprogram memory. It is in a form which is probably convenient for storage on the host computer's system and for transfer to the processor. The task is to convert the microprogram memory to the memory format used by the processor. This may require some additional code be added to the object code file before loading. The processor memory may be in the form of Programable Read Only Memory (PROM) or as a writeable control store (ROM). It also must be transported from the host computer to the microprocessor.

The programming of PROMs is a software task which must have knowledge of the microinstruction format and the particular PROM circuit chosen. First, the "Don't Care" states which may still exist in the program must be changed to either a logic '0' or '1' since the memory circuit can only store one state or the other. Some PROM circuits invert or complement all of its outputs and in such cases one would need to complement the object program file before loading into the PROM. Both of these problems are relatively easy to take care of during program transport.

The number of bits available in PROM circuits is increasing all the time. Yet, in most cases one can not put all of the microprogram into one single circuit. For example, the microinstruction for the simple scanner processor (figure 33 of reference [1]) has a width of 33 bits and requires at least 6 words. A survey of the available memory circuits shows that the economical choice for the simple scanner is to use PROMs with 32 words of 8 bits. There simply does not exist on the market a memory circuit with 6 words of 33 bits. Thus, the implementation of the instruction memory may be as shown in figure 9. Here two memories each containing 8 bits have been placed in parallel so that the first has outputs for the first 8 bits and the second contains the remaining 25 bits. Three bits of the second PROM are not used and are force to waste these bits. The same microprogram address is applied to both circuits. Since the next address logic of the simple scanner has an address field of only four bits, we have one address input line on both memories which is unused. In figure 9 we have tied that input to ground which forces that address line to 'B'.

The simple scanner example is a case where the program is smaller but the program instruction width is larger than any available memory circuit. When this PROM is programmed it is inserted into a special circuit which applies the required voltages to blow the fuses within the circuit or deposits the charge in the cells of EPROMs. In either case, the programming circuit needs as input from the object code only the bits from the whole instruction word which will be placed into one PROM circuit. This is one of the post-processing steps is to take the object file and generate a programming file for each PROM which is required to contain the whole instruction word.
Some microprocessors will have programs which are longer than the number of words available in PROMs. In this case, one would use multiple PROMs to contain the whole program as shown in figure 10. In this figure, we have illustrated how one can use the same 32 word by 8 bit PROMs to make a program memory of 64 words by 16 bits. The low order bits of the microprogram address are bused to each memory circuit. The most significant bit and its complement is generated so that a "Chip-Select" signal is sent to only one bank of PROMs at a time. Most memory circuits have a "Chip-Select" input which disables the output of the circuit when a False signal is received by the circuit. One can then tie the outputs of two circuits together to form what is called a "Wired-Or". Since only one memory bank is "Chip-Selected" at any time, the two circuit banks in figure 10 act as if they were one memory circuit of 64 words in length. The Wired-Or function is generally accomplished by having the outputs of the PROMs being either an Open Collector or a Tri-State output.

Figure 9: PROM layout for Simple Scanner.

4. SOFTWARE FOR TESTING MICROCIRCUITS.

Testing the microprogram may be a very difficult task, especially at the early stages of the project development when the microprocessor itself is not yet known to function properly. With ordinary random logic design if one does not get the correct results, then the fault must lie in the hardware. With ordinary micro or mini computers, if one does not get the correct result, then the fault must lie in the software. With microprogrammed processors, if one does not get the correct result, then the fault could be in either the hardware or software and one must try to isolate the problem. Thus, as well as carefully designing the processor and writing programs for it, one should also carefully design a means of testing the processor, and testing the programs that will run on it.

Let us consider for a moment, what the hardware faults in the processor might be. First of all, there may simply be a logic design error, for example a circuit may not perform as expected because one did not read the specifications carefully enough. Secondly, there may be errors in the fabrication of the processor, for example wires may be misplaced. There may be errors in the timing, for example, some results may be strobed into a register before they are ready. There may be "glitches", that is, noise pickup on some lines so that the wrong results are strobed into register or a clock input is generated at the wrong time. Although unlikely, there may even be some bad IC packages that need to be replaced with working ones.

4.1 TEST BOXES.

In the classical random logic design, one can usually "drive" the circuit with a pulse generator, and examine the functioning of the processor with an oscilloscope or logic probe. If the circuit has multiple input sources, one could build a test box to generate these inputs. The test bench setup would look like the one shown in figure 11. If we try to apply this same testing technique to microprocessor, we would undoubtedly run into some problems. The microprocessor is "driven" by its program and the processor operates on data coming from or going to external devices or memory. Thus we have three separate subsystems as shown in figure 12 and each of the subsystems must be tested to see if they function properly.
One could follow the same approach as with random
logic, that is by building a test box as shown in
figure 13. This test box can be more complex than
the random logic test box, however. For example, if
the microprocessor will eventually have a program
memory in PROM circuits, the test box may contain
a RAM memory so the user can put into it a variety of
test programs. When using the RAM, one would
disconnect the normal memory circuits from the micro-
processor. Of course, some switches and lights need
to be provided in order to read, write and modify the
contents of the memory. Also the device or data
memory may need to be simulated or at least a means
of preloading a data pattern into them must be
provided in order to test the processor.

A means of seeing what the processor has done must
be provided as well as a means of controlling it.
One could take as an example of the number of lights
and switches that should be provided, those items
that are used by a typical minicomputer. Table 2
lists what one finds on the front panel console of a
PDP-11/20 minicomputer. Indicator lights are
provided for the 18 address lines, the 16 data lines,
and several other miscellaneous status conditions
such as RUN, BUS, FETCH, etc. Sixteen switches are
provided for entering data or address information
which is controlled by the switches LOAD ADDRESS,
EXAMINE, and DEPOSIT. The processor itself in
controlled by the switches HALT/CONTINUE, SINGLE
STEP, and START. These indicators and switches are
about the minimal set that one could imagine in order
to test programs on the PDP-11/20. A test box should
at least contain these to test programs on a micro-
processor. With a microprocessor, however, the test
box might be even bigger than with a minicomputer
because typically the program memory has a width
greater than the data path and it is on a separate
bus, so that the test box must have enough switches
to handle both the program and data memory.

| TABLE 2 |
| PDP-11/20 Front Panel |

**INDICATOR LIGHTS:**
- ADDRESS BUS (18)
- DATA BUS (16)
- RUN
- BUS
- FETCH

**TOGGLE SWITCHES:**
- DATA (16)
- LOAD ADDRESS
- EXAMINE
- HALT/CONTINUE
- SINGLE STEP
- DEPOSIT
- START

Such test boxes have been build in the past to
code CAMAC. This approach has the advantage that
the box can be specialized to the needs of a
particular processor and it is a completely stand-
alone system. On the other hand, the test box
approach has certain disadvantages. It may have
limited capability, since it itself is probably
random logic. If one starts to add capabilities such
as stopping the processor at certain address or
loading the memory from some storage medium, the test
box may be more complex than the processor itself.
If the microprogram grows in size, it becomes
extremely tedious and vulnerable to error to manually
load the memory each time the power needs to be shut
off to make a hardware change.
4.2 USE OF LOGIC ANALYZER.

Even with a very good test box, there are difficulties in testing a microprocessor with certain programs. For example, an error in either the hardware or software may cause the processor to jump to some unusually address and begin to do seemingly random operations that make it difficult to trace back to the source of the error. With an oscilloscope or logic probe one sees only one or two signals at a time and then only after the scope is triggered. If the processor halts after the error than one has only a single trace to see on the scope or one must use a storage scope.

A very useful instrument which aids in testing the microprocessor is a logic analyzer. It is an instrument which records in an internal memory the logic level (i.e. 0 or 1) of its input in fixed time intervals. Since with microprocessor circuits, we are always dealing with standard logic signals, it is usually sufficient to look only at the logic level rather than the real signal. The advantage of doing this is that an analog analyzer can be built with many more channels of input than an ordinary oscilloscope. Logic analyzers are available on the market with up to 16 independent input channels. The number of samples recorded is limited by the size of the internal memory and it is typically up to 1024 samples. The sampling rate, or inversely the time between the samples, is limited by the speed of the internal memory. With the analyzers available today, it is typically 20 nsec, and with some models 10 nsec. The analyzers generally have switch selectable thresholds for the standard logic families such as ECL and TTL. The analyzer does not, however, record short signals or glitches if they don’t occur at the instance the signals are sampled. Some models have special input circuits called "glitch catchers" which take any transition in an input as the recorded level.

There are many advantages of a Logic Analyzer over an ordinary oscilloscope. For example, a very important advantage is in triggering. With an oscilloscope, one can see the input signals for a time period after the trigger signal. Similarly, the logic analyzer can record its inputs after a trigger signal has arrived which is called the "Pre-Triggering" mode. But the analyzer can operate in the other sense, that is, it can be continuously recording the input signals in a wrap around buffer and then stop recording when the trigger signal arrives. This mode is called "Post-Triggering" and it allows one to see all the input signals before the trigger signal. Some analyzers even allow one to put the triggering time in the middle of the memory storage so that one can see the input signals before and after the trigger time. Another important advantage is that once the analyzer is triggered, it can keep the recorded signals indefinitely, so that single shot events can easily been seen and studied.

The triggering abilities can be augmented by the use of a Word Recognizer which is frequently built into commercially available Logic Analyzers. This device allows the user to form his trigger on the combined state of many input conditions. The trigger can then be formed from something simple such as the transition of one signal input or as complex as particular bit pattern on the memory bus. Most available Word Recognizers accept up to 18 inputs to form the trigger and even allow one to make the trigger only after a number of occurrences of the same input pattern.

With most of the available Logic Analyzers, one has a choice of the way the recorded inputs are displayed. With an oscilloscope, one has only one mode. In this mode the signal levels are displayed vertically on the screen and time is displayed horizontally from left to right. Logic Analyzers can also display their data in this way and it is called the "Time Domain" mode. But they can also display their memory as data words with possibly a choice of binary, octal, or hexadecimal format. This mode is called the "Data Domain" and it is very useful for program development since it seems more like a program trace that the programmer is used to. Yet another mode is called the "Map" mode. It treats the input signals as one data word and plots a point on the screen for each possible data word. The most significant bits of the word form an displacement vertically while the least significant bits are used for a displacement horizontally. Thus one can get a feel of the flow of a program and the human eye can spot unusual events by points being very displaced from the normal pattern.

4.3 COMPUTER BASED DEVELOPMENT SYSTEM.

The logic analyzer greatly improves one’s ability to find errors in the microprocessor even with a simple test box. But not all the problems are solved with it. One still has the problems of loading by hand the processor’s memories and reading the results. Another approach is to use a micro or mini computer as the "test box". In this approach the design of the microprocessor would be made so that one could build an interface from a computer to the processor which allows the computer to gain access to the memories and various control points of the processor, for example the processor’s clock. One can than emulate the test box functions with software in the computer. Such a setup is illustrated in figure 14. The microcomputer in this setup is called the "Host Computer".

![Computer Based Development System](image-url)

Figure 14: Computer Based Development System.
Prototype boards are available to simulate the eventual microprogram P-ROM with RAM and a standard next address logic board using a microsequencer. This system costs U.S. $25,000. This price includes the AMDASM meta assembler, an operating system for file manipulation on the 8080 microcomputer, and other software to aid in debugging the microprocessor under test.

The use of a micro or mini computer to be the controlling element for the microprocessor is not unique in the computer field. One can find similar examples in large computer systems. For example, the Amdahl 478/V6 computer has a Data General NOVA computer built into its console display. The NOVA can also almost everybody register file. Another example is the Digital Equipment Corporation's VAX-11/780 in which an ISL-11 microcomputer handles the system terminal. In both computers the micro or mini computer handles the diagnostic routines and the system console. Neither machine has anything in the way we normally think of computer front panel consoles.

4.4 USE OF COMPUTER CENTER'S COMPUTER.

Given that one has a host computer that acts as the front panel for the microprocessor, one still has to consider on how one is going to store files and run the microprocessor's assembler program. This assembler program could well run on the host computer but not without some additional peripherals on this computer such as mass storage, and line printers. In High Energy Physics, most of the laboratories where one would be doing the microprocessor development work have a computer center and the question arises as to whether one should use the computer center for file storage and other miscellaneous tasks. In many cases a simple connection between the host computer and the computer center for the transfer of files is an approach which offers many advantages.

First of all, the peripherals are the most costly part of a computer system, especially with microcomputers since the CPU cost is very low. By using the computer center's peripherals one can greatly reduce the cost of the local computer system. Even with some local peripherals such as a floppy disk, one generally has a more limited program development ability on a small micro or mini computer. One is also more likely to find useful cross assembler software available for the computer center computers than the inexpensive local computer.

All of the above advantages are obvious at first glance and may very well justify using the computer center for microprocessor program development. But there are also many hidden advantages which may be equally important from the point of view of hardware costs and more important in terms of man power costs. First of all, one can reduce the cost of the local computer to the bare minimum. This could mean that it would consist of only the CPU, some RAM memory, an interface to a terminal, an interface to the computer center, and some ROM memory to get started. Since there will be no moving parts, the cost is only in the peripherals. The cost of the computer center also leads to a substantially reduced maintenance cost. Very little space in the laboratory would be required for the system which allows more room to work on the microprocessor or allows more flexibility on where to place the equipment.

There is also a very large reduction of time consuming tasks for the personal involved in the project. All the facilities of the computer center
are presumably already known so there is no lost of
time to retrain people with a new system. The
computer center most likely has a better text editing
system than those found on small systems. There will
also be a file management system with much more space
available to the user. Included with the file system
will be routine data management system, that is, a
system where by files are archived and backup tapes
are regularly made. When batch jobs are to be
submitted to do cross assembly work one can use the
job entry system which is already well known including
all the job control language that is necessary. The
computer center will have many more peripherals and
more kinds of peripherals than one would think of
putting on a low cost local computer system. They
may include high speed line printers, microfiche
printers, graphic plotters and terminals, etc. One
has global access to the computer center from any
terminal in the laboratory rather than the one
terminal for the local computer which allows more
flexibility in working. The computer center is also
set up to be multi-user, so that more than one person
can work at a time on software for the microprocessor
project and yet they can share the same data base
files.

The cost of the connection between the local
computer and the computer center can be minimized
when one realizes that the data transmission rate
need not be very high. For example one can let the
local computer emulate an ordinary terminal, that is, a
point of view of the computer center. Data files
could then be prepared before transmission with
a certain keyword characters so the local computer can
take the characters as data rather than repeating
them on its terminal. In this way no software for
the connection need be written for the computer
center's computer and the local computer need only
have an ordinary terminal interface to the center and
a very small amount of software. Such a simple
connection was used for the local computer for the
168/E test bench shown in figure 15. The terminal
emulation program located in ROM is only 256 words in
length.

Many laboratories have or are developing computer
networks which would allow high speed data
transmission between the computer center and the
local computer. They offer better facilities in
bringing down large programs with error checking and
correcting on the transmitted data.

5. SUMMARY

We have seen that there are two parts of
the software for microcircuits. The first part is the
programs for the microprocessors, while the second is
the software support programs. The support software
can turn out to be much more extensive than the
processor's programs. This does not mean that the
user of microprocessors will drown in a ocean of
support software, but he must simply learn how to
swim in deep water.

References

[1] Paul F. Kunz, "Microcircuits of High Energy
Physics", Proc. of the 1978 CERN School of Computing,


[5] V. Michael Powers and Jose H. Hernandez,
"Microprogram Assemblers of Bit-Slice


ON-LINE FILTERING

C. Verkerk
CERN, Geneva, Switzerland

1. INTRODUCTION

Present day electronic detectors used in high-energy physics make it possible to obtain high event rates and it is likely that future experiments will face even higher data rates than at present. The complexity of the apparatus increases very rapidly with time and also the criteria for selecting desired events become more and more complex. So complex in fact that the fast trigger system cannot be designed to fully cope with it. The interesting events become thus contaminated with multitudes of uninteresting ones. To distinguish the "good" events from the often overwhelming background of other events one has to resort to computing techniques.

Normally this selection is made in the first part of the analysis of the events, analysis normally performed on a powerful scientific computer. This implies however that many uninteresting or background events have to be recorded during the experiment for subsequent analysis. A number of undesired consequences result:

- the dead-time losses for recording the uninteresting or "bad" events can be considerable.
- a large number of magnetic tapes has to be written and read, imposing a heavy burden of work and administration.
- a very considerable amount of costly computer time is used in relatively simple calculations needed to narrow down the event selection.

Each single one of these effects constitutes a sufficient reason for trying to perform the selection at an earlier stage, in fact ideally before the events are recorded on magnetic tape. We will call this early selection "on-line filtering" and it will be the topic of the present lectures.

We will first have a closer look at some electronic detectors and experimental techniques in order to clarify more precisely why we want to perform filtering of events on-line. We will then, after having defined what we want to do, give some insight in how it can be done. This latter part will be based on several examples of filtering, performed on running or planned experiments and will cover a wide range of implementations: from pure software to the fastest hardware techniques.

As these lectures were intended to be of a tutorial nature, the examples taken from present or future experiments have been chosen with the sole purpose of illustrating the material presented. The experimenters should therefore not be held responsible for possible misrepresentations, which are entirely to be attributed to the author.

2. GENERAL

2.1 A typical large experiment

2.1.1 Detectors

A better understanding of the need for on-line filtering can be obtained by examining more closely a typical experiment. We have chosen for this purpose an experiment¹ which is at present in preparation and which is typical in the sense that it makes use of many different types of detectors, that the selection criteria are based on the signals from several of those detectors and that the experimenters plan to make extensive use of on-line filtering techniques.

A schematic of the planned lay-out of this experiment is shown in Fig. 1, while Fig. 2 gives an enlarged view of the first metres of the equipment. One of the characteristics of large experiments is apparent from inspection of these figures: many types of detectors are used. Each has specific properties and is therefore useful for measuring specific quantities of interest in the analysis of the events. Wire chambers or drift chambers provide space coordinates along particle trajectories, allowing momentum measurements when a magnetic field is present. Several detectors of the same type are then used to provide the coordinates: in Fig. 2 five drift chambers and ten planes of proportional chambers are shown.

Čerenkov counters measure velocity and thus make mass-determination possible when combined with momentum measurements. Calorimeters measure the amount of energy deposited when a particle traverses matter and make further distinction possible between particles. They also provide a direct energy measurement in case of total absorption.
2.1.2 Data volumes

The majority of these electronic detectors generate considerable amounts of data; some typical values are:

- Multiwire proportional chambers: 10,000 or more wires, spread over a dozen or more planes are typical for a present day experiment. Approximately 50 wire clusters will be hit in an event and must be read out and recorded.

- Drift chambers: 1000 wires are typical, with again approximately 50 hits in an event. Multiple hits on a single sense wire are possible. The coordinates are only known after a delay ($\approx 1 \mu s$), caused by the drift time of the electrons.

- Calorimeters: the scintillation counters which are sandwiched between plates of heavy material require of the order of 500 photomultipliers.
A typical event will activate > 20 channels, where also the pulse height of the signal must be measured and digitized.

- Scintillation counters: they are the workhorses of every experiment, as the primary trigger is derived from their signals. Several tens of counters are typically found in an experiment.

Other detectors, like ionization detectors and Time Projection Chambers are being developed, which will provide still larger amounts of data.

From this brief reminder of some of the characteristics of detectors one can see that a typical event is described by 200-500 words (16-bit) of data, giving mainly coordinates and pulse heights, but also patterns of counters which are hit and general information (magnet currents for instance). Larger experiments can easily reach 1000-2000 words/event and future experiments will probably require even more information to describe the event.

2.1.3 Limitation of the trigger selectivity

The trigger electronics in its first and very fast decision based on signals from scintillation and Čerenkov counters cannot take all the additional information into account and its selectivity is therefore limited. Several reasons make it difficult to improve the selectivity. To distinguish events from all the other things (noise) happening in the detectors, the time correlation of the particles is essential. The resolution time of the coincidence circuits is generally in the few nanosecond region, so the electronics must be very fast. The modules used accommodate only a few simple logic functions with a small number of inputs per module. For large numbers of counters and complicated trigger requirements the number of modules needed rises above all acceptable limits. The number of NIM crates (containing the fast electronics; at 20 modules/crate) averages about 30 for the experiments at present on the floor at CERN. The larger experiments use up to 70 NIM crates.

The counters used in the trigger are usually rather large and this has as a consequence that the selectivity on kinematical quantities is reduced. Another limitation on the fast trigger is that the precise data from wire and drift chambers cannot be used, either because it represents far too many input signals or the data are not available in time.

2.1.4 Types of background

The fast trigger, although essential for detecting time-relations and for providing the start signal for further actions, will therefore leave the experimenter with events highly contaminated by background. This background can be of different nature: firstly purely random occurrences of signals can lead to a trigger without a real event being present. Secondly there are the events which genuinely satisfy the trigger requirements, but which will turn out to be of an undesirable type when further and more sophisticated selection criteria are applied. Another class is formed by those events which, although they belong to the reaction being studied, are nevertheless in an uninteresting kinematical region. This could for instance be the case when elastic scattering at high momentum transfer is being studied. A simple trigger could detect the presence of a pion in one and of a proton in the other arm of a spectrometer (using Čerenkov counters to make this distinction). As the differential cross-section decreases exponentially with $|t|$ the events with $|t| \geq 5$ GeV will then be entirely drowned, if no adequate selection on the scattering angle is made at the same time.

2.2 Definition of on-line filtering

The primary purpose of the analysis is of course to eliminate the background, applying progressively more restrictive selection criteria. The above example of the elastic scattering shows however that a large pay-off will be obtained if part of the more stringent selection criteria could be applied in real-time, e.g. before the event is written onto tape and recorded for off-line analysis. We will call this process on-line filtering. It is important to realize that this definition of on-line filtering implies eliminating events (hopefully the large majority of triggers) before they are recorded. These events are therefore lost without any possibility of recovery.

On-line filtering should also be clearly distinguished from certain other treatments the data could undergo in real-time. Monitoring of an experiment by building up histograms and plots of rates, particle distributions, etc. is essentially different. So is sample analysis, a more sophisticated form of monitoring. Often the data are transformed on a word-by-word basis: reformatting, packing or clustering (of wire hits) are examples. In
general these processes have to remember one or more of the previous words of data, but they can be performed for a single detector, independently of the signals from the other detectors. A filter on the contrary works on the ensemble of the data from more than one detector (but not necessarily on all the data from all detectors).

On-line filtering introduces the concept of a multi-level decision process performed in real-time. Two levels are normal: the fast trigger followed by the slower filter. One can however very well think of several levels of filtering, each level working on the events accepted by the previous level.

The filter is obviously always slower than the fast trigger, but in most cases it must work under severe time constraints, to avoid excessive deadtime losses or the need for running at a lower trigger rate.

### 2.3 Example of filtering in a planned experiment

To illustrate some of the above we return to the experiment shown in Figs. 1 and 2. The purpose of this experiment, at present in preparation, is the observation of charmed particles, characterized by the production of a prompt electron. The presence of the electron is ascertained by the Čerenkov counter and by the energy deposited in the calorimeter. This energy must fall below the limits of 2 and 10 GeV. Electrons in this energy range are however produced in abundance by other well-known phenomena: decay of π⁺ and η-mesons followed by conversion of the γ-rays. A careful selection of the events must be made to discriminate against these and other sources of background. In the proposal for the experiment, the physicists made a careful analysis of these sources of background, which we will not reproduce in all detail. They propose to eliminate the undesired events in different steps. The primary trigger requires an electron depositing > 2 GeV in the calorimeter (class 0). Additional criteria for the rejection of events with a pair of electrons (instead of a single one), are also applied in the fast trigger logic. Also for this purpose, extra counters have been added catching the second electron when it goes off in uncommon directions. Table 1 shows the rates of events (per 10⁶ interactions) in which the second electron undergoes the fate given in the first column. The complete hardware trigger reduces this background by a factor 4, compared to the class 0 trigger.

<table>
<thead>
<tr>
<th>Fate of second electron</th>
<th>Rate of class 0</th>
<th>Rate of hardware trigger</th>
<th>Rate of software trigger (to tape)</th>
<th>Rate of after-off-line rejection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger calorimeter</td>
<td>10.84</td>
<td>0.00</td>
<td>0.78</td>
<td>0.06</td>
</tr>
<tr>
<td>H-Counters</td>
<td>56.30</td>
<td>0.00</td>
<td>0.20</td>
<td>0.01</td>
</tr>
<tr>
<td>Misses both but picked up in P4</td>
<td>8.00</td>
<td>8.00</td>
<td>0.78</td>
<td>0.06</td>
</tr>
<tr>
<td>Misses all three but picked up in P7 or P8</td>
<td>4.56</td>
<td>4.56</td>
<td>0.20</td>
<td>0.01</td>
</tr>
<tr>
<td>Emerges downstream but not detected</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>Hits sides of MDP-33 and picked up in P7/8 or P9/10</td>
<td>4.26</td>
<td>4.26</td>
<td>0.31</td>
<td>0.00</td>
</tr>
<tr>
<td>G-Counters</td>
<td>1.41</td>
<td>0.00</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>Hits MDP pole faces and picked up in P7/8 or P9/10</td>
<td>2.09</td>
<td>2.09</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>F-Counters (reflected electrons)</td>
<td>6.42</td>
<td>0.00</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>Other lost electrons</td>
<td>0.66</td>
<td>0.66</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>Total</td>
<td>74.44</td>
<td>19.50</td>
<td>2.09</td>
<td>0.86</td>
</tr>
</tbody>
</table>

An on-line filter, using microprogrammable processors (ESOP, see below) will further reduce the rates of these background events to the numbers given in the fourth column (the final rates after off-line analysis are given in the fifth column). The microprocessors will work on data from different detectors and apply different criteria to improve the selection. A block diagram is shown in Fig. 3, which is in fact part of the over-all data acquisition system. Different tasks are distributed to a number of processors. One of the processors can make a finer analysis of the pulse heights produced in the calorimeters, together with the position information provided by the drift chambers and thus ensure that not

---

**Fig. 3** Basic block diagram for 2-stage decision (hardware trigger followed by on-line filter) in the experiment shown in Figs. 1 and 2. In principle more than one microprocessor is foreseen, each working on a sub-task.
more than one electron was produced. Another can trace back particles found in wire planes P4-P10 in different combinations (see Table 1) and determine their momentum. A further aid in suppressing the background comes from the pronounced difference in $p_T$ distribution of electrons from $\pi^0$ decays and those from charmed particle decays (Fig. 4).

![Distribution of transverse momentum for electrons from charmed particle decays (full curve) and from $\pi^0 \rightarrow \gamma \gamma \rightarrow 4e$ decays (dotted curve).](image)

Fig. 4 Distribution of transverse momentum for electrons from charmed particle decays (full curve) and from $\pi^0 \rightarrow \gamma \gamma \rightarrow 4e$ decays (dotted curve). This is an example of a selection criterion applicable only after $p_T$ has been calculated (in an on-line filter).

From Table 1 one sees that this on-line filter, executed before the events are written to tape is expected to reduce the background by an order of magnitude; another factor 2-3 is gained in the off-line analysis.

Other sources of background have also been considered and also here, as Table 2 shows, the on-line filter is expected to reduce the rate by considerable factors (2-10). The numbers in Table 2 are based on an estimated rate of $1.4 \times 10^5$ interactions/burst. It is important to note that a hardware trigger rate of $\approx 500$ triggers per burst (duration $\approx 1$ second) is expected, which means that the filter is constrained to a time of less than 1 millisecond, if dead-time losses are to be kept reasonable.

### Table 2

<table>
<thead>
<tr>
<th>Background source</th>
<th>Triggers/burst</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hardware</td>
</tr>
<tr>
<td>1) $\pi^0, \eta$</td>
<td>280</td>
</tr>
<tr>
<td>2) $K^\pm + e^\pm, \pi^0\nu$</td>
<td>4</td>
</tr>
<tr>
<td>3) Prompt $e^\pm$</td>
<td>2</td>
</tr>
<tr>
<td>4) $\pi^\pm$</td>
<td>41</td>
</tr>
<tr>
<td>5) $\pi^\pm\pi^0$</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>507</td>
</tr>
</tbody>
</table>

2.4 Trigger rate and filter speed

We will examine a little more precisely the relationship between the filter speed, the trigger rate and the purity of the sample obtained by the first level trigger. We assume that we have $N$ triggers per second, containing $\rho N$ "good" events, e.g. events which are still considered acceptable after the filter process and retained for off-line analysis. We further assume that the filter introduces a dead-time after each trigger and that the dead-time $\tau_B$ following a "bad" event is different from the dead-time $\tau_G$ following a "good" event. The time $\tau_T$ needed for a complete read-out of the event data is included in $\tau_I$, while $\tau_B$ includes the time for partial read-out, limited to the data needed for the filter. Without the filter a dead-time of $\tau_I$ follows each trigger and the observed rate of unfiltered events is therefore:

$$U = \frac{N}{1 + N \tau_T},$$

containing

$$U_g = \frac{\rho N}{1 + N \tau_G}$$

good events.

With the filter in place the observed rates of bad and good events will be $R_b$ and $R_g$, respectively. Fractions $R_b \tau_B$ and $R_g \tau_G$ of time are then lost following the triggers on bad and good events, respectively, so that:

$$R_g = (1 - R_g \tau_G - R_b \tau_B) \rho N$$

and

$$R_b = (1 - R_g \tau_G - R_b \tau_B)(1 - \rho)N.$$
From (3) and (4) follows:

\[ R = \frac{dN}{\rho} \left( 1 + (1 - \rho)N_{\tau} + \rho N_{\tau} \right). \]  

(5)

The on-line filter will be particularly attractive if it would allow us to record more good events on tape in a given time interval than would be possible without filter. In other words: can we have situations where

\[ R > U. \]  

(6)

From (2) and (5) it is immediately seen that the condition for \( R > U \) is (supposing \( \tau = \tau_g \)):

\[ \alpha - \alpha \rho + \rho < 1, \]  

(7)

where \( \alpha = \frac{\tau_b}{\tau_g} \). As \( \rho \leq 1 \) by definition, \( \alpha \) must be less than one, for (6) to be satisfied. The necessary condition for \( R > U \) is:

\[ \alpha = \frac{\tau_b}{\tau_g} < 1. \]  

(7)

The gain in the rate at which good events can be recorded is then:

\[ G = \frac{R}{U} = \frac{1 + N_{\tau}}{1 + (1 - \rho)N_{\tau} + \rho N_{\tau}}. \]  

(8)

Maximal values of \( G \) are given in Table 3 for \( \tau_g = \tau_r \).

Table 3

Gain factors in recording good events on tape

\[ G = \frac{1 + N_{\tau}}{1 + (1 - \rho)N_{\tau} + \rho N_{\tau}} \]

The table entries are for \( \tau_g = \tau_r \), \( \tau_b = \tau_g \).

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( \rho = 0.1 )</th>
<th>( \rho = 0.01 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_{\tau} ) =</td>
<td>( N_{\tau} ) =</td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>0.5</td>
<td>1.04</td>
<td>1.19</td>
</tr>
<tr>
<td>0.1</td>
<td>1.08</td>
<td>1.68</td>
</tr>
<tr>
<td>0.05</td>
<td>1.08</td>
<td>1.75</td>
</tr>
</tbody>
</table>

One sees that for reasonable values of \( \rho (\approx 0.1) \) it is in fact possible to increase the rate of recording good events, but a considerable gain can be obtained only when the dead time following a bad event is small and when \( N_{\tau} \) is large.

We can also interpret the result in a slightly different way. If one accepts a certain level of dead-time losses, then an on-line filter will allow running at a higher trigger rate, without increasing these dead-time losses. Without a filter the trigger rate for a dead-time loss of a factor \( 1/(1 + k) \) is \( N_{\tau} = k \). With the filter in place the same losses will result for a trigger rate \( N' \) when

\[ (1 - \rho)N'_{\tau} + \rho N'_{\tau} = k. \]  

(9)

Supposing again that \( \tau_g = \tau_r \) we find

\[ \frac{N'_{\tau}}{N} = \frac{1}{(1 - \rho)N + \rho}. \]  

(10)

The trigger rate can thus be increased by the values given in Table 3, without affecting dead-time losses.

2.5 Other limiting factors affecting event rates

Until now we have tacitly assumed that the read-out time of an event was the limiting factor in obtaining high recording rates. Two other factors may limit this rate and thus have an influence on the desirability of filtering: tape writing speed and the size of the available buffer memory. A distinction must be made between continuous machines (the ISR for example) and pulsed machines (e.g. SPS).

Supposing that we use normal 1600 bpi, 75 ips tape, the highest rate at which we can write 16-bit words is approximately 50K words/s. Assuming that a good event at the ISR requires 500 words, we cannot hope to record more than 100 events per second. The read-out time will be close to 1 ms and if we assume that a bad event could be rejected in 100 \( \mu \) s, we find from (5) that the trigger rate would be limited to \( \approx 1200 \text{ s}^{-1} \) if \( \rho = 0.1 \). The dead-time losses would then be \( \approx 23\% \). We are therefore limited in rate, but on the other hand, since we are recording continuously, we do not need more buffer capacity than is required for two physical records on tape.

For a pulsed accelerator like the SPS one can make use of the interval between pulses to write tape, but the events have to be stored in a buffer during the beam bursts. We will assume that events at the SPS are larger than at the ISR and consists of 1000 words of data. If we also assume that they can be read at a typical DMA speed of 1 word/\( \mu \) s, we could read out an absolute maximum of 1000 events per 1 second burst and write at most 300 events on tape during a full 6 second cycle. In principle, we could buffer the raw events and apply the filter before the events are written to tape. We would then need a very large buffer: 1M word. On the other hand the time one can spend on a good event would be of the order of 10 ms and one could think of times
of the order of 1 ms for rejecting bad ones. Tape writing would not constitute a real limit for \( p \leq 0.3 \).

When we buffer only the good events, \( \tau_b \) has again to be small compared to \( \tau_r \). Assuming \( \tau_b = 50 \text{ ms} \) and \( \tau_r = \tau_g = 1 \text{ ms} \), we find that for \( p = 0.1 \) we can run at a trigger rate of \( \approx 5000 \text{ s}^{-1} \), with \( \approx 37\% \) dead-time losses. The buffer needed would be \( 300 \text{K words} \). These figures represent severe conditions for an on-line filter, but one can conclude that in most cases the tape-writing is not the limiting factor, but rather the performance of the filter itself and/or the buffer size. This is the more true for 6250 bpi tapes.

2.6 Requirements for filter algorithms

We have seen that for an on-line filter to be effective it is very important that it is capable of rejecting undesired events in the shortest possible time. Obviously one also wants to spend a minimum amount of time on the good events, although this is of lesser importance, particularly for small values of \( p \). We have also seen that the decision making requires in general numerical calculations to be made on part of the data of an event.

These calculations can in general be performed with a rather low precision, since we are not trying to obtain accurate results. We only want to calculate the value(s) of one or more parameters, so that a decision may be taken. In making the decision we will always be prudent and avoid throwing away good events. Safety margins are therefore large and the accuracy attained in the calculation of the parameter is of secondary importance. In practically all cases the calculations can therefore be performed using integers of 16 (or less) bits.

The speed requirement on the contrary may impose severe constraints on the filter algorithms. Each case has to be considered on its merits, but generally speaking it is imperative to avoid time consuming procedures such as least squares fits and iterative approximations. One has also to beware of combinatorial problems, where the execution time \( T = \alpha n^k \) increases with a large power \( (k \geq 3) \) of the number \( n \) of data points (the amount of data is roughly proportional to the number of particles, and thus increases with energy). Unfortunately this dependency on a power of the number of particles cannot always be avoided entirely. It is therefore worth while to examine the problem carefully, and to try using short cuts, with the aim of reducing at least the proportionality factor \( \alpha \). For instance, one should avoid continuing the execution of nested loops, when the application of a simple test would indicate that no further solutions can be found.

Another point to take into consideration is the importance to be given to the treatment of particular cases. Each particular case will complicate the algorithm, which might be undesirable. It is particularly awkward if a hardware implementation is envisaged. It need not necessarily lead to a loss in speed, although it often does add some extra time. The disadvantages of a proper treatment of the special cases must therefore be carefully balanced against the expected gain. A point-finding algorithm, to determine space points in wire chambers with four wire planes may serve as an example. The algorithm, which has been described in another elsewhere, is basically very simple and consists of executing nested loops over four sets of wire coordinates and checking if two conditions

\[
|X + Y - 2U| < \epsilon \quad (11a)
\]

and

\[
|Y - X - 2V| < \epsilon \quad (11b)
\]

are fulfilled simultaneously. In a time proportional to \( n^3 \) (\( n \) is the average number of hits per wire plane) the space points are found where the traversing particle generated a signal in all four wire planes. To find also those particles which gave a signal on three planes only (because of inefficiency of the fourth) the algorithm has to be extended, adding considerable complication. Two more scans through all the coordinates are required, so the execution speed is affected by a factor 3. This loss in speed and the added complication must be balanced against the gain in efficiency of finding space points. For an average efficiency of a wire plane of \( 1 - \eta \) (\( \eta \ll 1 \)), a fraction \( 4\eta \) of points cannot be found by the basic algorithm, which requires a signal on all four planes. Considering that the space points of several chambers are often needed in further stages of the filter process, the fraction of events lost can be rather large. For instance, if the chambers are 98% efficient and points in 4 chambers are needed for event reconstruction, one would lose around 16% of the events, if the simple basic algorithm would be used.

Sometimes the algorithm can be simplified or its speed improved by an appropriate design of the detectors. Examples exist of wire chambers designed
to avoid the need for multiplications in the point-finding algorithm. This has for instance been done by choosing a wire spacing in the planes with inclined wires (by ±45°) a factor \( \sqrt{2} \) different from the spacing in the planes with wires at 0° and 90°\(^\circ\)\). This is the reason why in (11a) and (11b) the factor 2 appears, instead of \( \sqrt{2} \). In any implementation of the algorithm (except on the largest scientific computers) the factor \( \sqrt{2} \) would have been much more awkward.

One can also make judicious choices of the wire directions. A simple example is given in the experiment described above\(^1\). (For another example, see Section 5.3.1.) From Fig. 5 it is clear that

![Figure 5](image)

**Fig. 5** The coordinates \( x, y \) of a particle can be determined from the wire hits \( n_1 \) and \( n_2 \) in two planes with inclined wires. A judicious choice of \( \phi \) can speed up the calculations (see text).

when wire \( n_1 \) is hit in the first plane and \( n_2 \) in the second, the \( x \) and \( y \) coordinates of the particle are given by:

\[
x = n_1 + y \tan \phi
\]
\[
y = n_2 - y \tan \phi
\]

from which follows:

\[
x = \frac{n_1 + n_2}{2} \quad (12a)
\]
\[
y = \frac{n_2 - n_1}{2} \times \frac{1}{\tan \phi} \quad (12b)
\]

\( \phi \) can be chosen so that \( \tan \phi = \frac{1}{2} \) or \( = \frac{1}{4}, \frac{1}{8}, \) and (12a) and (12b) become very simple.

Straight-line finding can be simplified -- by eliminating multiplications -- when the distances between detectors can be chosen judiciously.

2.7 Over-all system design

Before we will look at different implementations of on-line filters, we must emphasize one final point of general importance: the need for a good design of the whole system. When not enough attention is given to this point, the result might be deceptive! Speed is generally the dominant requirement and one should make a careful analysis of the over-all problem to see where the bottlenecks really are and try to avoid them. The pay-off will be considerable if the total data flow is examined and not just a particular part of it. Speeding up a subroutine by a factor two or so can have a large effect, but it can equally well result in a gain of a few percent only. It depends of course on how central this subroutine is to the problem. It is relatively easy to be misled here and to form the wrong idea. Ideally one should make measurements, but this is practically impossible during the design phase of an experiment.

The importance of eliminating the bad events as quickly as possible has been shown in the preceding paragraphs. A corollary of this which merits attention is that one should not give all events a treatment which only the good ones deserve. Complete read-out and tape-writing have already been given as obvious examples of this. There are others like reformatting of the data or reconstruction of all space points or all tracks.

There are many other things that can influence the over-all speed and a good system design would include some or all of the following features:

- fast read-out system (of the order of 100 ns per word);
- selective read-out, which makes it possible to read out only those detectors which are needed for the filter calculations or to read them out before the others);
- parallel read-out. This can be very effective if one can give a meaningful treatment to the partial data;
- hardware clustering of wire hits. This can be done on the fly by hardware, while it is slow in software;
- wire numbers should be counted from zero for each plane, or from a meaningful constant. This is again easy to organize in hardware;
- inside a program one should have a good data structure, which avoids complicated address calculations (either user coded or compiler generated);
- a simple data flow inside the program, with a minimum number of intermediate stops for the data. Communication of data between parts of programs is an important factor, particularly if several programmers work on the parts. Transfer of all data from an output buffer area to an input buffer located elsewhere in memory is not the fastest way of assuring this communication;

- efficient programming.

It is very unlikely that this list is exhaustive!

3. SOFTWARE FILTERS

3.1 Software versus hardware methods

Until a year or so ago an on-line filter implemented in software was necessarily residing in the experimental minicomputer. Software methods were then superior in the flexibility they offered, but were lagging far behind in speed, compared to hardware methods. Technological progress, e.g. the introduction of bit-slice microprocessors, apparently tends to bring the two methods closer together. The bit-slices are building blocks from which programmable processors can be built, leaving a large amount of freedom to the designer as to the structure and the functions of the processor. At first sight they therefore offer the best of both worlds, but at closer inspection also here speed and flexibility are closely related. The maximum speed will be attained when the bit-slice processor is closely integrated into the experiment, interfacing directly to the read-out of the different detectors. Most likely the structure will then also be adapted to the task at hand. The consequence is that the processor can only be programmed directly in microcode by a person who knows very well all hardware details. The processor is then still "programmable", but it can hardly be said to be "flexible".

The opposite solution that the bit-slices offer is to make the final product resemble as closely as possible a mini or other computer. This emulation could go as far as producing a machine capable of executing a program originally written for another machine. This would offer full flexibility, as the programs could be written in high-level language if desired, but speed is largely lost. In fact the final product in this case is a minicomputer, comparable in speed if the same technology has been used. It would have the same sort of interface to the experiment, unless special equipment has been added.

As the choice between a hardware filter or a software filter is really one between speed and flexibility, the bit-slices have not solved the dilemma, but added another dimension to it. The reader himself will be able to appreciate this from the few examples of the use of bit-slice processors that are given below.

Making speed comparisons between soft and hardware filters in general terms is nearly impossible as it depends too strongly on the problem, the over-all system design and, for the software, the machine and the language used. Very broadly speaking, one can however say that, except for very simple problems, a dedicated minicomputer would not be able to filter more than 10-50 events per second. In most experiments the mini available is not entirely dedicated to filtering, but busy doing other things as well. One can estimate that an purely computational problems a bit-slice processor, not too different in structure from a mini, could attain about twice the mini speed, mainly due to the use of fast memory. As stated before, better speeds could be attained if the bit-slices were to be closely integrated into the experimental set-up; in this way however, the approach would be close to a hardware solution. The relatively low cost of a bit-slice micro is, however, an incentive to use more than one for filtering purposes, so that the desired speed increase might be found in multiprocessing. We will consider this aspect in more detail later. Even so, it will be difficult to reach the speed of pure hardwired processors. In a particular case (point-finding) a hardwired processor was found to process the data 80 times faster than a PDP 11/40 programmed in assembly language.

An additional remark about the so highly desirable flexibility is also in order. Flexibility, as we saw above, decreases when going from the minicomputer, via the bit-slice microprocessors to hardwired logic. Flexibility in this context is inversely proportional to the resistance of the system to changes. If the system is programmable, it is not yet necessarily flexible. The bit-slice microprocessor can be used in a way where it is not much more flexible than hardwired logic. The programmability can be very poor, when difficult microcodes must be produced. In addition some adaptations might not be possible without making changes to the hardware.

Although the hardwired processor is the loser in the flexibility race, it is an honourable loser. Real changes to the algorithm can only be made by
modifying the hardware, but a good design will try to make a number of options already available to the user. Obviously all constants and parameters used by the algorithm must be loadable by the user from the data-acquisition mini. In addition it can often be arranged that certain parameters control the flow through the process and that a considerable adjustment can be made to changing requirements.

Generally speaking, the trade-off between speed and flexibility will then lead to the following choices:

- hardwired processor, when speed is paramount and the process is known in all foreseeable detail well in advance, to allow for the time to design and construct the hardware;

- software, if speed is not important and if one expects frequent changes and adaptations to be made;

- the bit-slice microprocessor will be a good choice if one understands well enough the way it should be integrated into the environment but one cannot yet precisely predict the filter algorithm it should execute. The total effort required (design, construction, check-out, development of software tools like cross-assembler and production of user software) is comparable to the effort for producing a hardwired processor.

3.2 Example of a software filter: SPM

The Split Field Magnet facility at CERN has recently been augmented by the addition of several peripheral detectors, which play an important rôle for triggering on two sorts of events:

i) events with an electron produced around \( \theta = 90^\circ \) and with transverse momentum \( p_T \geq 500 \text{ MeV/c} \).

ii) Events with a hadron at \( \approx 45^\circ \) to accumulate high statistics for \( p_T \geq 5 \text{ GeV/c} \).

A typical event in the present set-up is shown in Fig. 6, and a block diagram of the data acquisition system in Fig. 7. On the latter we see that a PDP 11/20, marked "filter machine" has been added specifically for running the on-line filter program. A slow trigger, based on the presence of hits in wire groups lying within certain predefined "roads", makes a first selection of those events where a track of one of the types defined above is present.

The hardware in front of the filter and the data acquisition computer (ROB and multiplexer) then make a selective read-out possible in one of the two branches. Thus the filter computer only receives data for which there is a good probability of finding tracks. The purpose of the filter is precisely to find those tracks and to transmit their parameters to the data acquisition computer. A track-finding algorithm has been developed which uses linear parametrization to find candidates in projection and to match them in space. From the points found in the first \( (n - 1) \) planes, which are supposed to lie on a track, the predicted intersect with plane \( n \) is calculated from a linear form and required to fall within predefined limits:

\[
C_{2n} \leq \left| Y_n - \sum_{i=1}^{n-1} a_{in} Y_i \right| < C_{1n} \tag{13}
\]

\( Y_n \) and \( Y_i \) are measured points (not necessarily in the same projection); \( a_{in} \) are the coefficients determined beforehand by tracking high momentum tracks through the magnetic field. The field is however very inhomogeneous and the detector volume must be divided into 20 regions, each with its own set of \( a_{in} \). In addition, to predict point \( n \) the set \( a_{in} \) has elements differing from those of the set \( a_{i,n-1} \).

The track following starts from the outside of the detector going toward the centre. At the outside, tracks are generally well separated and ambiguities are avoided to a large extent. This method of track finding requires some storage space for the coefficients, but it is fast as it does not need geometrical calculations. Also the program flow is simple. The momentum of a track can be approximated at all stages using a linear form and another set of coefficients \( \beta_{in} \). The algorithm was developed for off-line analysis and then recorded in assembly language for the on-line filter. Earlier this year the data taking rate was limited to 11 events/second (due to tape-writing), but the filter saturated only at 90 events per second. The software filter spends an average of 4.7 ms per event, including the selective read-out and some reformatting of the data.

During test runs with the high \( p_T \) trigger, 10% of the events were found to have a track with \( p_T > 3 \text{ GeV/c} \). With the electron trigger, the trigger rate was 30 s\(^{-1}\) and it is expected that the filter will reduce this by a factor 2-3 so that a good match with tape-writing may be obtained.

3.3 Speeding up software

Is it possible to speed up software, without losing its inherent flexibility? The answer is of course affirmative and different methods exist, of
Fig. 6 A typical event in the present set-up of the SFM facility. Different detectors are schematically indicated.

Fig. 7 On-line computer system of SFM. Two PDP-11's are used; the on-line filter computer is the PDP-11/20 at the right.
which some have been applied. The first is very obvious and should not need to be mentioned.

3.3.1 Efficient programming

Efficient means resulting in the fastest execution. This often conflicts with other requirements like readability, maintainability, etc., of the program, but the emphasis should be placed where it needs to be. Programming the on-line filter -- or at least critical parts of it -- in assembler language is worth the effort as can be seen from the filter for the SPX.

3.3.2 Microprogramming

Some commercial minicomputers are user-microprogrammable. The Hewlett-Packard HP2100 is an example, although it is not expected that users will make large use of this facility. No particular effort has therefore been made towards making it "user-friendly". It is also considerably more difficult to microprogram this particular machine than it is to write an assembler program for it. Nevertheless, a factor of 2 or more may be gained by avoiding the storing of intermediate results, or by making use of the possibility to do certain elementary operations simultaneously with others. The gain is largest for repetitive operations requiring one or two registers for intermediate results (multiplication algorithm, for example). Microprograms have been written for critical parts of the vertex finding in a proton scattering radiography experiment.

3.3.3 Add hardwired functional units

This is an old and well-known technique: to the basic computer which 25 years ago had only the capability of adding two numbers, fixed-point multiply/divide and floating-point units have been added. The idea can be pushed further and one could add other hardwired functional units, like matrix multiply, inner product or polynomial units. Such a unit can be very fast and a mismatch between processing speed and I/O rate for such a special unit is not exceptional.

An example of a special unit which has been built to speed up a software filter is the "linear operator", connected to the SPX filter computer and also shown in Fig. 7. A block diagram of this unit is shown in Fig. 8. The unit is connected to the PDP-11 Unibus and it calculates the linear form \( \sum_{i=1}^{n} \alpha_i x_i \), including the comparison, if required. During initialization all the different sets of coefficients are stored in the coefficient store, which has 70 ns access time. The coordinates are stored in a separate memory, under control of the PDP-11. The process is then started by loading a pointer to the coefficient table. The result \( \sum_{i=1}^{n} \alpha_i x_i \) can be read back by the filter computer which then decides on the next step. Alternatively \( c_{1n} \) and \( c_{2n} \) are also retrieved from the coefficient store and the comparison made. If the outcome is positive the PDP-11 has only to load the next coordinate (in plane \( n+1 \)) following the others (from plane 1 to \( n \)) in the memory and then provide a new pointer. The linear operator is very fast: one term is added to the sum in 300-350 ns. Nevertheless, the over-all gain in the SPX filter was small when the unit was used to calculate the sum, passing the result back. Less than 20% was gained on the 4.7 ms. This is due to the fact that only about half of this time is spent in the filter algorithm and that only a fraction of the latter is spent on calculating \( \sum_{i=1}^{n} \alpha_i x_i \). In addition, in the mode described, the processing time of the unit is small compared to I/O, which takes place at PDP-11 speed. Using the other mode of operation will give an additional speed-up.
3.3.4 Multiprocessing

To speed up filter software multiprocessing should be considered. Generally speaking, two different ways may be envisaged:

- parallel processing on several events at a time. This is conceptually easy, a new event is given to a free processor for treatment. There are problems however if one wants to apply this to an on-line filter: buffer storage must be provided for the data of all events in the course of being processed. If this is not provided one cannot gain over a monoprocessor. Unfortunately the entire data must be stored and selective read-out is therefore of little or no use;

- distributed processing in several processors, each working on a part of the task for a single event. This is very attractive in principle, but not easy in practice. Dividing the task into (nearly) independent subtasks might not be easy, but the problem can be solved for certain cases. Even so, the effort will not be negligible, communication (of data, results and status) can be complex and so is the problem of synchronization. The system design will often not be easy. It is very likely however that in the coming years we will see examples of some form of distributed processing applied to on-line filtering.

An example of multiprocessing is given by experiment WA10 at present running at the SPS. It is a parallel scheme where the two subtasks are distributed in time. Five to eight processors are used (see Fig. 9) in the experiment which studies the reaction

\[
K^+ p + K^- p \rightarrow \eta^+ \eta^- + p
\]

and similar topologies.

There is no magnetic field and by measuring the directions of all tracks and the energy of the recoil proton (by time of flight) a 2-constraint fit can be made. The processors perform the complete kinematical analysis before writing events to tape. The processors DPNC 811 are home made and slightly simplified versions \(^{12}\) of the PDP-11, based on the MMI 6701 bit-slice and the Intel 3001 sequencer.

The system works as follows: during the 1 s burst events are read and stored in the DPNC 811's.

Fig. 9 Block diagram of the data acquisition system for experiment WA10. A number of small computers (DPNC 811) perform the on-line filtering between accelerator bursts. They serve also as buffer memory for the events.

This is done under control of the PDP-11/45, which directs each new event to a free processor. The data do not transit through the PDP-11/45's memory however. Upon reception of an event the DPNC 811, executing a program residing in a 4K bipolar memory, compresses the data from approximately 200 words to 100 words (mainly by packing two data words into one memory location). The compressed event is stored in a 24K MOS memory. The system therefore acts also as a large buffer memory for the events. Between bursts the DPNC 811's perform the kinematical analysis and whenever a result is available the event is transferred, under control of the PDP-11/45 again, to the IBM 1800 which does the tape-writing. Each 811 can process 120 events between bursts and with five processors an event rate of 600/burst has been obtained\(^ {13} \). It is intended to expand the system to double capacity.

3.3.5 Build your own fast processor

The last possibility for speeding up a software filter is building a programmable but fast machine. From the hardware point of view this is an attractive solution as you can

- specialize the structure to suit the particular problem;
- add special functional units with a minimum overhead in data flow;
So there is the possibility of beating every existing machine in speed. Viewed from a software standpoint, enthusiasm risks being mitigated, since programming this machine will unavoidably be awkward.

To make programming less tiresome a large effort will be needed in writing assemblers, loaders, linkers, etc., with the risk of also slowing down the machine considerably.

There are several examples of such machines implemented using MSI technology (and more if we include bit-slices; those will be treated further on). The oldest example is probably the processor originally called the "Formula Evaluator" developed at the Rutherford Laboratory. This processor centres around an arithmetic unit specialized to calculate \( y = mx + c \). In its present version (Fig. 10) other functional units have been added, including a general purpose unit based on the Am2901.

![Fig. 10 Block diagram of the Rutherford Special Purpose Processor, consisting of different special function units.](image)

The second example is a microprogrammable processor\(^{15}\) built for a muon experiment (#400) at FNAL and called the M7. It is intended for on-line filtering and specialized to calculate

\[
c = ax \pm by
\]

very fast, but with limited precision (12 bit). It is implemented in ECL and uses two parallel multipliers, while simultaneous access can be made to the four operands. In 18 successive steps, making use of the elementary operation, the transverse momentum and the effective diumon mass can be calculated:

\[
p_T^2 = \frac{K^2[(A_1 \cdot \Delta x_2 - B_1 \cdot \Delta x_1)^2 + (A_2 \cdot \Delta y_2 - B_2 \cdot \Delta y_1)^2]}{(A_3 y_2 - B_3 y_1)(A_4 y_2 - B_4 y_1)}
\]

(15)

\( K, A_1 \) and \( B_1 \) are coefficients, \( x_1, y_1, \bar{x}_1 \) and \( \bar{y}_1 \) are coordinates measured in drift chambers, and \( \Delta x_1 = \bar{x}_1 - x_1 \). The design aim was that the decision must be taken within 5 \( \mu s \). The present implementation which uses a three-stage instruction pipeline executes the program to calculate (14) and (15) in 2.1 \( \mu s \).

The last example is ESP\(^{16}\), originally developed at CERN to perform track filtering on the Erasne machines for measuring bubble chamber film. This microprogrammable processor is implemented in Schottky TTL and has a very short microcycle \((\sim 100 \text{ ns})\). Besides being used in Erasne, preparations are under way to use it in different electronic experiments. A new CPU board contains the necessary logic to make a "fuzzy compare", that is to check relations of the type

\[
|a - b| < c
\]

and to branch accordingly. Some other special function units have been designed, e.g. a bit test and shift unit\(^{17}\). In addition the interfaces to CAMAC have been improved and a buffer memory system has been developed\(^{18}\). On the other hand, the program memory (separated from the data memory) is limited in size (256 words) and programming the device is not particularly easy.

4. BIT-SLICE MICROPROCESSORS

As we have seen before, the bit-slice microprocessors form a sort of transition from the software to the hardware filters. The bit-slices are a convenient way of packaging, in a single integrated circuit, arithmetic and logic units together with register files. They do not bring fundamentally new architectural ideas to the construction of processors, be they general or special purpose. It is however easier and faster to build a processor out of bit-slices than building it out of "discrete" elements\(^{13}\), as was the case with the processors described in the foregoing section. They do also leave freedom to the designer who is sufficiently expert, so that he is not limited to making little variants to a manufacturer's standard design. Somewhat better speed performance can in general be obtained however, by using MSI elements.

When a bit-slice processor has been designed for easy programmability it is not fundamentally
different from a minicomputer as was mentioned before. For use in a filter a few differences should be noted. In this application there is no need for sophisticated peripherals nor for the execution of large programs. The price can therefore be kept rather low and this may stimulate using more than one processor in an experiment as we have already seen.

When on the other hand the programmability is kept at the level of microcode, the bit-slice processor approaches very much the philosophy of the specialized processors described before. Much more liberty can then be taken in the architecture and the processor can also be very intimately integrated with the read-out or the control of the experiment. Clearly one can go too far in this direction: when the architecture becomes too specialized for the algorithm, no real difference exists then anymore with a hardwired processor. The development effort will be large, programming cumbersome and modifications of the algorithm difficult to implement. There is another pitfall: the temptation will be great to use bit-slice processors even in those cases where a hardwired processor would be the best solution from all points of view (parts cost, development, programming effort, interfaces). For example, it would be unwise to replace hardwired cluster-logic by a microprocessor.

The next few years will undoubtedly see several applications of bit-slice microprocessors to particle physics experiments. Large efforts will probably be dedicated to developing bit-slice based systems and a large amount of practical experience will be gained. At present already a number of examples can be given of their application.

4.1 168/E

The 168/E\textsuperscript{19}, developed at SLAC makes use of the Amd 2901 bit-slice RALU, but decoding and sequencing of micro-instruction relies largely on random logic. The present version\textsuperscript{3} executes a large subset of the IBM 370 series instructions, including single precision (32-bit) floating point instructions. It therefore emulates an IBM 370/168 and programs written for this large machine can be run on the pocket version. One additional operation is required on a program which runs satisfactorily on the IBM 370/168 to prepare it for running on the 168/E: a translator transforms the sequence of 370 instructions into a sequence of 168/E micro-instructions. The microcode is executed directly by the 168/E, which does not recognize 370 code at all. The 168/E therefore does not need peripherals for program development. The powerful facilities of a large computer are used instead and only at the last stage is the program transformed and run on the 168/E.

Programmability of the 168/E is therefore excellent and the processor is well adapted for number crunching. Present plans at SLAC seem to be more directed towards using the 168/E for the full analysis of experiments than in on-line applications. The 168/E has a program memory separate from data storage. It is capable of executing typical code at a speed which is only 1.5-2 times less than the speed of the large IBM 370/168. An important factor contributing to this good performance is the fact that the direct execution of microcode suppresses the (macro)-instruction fetch and decode times.

4.2 GESPRO

GESPRO is a processor developed at Strasbourg\textsuperscript{20} and originally intended to serve as an intelligent CAMAC controller. The data acquisition computer can specify which read-out sequence to execute and GESPRO would go through all the individual steps autonomously. The intention is also to use it for data compression in experiment WA2, which studies hyperons. It is based on the Intel 3000 series and is one of the first bit-slice processors developed in Europe for particle physics. Programs, which must be written in microcode, can be developed using a cross-assembler implemented on a NORD-10.

The processor developed by Guzik for the channeling experiment at FNAL is mentioned elsewhere in these proceedings\textsuperscript{5}.

4.3 μ77

A very fast processor\textsuperscript{21}, based on the Amd 2900 series has been developed at the Ecole Polytechnique for use at CERN in an experiment to measure with high precision the total cross-section for $\pi^+ p$\textsuperscript{22}. Small angle scattering is observed, in the region of interference with Coulomb scattering. A sketch of how the differential cross-section varies with the angle $\phi$ is shown in Fig. 11. Extrapolation to $\phi = 0^\circ$ allows us to find the total cross-section (optical theorem). The events beyond the Coulomb scattering limit are retained and the others rejected. The processor is intended for this filtering problem, which is solved by using particle coordinates from 5 small wire chambers.

We are approaching more and more a specialized solution, as the read-out of these chambers is very
intimately related to the processor. A block diagram of the processor is shown in Fig. 12. The read-out and encoding of the wire planes is performed in a sort of "peripheral processor" (PP1 to PP5). The wire chambers are quite small (256 wires) and encoding is done by priority encoders in 65 ns (or 130 ns in case of double hits). The wire coordinates are stored in scratch pad memories of 16 words by

8 bits, from where they are retrieved by the processor. The process is pipelined and the calculations on event \( n \) take place simultaneously with the encoding of event \( n + 1 \). At the same time event \( n + 2 \) can be latched. The processor executes a short cycle to select events on the basis of the scattering angle:

\[
\theta_x = \sum \frac{3}{1} \alpha_1 n_1 + b \quad \text{and similarly for } \theta_y
\]

(16)

\[\theta^2 = \theta_x^2 + \theta_y^2.\]

(17)

In addition the synchronization with the trigger and the loop control is done in this short cycle, which consists of 22 micro-instructions. Since a micro-instruction requires 168 ns for execution, the total time spent in the short cycle is 3.5 us. This time has been obtained by performing the multiplication in an \( 8 \times 8 \) bit array multiplier.

The accepted events (< 10% at this stage) can be submitted to further treatment. Here we see clearly the advantage of a programmable processor. For instance \( \Delta p/p \) can be calculated for the incoming beam particle

\[
\frac{\Delta p}{p} = \sum \alpha n_1 + k
\]

(18)

so that a two-dimensional histogram (frequency of \( \theta \) and of \( \Delta p \)) can be updated and maintained. Approximately 2% of the triggers contribute an event to the histogram. The processor is capable of treating \( \approx 3 \times 10^5 \) events/second and a two-dimensional histogram of \( 10^6 \) bins is filled in about 100 bursts at the FS (\( \approx 5 \) minutes). The update of the histogram is transmitted to the minicomputer, which keeps the master histogram and prints out results. Remarkably enough, no magnetic tape is used in the experiment.

Another interesting point is the fast rejection of bad events, while some extra time is being spent on the accepted ones to refine the selection. Finally one is left with a small fraction (< 1%) of pathological events, which cause a dilemma. Should they be treated further, at the cost of slowing down everything, or should they be rejected or accepted? A valid answer can of course only be obtained after careful study of these pathological events. All in all, this experiment is a very good example, not only concerning the use of bit-slice microprocessors, but also with respect to the application of general principles of on-line filtering.
5. HARDWARE FILTERS

In the domain of hardware filters it is useful to distinguish two speed classes:
- the processor type, with speeds in the few to 100 μs range;
- the "slow trigger" or second level decision type with a decision time of the order of one to a few microseconds.

5.1 Hardwired processors

The most salient advantage of a hardwired processor is the fact that for a given algorithm we can realize the fastest design possible, within the constraints of technology and budget. In fact one can fully exploit all tricks of the trade, which are mainly
- parallelism
- pipelining
- duplication of elements
- non-sequential control.

As is well known, there is a price to pay for achieving top performance. This price is expressed in long development time, for rather bulky and expensive equipment, which is difficult to maintain. It should therefore always be carefully considered if a hardwired processor is the right solution to the problem at hand. If it is, then interesting results can be obtained, as we will see from some of the examples.

5.2 Some history: global methods

A few years ago, global methods of pattern recognition seemed very attractive and particularly suited for implementation on hardwired processors. They presented in fact severe combinatorial problems and were therefore very time consuming on a general purpose computer. The idea was to recognize all tracks in an event, relying on methods like principal components\(^2\) or nearest neighbours\(^3\). As these methods have been described elsewhere we will only recall the principle. All possible combinations of points detected in several detectors (taking one point per detector plane) are checked either against certain constraints (principal component method) or for their distance in multi-dimensional space to some standard set of track points (nearest neighbours). For \(n\) particles detected in \(m\) detector planes, \(n^m\) combinations must be checked in the absence of pre-selection criteria.

It turned out that these methods were easily implemented in hardware if one limited oneself to the basic algorithm\(^2,3\). As the time necessary was still considerable, the introduction of pre-selection criteria was desirable. Such a pre-selection could be made on the assumption that tracks in a magnetic field are approximately straight lines in one projection. Checking for straight lines can be done in a time proportional to \(n^3\), which represented an enormous gain over \(n^{16}\) or \(n^{12}\). This introduced however considerable difficulties. Very awkward complications are introduced by the fact that one has to take into account many types of special cases: for example, a missing point in a detector upsets the principal component method entirely. The result is that a hardwired processor for performing pattern recognition by such a global method becomes very complex.

A still more fundamental criticism can be made. The methods are aimed at finding all tracks. This is very good if one wants to save computer time in the analysis phase, but it is not necessarily a requirement for on-line filtering and it certainly conflicts with the principle of rejecting bad events quickly. In addition a Monte Carlo study must be made first and the results of the Monte Carlo provide the parameters used in the calculations. This does not contribute to making the methods acceptable for the physicists.

5.3 Hardwired filter processors

A number of processors have been built with the aim of providing fast event rejection. Examples are the processor for the Mark II spectrometer at SPEAR\(^4\) (described by P. Kunz in a seminar at this school), and the filter processors for experiments WA7 and WA18 at CERN.

5.3.1 Coplanarity processor

Experiment WA7\(^5\) studies two-body reactions at large \(p_T\), where the cross-section is very small. The trigger is based on a coplanarity test, performed by coincidence matrices receiving signals from scintillator hodoscopes. As the hodoscope cells are rather large (\(\approx 20 \times 20 \text{ cm}^2\)) the trigger is not very selective. A filter processor\(^6\) is used for a second level decision, made with the full precision of the wire chambers. The processor can be used in master or in slave mode. As a slave it only transmits its decision to the data acquisition computer, as a master it also resets the read-out system in case of a rejected event. The processor receives the wire chamber data selectively (the essential wire planes only) at a rate of \(\approx 120 \text{ ns per coordinate}\). It consists of three separate processors. The
first one (pre-processor) is in charge of preliminary tests on the number of hits per plane, the number of planes hit, etc. It is also in charge of communication with the read-out, the data-acquisition mini and the other two processors. Finally it performs the operations which can be done on a single coordinate, like adding or subtracting it to or from a constant in order to express all coordinates in a common reference system.

The second processor performs point-finding in three chambers sequentially. The wire chambers have \( U \) and \( V \) planes with wires at angles given by \( \sin \alpha = \pm 8/17 \) and \( \cos \alpha = 15/17 \) (see Fig. 13). The point-finder therefore checks the following relations:

\[
\begin{align*}
|15X + 8Y - 17U| &\leq \varepsilon \quad (19a) \\
|15X - 8Y - 17V| &\leq \varepsilon . \quad (19b)
\end{align*}
\]

The required pre-multiplication is also done in the pre-processor.

Fig. 13 Coordinate axes for a four-plane wire chamber (the wires are perpendicular to these axes). A correct choice of \( \alpha \) leads to formulae (19a) and (19b) which can be evaluated without needing multiplications.

The last processor checks the coplanarity of two outgoing (straight) tracks with the incoming beam and checks the opening angle. The following formulae are used:

\[
\begin{align*}
\text{COP} &= \Delta X_L \cdot \Delta Y_R - \Delta Y_L \cdot \Delta X_R + k_{\text{COP}} (\Delta Y_L - \Delta Y_R) + k_{\text{COP}} \\
\text{OPA} &= \Delta X_L \cdot \Delta X_R + \Delta Y_L \cdot \Delta Y_R + k_{\text{OPA}} 
\end{align*}
\]

(20) (21)

where \( \Delta X_L \) is the lateral displacement in the \( X \)-direction for the particle in the left arm of the spectrometer and so on. \( k_{\text{COP}}, k_{\text{COP}} \) and \( k_{\text{OPA}} \) are constants, which partly correct for the influence of the magnetic field between the target and the first detector. The coplanarity processor is the slowest. For an ideal, clean event the whole process would take no more than \( \approx 20 \) \( \mu s \), but when spurious hits or particles are present, the timing for the coplanarity test increases as \( n^2 \). The read-out by the mini-computer via CAMAC takes, however, 4 \( ms \) so the processor is fast enough except in extremely complicated cases. The processor is essential for the enrichment of the sample, as in test runs, without it, very few two-body events are found in the recorded events \( (\approx 1 \times 10^4) \). The reader interested in knowing more about the techniques used to attain the fast execution is referred to more extensive descriptions of similar processors.

5.3.2 Histogramming processor

The experiment WA18 proposes to study neutral current processes in \( v \) and \( \bar{v} \) interactions. A large part of the experimental set-up is occupied by a series of marble plates, sandwiched with counter hodoscopes. In each counter plane, perpendicular to the beam axis, there are 20 strips of scintillators. Planes with horizontal strips alternate with planes having vertical strips. There are 80 counter planes in total, i.e. 1600 scintillation counters.

About 300 primary triggers are expected during a 2 \( ms \) spill. In order to reduce the number of events a selection should be made between those events where an electron shower is generated and those where a hadron shower is present. A processor has been built which should make this selection in a very short time. The processor makes a histogram of the number of hits per vertical strip, totalled over the 40 planes (see Fig. 14). Similarly, for the 40 planes with horizontal strips. For each

Fig. 14 A schematic view of a shower in the set-up of the neutrino experiment WA18. Only two of the 40 planes with vertical counter strips are shown and the number of strips shown is 6 instead of 20. All the counter hits are projected on the end plane and histogrammed.

histogram (containing 20 bins) the maximum content of a bin is found and a parameter calculated:

\[
P_H = \frac{\text{total number of entries}}{\text{maximum}}
\]

Similarly for \( P_V \). The final parameter is

\[
P = P_H + P_V.
\]
Figure 15 gives an idea of the selection the parameter P allows to obtain. Above P = 5, 96% of the hadron showers are expected, contaminated with 1% of electron showers.

Fig. 15 Distribution of the parameter P for hadron and electron showers. A cut-off at P = 5 will select 96% of the hadron showers, contaminated by 1% of the electron showers.

A block diagram of the read-out and processing system is shown in Fig. 16. The 80 counter planes are read out by four fast systems identical to a

wire chamber read-out [3], operating in parallel. Each read-out writes into a buffer memory and simultaneously into the processor. Each word transmitted represents a 20-bit hit pattern of a plane and a 5-bit identification. When the event is rejected by the filter processor, the pointers to addresses in the buffer memories are reset to their previous values. The next event then overwrites the data of the rejected event. A block diagram of the processor itself is shown in Fig. 17. The four read-out

Fig. 17 Block diagram of the hardwired histogramming processor for experiment WA18. Note that the division necessary to determine P_H and P_V is done with look-up tables.

systems operate independently of each other and therefore FIFO memories are needed to obtain the necessary synchronization at the input of the processor. Depending on the identification number the data is routed to the horizontal or vertical histogramming unit. This summation unit adds 0, 1, 2, 3 or 4 to the contents of the different bins in the histograms. This is done simultaneously for all bins. The adders used need only 5-bit precision; the contents of a single bin is theoretically limited to 40 and in practice to 31. The maximum bin content is found and a table look-up scheme used to determine the values of P_H and P_V. This is possible since the total content is limited to 8 bits and the maximum content to 5 bits. There are therefore not more than 2^8 = 256 = 8192 possible values of P_H. The
histograms are filled on the fly during read-out of the data and this takes a maximum time of 3μs. An additional 0.5μs are needed to find P and to take the final decision.

5.4 Slow trigger processors

In the previous example a table look-up scheme was used to perform a rather complicated arithmetic operation (division) in a single memory access time, e.g. in approximately 50 ns. This table look-up technique is not only valid to replace arithmetic operations whenever the precision of the operands is limited, but can also be fruitfully applied in decision logic. In fact the technique can be used to recognize as valid or invalid any bit pattern of limited length, whatever its origin. The bit pattern under examination is used as address to access a memory location in which the correct answer has been stored previously. In case a decision valid/invalid has to be taken, the memory need only be 1 bit wide. When an arithmetic result is desired the memory can be as many bits wide as desired. A memory can even be used as a many input universal trigger logic unit\(^1\)), provided the trigger signals -- which together form a bit pattern -- are applied simultaneously and last for a time of the order of the access time of the memory. The memory has been loaded with a "1" for those input patterns on which one wants to trigger and with a "0" in all other locations. The bit patterns applied could, for instance, represent the rough shape of a track, the position of which has been determined in say 5 planes, with a 3-bit precision in each plane.

There is however a practical limit to the applicability of these methods. When the bit pattern exceeds 15 bits, the memory size becomes excessive (> 32K). There exist however, several methods to extend the possibilities, which are all based on the fact that in practical situations the vast majority of memory cells will be loaded with zeroes. For instance, in a particular case it was found that out of 32768 possible track shapes defined by 3 bits in each of 5 successive planes, less than 256 represented possible interesting tracks.

5.4.1 Extension of the input range

How can the input range be extended beyond 15 bits? Several possibilities exist:

i) store the valid codes (bit patterns) and compare with the one at hand making a binary search. This can be done by hardware, in log\(_M\) M memory cycles, where M is the total number of codes stored. The time will in general not exceed 1-2 μs.

ii) The valid codes can be stored in a content addressable memory. A single cycle is sufficient to decide on the validity of the code at hand. Unfortunately, commercially available content addressable memory (CAM) chips have very small capacity (8 × 2 bits) and are expensive.

iii) The valid codes can be stored in a Programmable Logic Array (PLA). Again a single access cycle (50 ns) is sufficient to obtain the answer. At best, PLAs exist however in field programmable versions, programmed by blowing fuses at the appropriate places in the array. Re-programming therefore means using a new -- expensive -- chip.

iv) Cascaded access. The bit pattern is broken up in a number of fields, not necessarily of the same width, each containing a fraction of the total number of bits. The first field is used to address a memory containing pointers. If none of the valid codes contain the bit pattern in the field at hand, the pointer gives an immediate "no" answer. If the field can lead to a valid result, the pointer indicates the address of a further memory block. Individual words in this block can be accessed by a new field from the original code. The word accessed will again contain a pointer and so on, until all fields have been tested. The scheme is illustrated in Fig. 18. The advantage of this cascaded access can be seen from the following: changing from a 15- to an 18-bit pattern, the memory size must be increased from 32 K to 256 K, i.e. by a factor \(2^3\) when direct access is used. If the result is obtained in three cascaded accesses, the total size of the memory need only increase by a factor two, since the extra 3 bits can be distributed over the 3 fields. Table 4 gives some examples for bit patterns of different lengths and for access in one, two and three stages. It also indicates how the bit pattern should be broken up in fields.

v) Hash-coding. This method is well known from software for data retrieval problems and can be directly applied to the recognition of bit patterns by hardware\(^2\)). The only problem resides in conflict resolution when the hashing algorithm gives the same result for two different input patterns.

The table look-up technique is very promising and in fact already applied in several instances: the "Programmable Track Selector"\(^3\)) is based on it and one experiment at CERN uses fast 1 K ECL memories in its trigger logic\(^4\)). Also the hardwired processor for the Mark II detector at SPEAR\(^5\)) uses table look-up in at least two places. Look-up tables can
be useful to replace complete formulae and not just a single arithmetic operation.

This application has been suggested for straight line finding. The memory size can be drastically reduced if the expression to be evaluated is linear. This is illustrated in Fig. 19 for the case of two independent variables. Straightforward application of the table look-up method to two 8-bit variables would require a 64 K memory (1 K = 1024). The variables can however be broken up in a most and a least significant part of 4-bit each. The expression is evaluated using the most significant parts of the variables to form an 8-bit address and similarly for the least significant parts. Since the expression is linear, the final result is the sum of the two partial results. The memory size has been reduced from 64 K to 2 × 256 words! Memories can also be used to build coincidence matrices with two, three or possibly more sets of inputs. They can also be used to encode simultaneously more than one bit in an input pattern, in contrast to a priority encoder which will give only the position of the most significant bit.

However, several of these applications would lead us astray from our subject, on-line filtering.
6. CONCLUSION

These lectures attempted to give an overview of the area of on-line filtering in high-energy physics experiments. A large range of applications has been shown as examples. They covered a wide ground, from pure software to very fast table look-up methods, showing that considerable progress has been made over the last few years. On-line filtering is becoming widely accepted now and its application to large scale experiments is more and more unavoidable.

Hopefully these lecture notes may help the reader to find the optimal solution if he is confronted with an on-line filter problem.

7. ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to the numerous colleagues who contributed to these lectures by providing me with valuable and new information. The credit for the ideas and work in most of the examples is theirs. Amongst them I particularly wish to thank Drs. E. Barrelet, F. Gagliardi, M. Letheren, G. Lütjens and A. Vriens.

References


5) P.F. Kunz, Software for microcircuit systems, in Proc. 1978 CERN School of Computing. See also:
P.F. Kunz, Microcircuits for high-energy physics, ibid.


8) A. Norton and F. Gagliardi, private communication.

9) Y. Perrin, private communication.

10) Examples of commercially available equipment are the AP-120B or AP-190L array processors of Floating Point Systems, Inc., and MAP of CSP Inc.

11) F. Gagliardi, J. Joosten and J. Hendel, private communication.


13) A. Vriens, private communication. See also:


16) T. Lingjaerde, A fast microprogrammable processor, CERN DD/75/17.

T. Lingjaerde, CMAC interface for the ESOP microprocessor, CERN DD/76/9.


21) E. Barrelet, private communication.

22) P. Baillon et al. (CERN-Clôüde de France-Ecole Polytechnique Collaboration), Status report high precision measurement of $\sqrt{s}$ total cross-section, Internal document CERN PSC/78/6.

23) H. Wind, Function parametrization, Proc. 1972 CERN School of Computing, CERN 72-21, p. 53-106. See also:


26) C. Baglin et al. (CERN-Annecy(LAPP)-Genoa- Copenhagen-Oslo-University College London Collaboration), Two body reactions at large PT, Internal document CERN/SPSC/74-28.

27) J. Joosten, M.F. Letheren and B. Martin, private communication.


29) M.F. Letheren, private communication.

30) J.B. Lindsay, C. Millerin, J.C. Tarlé, H. Verwey and H. Wendler, A fast and flexible data acquisition system for multiwire proportional chambers and other detectors; to be published in Nuclear Instrum. Methods.


32) A. de Bellefin et al., Application de l'adresse dispersé (hash-coding) à la reconnaissance de mots binaires, CERN/EP/77-1.


34) M. Conversi and C. Verkerk, Flash chambers as 'active' detectors and a special processor for fast decision making, submitted to Nuclear Instrum. Methods (also CERN DD/78/3).

35) M.F. Letheren, private communication.

MINICOMPUTER SOFTWARE
FOR HIGH-ENERGY PHYSICS EXPERIMENTS

D.M. Sendall
CERN, Geneva, Switzerland

1. INTRODUCTION

These lectures will be an introductory survey rather than a treatment in depth. I shall be concentrating on a specific application; how small computers are used in high-energy physics experiments. I shall be talking mainly about current practice in laboratories, rather than emphasizing theoretical aspects or future developments. The first two to three lectures will probably be of interest mainly to people without experience in the use of small computers in experiments; I shall try to explain the way in which they are used. For the last part of the lectures I shall discuss some of the alternative ways of tackling the problems met with in these applications.

1.1 WHY USE A MINI?

Most of you will be familiar with the advantages of using small computers (‘minis’) in instrumentation and experiments. The essential characteristics of a mini are that it is small, cheap, and easily interfaced to special equipment; usually its power is limited and it handles data in rather simple ways. In these respects it is unlike traditional larger computers, but it can easily be built into an experiment. It can then be regarded as a component of the overall data-acquisition and monitoring system, along with the other parts of the system (particle detectors, fast electronics, etc.). What it contributes is essentially flexibility in data handling: both the flow of data, and the treatment of it, can be easily modified to meet changing circumstances. This makes the mini an essential part of all high-energy physics experiments today.

1.2 WHAT SORT OF MINI?

Just to give you an idea of the sort of computer we are talking about, here are some ‘typical’ characteristics of ones in use in high-energy physics laboratories. The instruction set is rather simple, aimed at data transfer and simple tests rather than calculation. Instruction times are about 1 usec. Memory sizes are normally in the range 16 to 128k 16-bit words; this has to contain the program and work space, data buffers, and histograms and other results. Normally the mini will be interfaced to CAMAC (Ref.3), which may be the only ‘special’ I/O device there is. In addition a number of peripherals will be needed, interfaced either directly or through CAMAC.

The most important are the following:

a) A keyboard/display, so that the experimenter can communicate with the computer for control purposes, and to provide visual displays of histograms etc. on demand. It can also be used for program development (typing-in, editing etc.); I shall say more about this later.

b) A printer, usually slow and cheap, to record results, error messages etc.

c) A magnetic tape for data recording; normally experimental data is written on tape and analysed later on larger computers. In a simple test set-up the magnetic tape may not be needed, but for a real experiment it almost always is.

d) Some means of loading programs. This could be a paper tape, floppy disk or digital cassette system. A link to a large computer may be used instead; this has other uses (for program development or central data recording) as will appear later.

e) Finally, disks holding several million words, with random access times of the order of 100 msecs, are used on some large systems to hold programs, large histograms, etc.

The cost of a configuration like this will be from say 20000 to 50000 Swiss francs, depending on the design and complexity of the experiment. This is normally a rather small fraction of the total cost of the apparatus.

2. THE EXPERIMENT

A high-energy physics experiment studies the properties of elementary particles via the interactions of particle beams with stationary targets or with other beams. The acceleration of the particles, and the production and transportation of beams, are specialized fields; in both, computers are used extensively for control functions, but I shall not be talking about these. Nor shall I discuss bubble chambers, where events are recorded on film and analysed later; here too computers play a big part. We shall assume that the beams exist, and consider a so-called ‘electronic’ experiment set up around the interaction region.

Ideally, we should like to measure the mass, charge, and momentum of all the particles (some may be neutral!). In practice this is usually impossible for reasons of physics, space, cost etc. and we only measure a subset of the information, which is of most interest for a given experiment.

2.1 THE EXPERIMENTAL SETUP

A simplified block-diagram (Fig.1) shows the relation between the different parts of the apparatus. The information from the various detectors described below is recorded on magnetic tape for later analysis ‘offline’ on a big computer. This computer will do the work of reconstructing particle trajectories, computing momenta and other quantities of interest, distinguishing between different types of events, and finally coming up with the distributions of
quantities that the experiment was designed to measure. This full analysis can take a lot of time (several seconds of a powerful computer like the CDC 7600 per event).

Here we will concentrate on the 'online' aspects (up to the recording stage) and look into the components of this setup in more detail.

2.2 DETECTORS

I will mention three important classes of detectors to give you an overall picture.

a) Scintillation Counters. Here the passage of a charged particle through the counter (often a strip of plastic) causes a flash of light to be emitted. This is fed via a light-guide to a photomultiplier, causing a detectable electronic signal. This tells us that the particle passed through this counter, and also at what moment. Often arrays of scintillators known as hodoscopes are used, giving rough spatial resolution and good time resolution.

b) Wire Chambers. These are constructed of arrays of parallel electrically charged wires arranged in planes. The passage of a charged particle ionizes the gas in the chamber, and the ions drift towards the nearest wire. The point at which the particle crossed the plane can be accurately determined in one of two ways. Either we have a large number of closely spaced wires, and detect which one receives the ions (multi-wire chamber); or else we have fewer wires but we measure the time taken for the ions to reach their wire (drift-chamber). In the last case we must have a scintillation counter to tell us when the particle actually passed through. In both cases, the arrival of the ions at the wire causes a pulse which can be detected in a number of ways. These detectors give good spatial resolution but poorer time resolution.

In both (a) and (b) we are measuring points on particle trajectories. Momentum is measured by the deviation of the trajectory in a magnetic field. We can sometimes also measure velocity directly from the time of flight between two counters.

c) Cerenkov Detectors. These use the fact that a charged particle travelling faster than light in a transparent medium emits light, at an angle depending on its velocity. We can measure the velocity by measuring the angle; alternatively we can discriminate between particles of different mass but the same momentum, by arranging the density of the medium so that one travels faster than light, giving a signal, and the other does not. Thus we can distinguish between different types of particles, with good time resolution but poor space resolution.

Other types of detectors besides these 'classical' ones have been developed in recent years, but I will not go into any details here.

2.3 FAST ELECTRONICS

The information coming out of the detectors is in the form of an analogue pulses. These must be treated in various ways to give useful information; some of the most important are listed below:

a) Standardisation. Discriminators accept pulses from the detectors and for each one above a certain threshold deliver a standard logic pulse of defined amplitude and width. This is an essential first step to digital decision-making and data acquisition.

b) Selection. The number of interactions in a beam is very large, and usually only a small fraction are of the kind we wish to study. It is not practical to record everything and sort it out later; we must be selective and ask questions like 'Is there an antiproton in the final state?'. This is done by using the fast response of scintillators and Cerenkov detectors. Standardized pulses from these are fed into electronic logic which can indicate that an event of potential interest has occurred. This is called the formation of a trigger.

For instance (Fig.2), if pulses from scintillation counters S1S2S3S4 occur

---

Fig. 1 Schematic diagram of the apparatus

---

Fig. 2 Scintillators in coincidence
within a few μsec of each other, you are likely to have an event where the incoming particle interacts and an outgoing particle is produced at an angle 0.

Usually after an event has been selected by some logic of this sort, more exact spatial information is provided by the other detectors (wire chambers etc.).

Triggers in real experiments are of course much more complicated than this, and generally involve large arrays of hodoscopes or complex selection criteria based on groups of wires in wire-chambers.

c) Pulse-Height and Time Measurement. Pulse-height measurement may be important since the signal from the detector can give information on the charge and velocity of the particle. The signals can be sent to analogue-to-digital converters (ADC’s), which digitize the pulse-height and store the result in a register.

Time measurements are also needed, either for time-of-flight measurements or for drift-chamber readout. For this, time-to-digital converters (TDC’s) are available, which measure the time between two standard input pulses and store the result in a register.

d) Wire-chamber Readout. For multi-wire chambers it is not practical simply to record for each wire whether it is hit or not (there may be tens of thousands of wires in an experiment). Instead, fast circuits encode the addresses of the wires hit and store them for later readout. Again in reality complications occur, such as clusters of 'hits' on adjacent wires caused by a single particle, and these have to be dealt with in the readout process. Drift-chamber readout also presents some tricky problems in practice, and several methods exist for handling them.

2.4 CAMAC

In discussing fast electronics I have not made a clear distinction between the two boxes 'Fast Electronics' and 'CAMAC' in Fig.1. In practice the various operations can be shared between CAMAC and non-CAMAC in different ways. Usually trigger logic is done outside, ADCs and TDCs are CAMAC modules, and wire-chamber readout is a mixture!

The essential feature, however, is that all information required by the computer must be available in CAMAC modules. When this is so a final event trigger is generated and sent to the computer; this will normally be later than the first-level trigger described above, to give time for all the detectors and electronics to respond. The computer will then read the complete event from CAMAC.

Let me just remind you of the essential features of CAMAC from the computer's point of view. (Those interested in more detailed information can consult Ref.3). A CAMAC system consists of a number of modules arranged in crates. Each crate has a controller, and these controllers are either interfaced directly to the computer, or are connected together as a CAMAC branch and interfaced to the computer via a branch driver. This provides a simple way of exchanging information between the computer and the registers in the modules; each register has a unique address in the overall system. Suppose we want to read a register in subaddress 0 of station 20 in crate 3; the computer simply sends the corresponding CNAF suitably coded (where C=3, N=20, A=9, F=0) to the branch-driver, which takes care of the transfer, presenting the data from the register to the computer when it is ready. Detailed design of branch-drivers varies, but the essential features are the same. In addition a mechanism is provided whereby a module can request attention by a 'look-at-me' signal (LAM) in real time. These demands are passed on to the branch-driver, which will normally interrupt the computer in the appropriate way.

A typical CAMAC system will include:

a) pattern units, recording hits in scintillators and Cerenkovs
b) ADCs, recording pulse-heights
c) TDCs, recording time-of-flight or drift times
d) wire-chamber readout modules, giving wire addresses.

I should like to emphasize that CAMAC is an essential part of high-energy physics instrumentation. Nearly all high-energy experiments now use it, and in most of them it is the only interface between the computer and the experiment. This has brought about enormous savings in hardware design, and this is also true in the software field; I shall come back to this point later.

2.5 SOME ORDERS OF MAGNITUDE

Before going on to consider the role of the computer in more detail, I will remind you of some of the orders of magnitude involved. Experiments vary greatly in scale and complexity; a simple test may have a small wire-chamber and a few counters, requiring one CAMAC crate with a few modules (I will describe an example later). Big experiments, on the other hand, can have many chambers with tens of thousands of wires in all; there may easily be tens of CAMAC crates containing several hundred modules (see Ref.2 for examples). The amount of data collected will of course also vary, but one might take 1000 words as a 'typical' event size.

Fast electronics works on timescales of a few nanoseconds. The basic CAMAC operation cycle, and also that of typical computer memories, is 1 μsec. This governs the speed at which the data can flow into the computer; it takes a few msec to read an event. The rate at which events occur depends again on the experiment; it may vary between one event every few seconds up to rates much faster than the rate at which data could be read.
Actual operations on the data inside the computer are slower; to do anything at all with it will take at least tens of usec per word. Data can be recorded on magnetic tape at average rates of typically 20000 to 50000 words/sec; it can be sent down data-links to other computers at similar speeds.

Evidently there are some mismatches here. The fundamental rates of the particle interactions, plus the low dead-time of modern detectors and the speed of fast electronics, can produce data very quickly. However, we cannot get it into the computer at this rate. Still less can we process or even record it; often even 'offline' processing would be too lengthy and expensive to be practical. Thus we try to aim at levels of selection and reduction as the data flows through the system. Examples:

- The fast trigger itself,
- Wire-number encoding,
- Preprocessing (e.g. of clusters of wires),
- Selection and reduction in the mini,
- Selection and reduction offline.

At each stage we do more complicated tests, but on less data. Aspects of this problem are treated in Refs.1 and 2.

3. WHAT DOES THE COMPUTER DO IN THE EXPERIMENT?

Basically there are three sorts of activity:

a) organising the reading, buffering, and recording of data,
b) checking the performance of equipment,
c) setting up and controlling the experiment.

In addition, two other functions are becoming more important:

d) sending data samples for remote analysis,
e) preprocessing the data.

I will discuss each of these briefly.

3.1 DATA ACQUISITION

The most essential task of the computer is simply to read the data from CAMAC, buffer it suitably, and write it onto magnetic tape. The computer must respond to some signal (trigger) indicating that an event has occurred, and issue the appropriate CAMAC commands (CMAF's) to read the data into memory; then when sufficient data has been accumulated it writes a record to magnetic tape.

Even in this simple activity the flexibility of the computer is useful. We may want to change the details of the readout (adding new chambers etc.) or the length or format of the magnetic tape records. In fact we often want to run in many different ways in the same experiment, in order to test different parts of the apparatus, calibrate, etc., as well as take real data. The computer allows us to change conditions easily by just typing on a keyboard, or at worst loading a different program.

Let me just remind you of why buffering is necessary; it allows us not only to smooth out the random (Poisson) arrival of events, but also large-scale irregularities like the duty cycle of an accelerator. A proton-synchrotron takes several seconds to accelerate particles, followed by a short burst (perhaps 1/2 second) when they are ejected to provide beams for experiments. The data must all be collected during this short burst, but it cannot usually be processed or recorded in this time. Thus we may need to store a lot of data in order to make use of the time between bursts to deal with it. In addition, the natural unit of data-collection (event) is often not well matched to the physics needs of the experiment (e.g. on magnetic tape); it may be too small or too large, but in general we shall need to store the data in order to reorganize it into suitable blocks for recording.

3.2 PERFORMANCE CHECKING

This tends to fall into two rather distinct phases; the distinction will be important when software is discussed later. First comes the testing and checking out of equipment whilst the apparatus is being set up. There the computer will not necessarily do any data recording, but is simply used as a flexible test system, for anything from checking the CAMAC itself for electronics faults, to full-scale measurements of wire-chamber efficiencies. The second phase is the monitoring of performance during the experiment itself when data is being taken. Again, this can be at many levels, from periodic checks of power-supply voltages to rough calculations of natural unit of recording (e.g. power supply, high voltage, chamber efficiency, etc.) as the experiment progresses.

Experiments today can be very complex, and accelerator time is expensive and tightly scheduled. In addition data tapes are often taken away and analysed at other labs. Thus it is essential to check that everything is working properly and that the data makes sense; errors and malfunctions must be detected as soon as possible.

An example of what might be monitored is multi-wire chamber performance. Whenever the scintillation counters indicate that a particle has passed through a particular chamber, we record whether the chamber gave a signal or not; this gives a continuous check of efficiency. In addition, histograms of wires hit may be built up and displayed on request to show up dead and noisy wires (Fig.3). The distribution of clusters, where a single particle gives rise to multiple signals from adjacent wires, can also be monitored. All this is invaluable to check on chamber performance; it allows problems with gas supply, high-voltage supply, amplifiers, etc., to be detected and corrected.

Checking of this sort will normally be done on a sample of data as it flows through the computer. Data-acquisition and recording will usually take precedence, with checks done in whatever time the computer has available. Normally a substantial fraction of the computer time will be used on checks of this sort. Notice again how
flexibility is important. We may run a general checking program, and then if we get suspicious of some piece of the apparatus we may choose a different program to look more thoroughly at that particular problem.

Histogramming is a typical activity in on-line monitoring, providing quickly understood visual checks for the physicist. Another useful visual technique is event displays; here the apparatus is actually drawn on the screen (chamber planes, hodoscope counters etc.) and the hits in the various detectors marked. This provides information at a glance about detector performance, beams, background etc. (Fig. 4 shows an example of this sort of output.) The importance of these visual, interactive checks is the reason why all experiments have some kind of display, usually a VDU terminal with graphics facilities and keyboard.

Finally, apart from providing useful information at the time, the computer can record performance statistics etc. directly on tape for future use.

3.3 RUN SET-UP AND CONTROL

Data-taking is usually organized into runs, each a few hours long, which may differ in the detectors in use, trigger conditions, magnet currents, beam energy and many other parameters. Traditionally, these tended to be set by hand and noted in log-books; the possibility of mistakes and confusion was considerable. The computer provides a convenient way for the physicist to set things up from the keyboard. The run conditions can then be automatically recorded on tape, and also displayed on request at any time. This assumes in some cases that special hardware exists to provide computer control of the apparatus. For instance CAMAC modules exist to allow
the computer to set up delays in fast electronics, or to program the fast trigger logic. Often too the experimental computer can communicate with the beam and accelerator control computers via hardware links; it can then record or display information on beam conditions, or in some cases request changes directly. All these general control functions are becoming more important as experiments become more complex.

3.4 SENDING DATA SAMPLES FOR REMOTE ANALYSIS

Although the mini can make checks on the functioning of the apparatus, it is not powerful enough to analyse the events completely. Normally the programs to do this run 'off line' on big computers, analysing magnetically tapes written by the mini. However, if the mini has a data-link to the computer centre so that it can communicate with the big computer, it can send a sample of events and have them analysed. The physicist then asks for results at his terminal without leaving the experiment. This can provide a lot of information which the mini alone is not powerful enough to give.

In some laboratories, the mini sends all the data direct to the computer centre over data-links. It is then recorded centrally rather than on a magnetic tape at the mini itself. In such a system the amount of checking done centrally may be much greater than with the approach we have been describing.

3.5 PREPROCESSING THE DATA

Normally, as we have seen, the computer records the raw data on tape, and in parallel checks a sample of it. In some applications, however, it may preprocess the data before it is written on tape. Two examples of this kind of activity are data reduction and on-line event filtering. In the first, the computer may be able to re-format and compress the data by leaving out redundant information; this can result in important saving of space and reduction of tape-handling costs later. In on-line filtering the computer may decide whether or not a particular event is interesting; if not it is thrown away and not recorded. It may be more economical and simpler to do this on-line rather than later. This can be seen as an extension of the trigger logic; a final stage of "software trigger" allows more complex (but slower) checks to be made than is possible with hardwired logic. All these preprocessing functions demand more computing power in the mini, and complicate the programs, but are becoming more important as experiments get more complex (Ref.2).

4. HOW IS THE COMPUTER USED?

So far I have tried to indicate what sort of computer you might expect to find in a high-energy physics experiment and what it is likely to be doing there. Now I should like to go into some aspects in more detail to give you an idea of how it is used.

4.1 EXAMPLE OF A PROGRAM

Let us consider a concrete case. Suppose we want to study spatial resolution in a simple drift chamber. We set up a series of three chambers with one wire each in a test beam, and arrange a simple trigger counter set-up (Fig.5). You will remember the principle of the read-out: each chamber produces a signal a certain time after the particle has passed, depending on its distance from the wire. These signals start clocks in TDC units which are all stopped together by a delayed trigger signal; at this moment all the information required is available. We want to know how accurately the readings from the chamber lie on a straight line.

![Fig. 5 Layout of drift-chamber resolution test](attachment:layout.png)

Now you will remember that the idea is that all the required information ends up tidily in CAMAC units ready to be read by CNAP's. Let us see how that might be done here. We will assume a TDC unit containing three independent TDC's; it would have four front-panel inputs (Fig.6). Into three we feed the signals from the chambers, suitably shaped, and into the fourth the trigger delayed by a time longer than the longest possible drift time. This acts as a common stop, and we will assume that it also sets a LAM to indicate that the unit is ready to be read; this is the final event trigger as far as the computer is concerned. So our CAMAC

![Fig. 6 Electronics for drift-chamber test](attachment:electronics.png)
system is very simple; there is an empty crate except for our TDC unit at one end in station 1 say, and the crate controller at the other, interfaced to the computer. (In reality there would be a bit more logic than this, to stop new triggers from arriving whilst we were dealing with the current one.)

The essentials are that at the moment the LAM is put up, the three time values are available in CAMAC registers. How can we deal with this in the computer? To simplify things, let us forget about interrupts for the moment and assume that the computer will test whether the LAM is up; when it is, the numbers are read in, the calculation done, and everything reset. How might this look?

Let us work in FORTRAN which you probably all know, and assume that someone has provided us with a subroutine that sends a CNAF; it looks like this:

```
CALL CAMAC (C,N,A,F,D,Q)
```

We specify the CNAF, and if it is a read or write function a data word gets read into or written from the variable D. The Q response is available if we are interested. Then we can write a small program like this: (It is fairly obvious from the annotation how it works).

```
C WAIT FOR LAM INDICATING DATA READY
 1 CALL CAMAC (1,1,0,8,6,D,Q)
  IF (Q.EQ.0) GO TO 1
C NOW READ THE THREE TDC CHANNELS
  CALL CAMAC (1,1,0,T1,D)
  CALL CAMAC (1,1,0,T2,D)
  CALL CAMAC (1,1,2,T3,D)
C CONVERT TO CENTIMETRES
  X1=SCALE*(TMAX-T1)
  X2=SCALE*(TMAX-T2)
  X3=SCALE*(TMAX-T3)
C CALCULATE DEVIATION FROM STRAIGHT LINE
  DEV = (X1+F)*X1 + (X2+F)*X2
C HISTOGRAM IT IN HISTOGRAM NUMBER ONE
  CALL HIST (1,DEV)
C RESET LAM AND CYCLE FOR MORE DATA
  CALL CAMAC (1,1,0,18,D,Q)
  GO TO 1
```

This, of course, is only the central loop of the program. The real program would include a setting-up phase to initialize the CAMAC and histograms, and some kind of mechanism for outputting the histograms; it would also include more checks for things going wrong. Notice, however, the simplicity of the program; it would be easy to change the set-up, include more wires, histogram different regions of the chambers, etc., and you can quickly appreciate how powerful a tool the computer can be.

This simple example illustrates several important points:

- a) direct access to CAMAC modules from high-level language
- b) use of standard library facilities (histogramming etc.)

Notice how by providing these two facilities we can already allow the physicist to cope with a wide range of problems on his own, using straightforward FORTRAN programs.

4.2 MODIFICATIONS OF SIMPLE APPROACH

In practice there are two important ways in which this simple scheme has to be modified in reality (although you could use it more or less as its stands in this simple case). To see what they are we can start from two questions you may already be asking yourselves about the example. First, how do we write the program, get it into the computer, and test it? Second, what about recording the data? The example does not do any recording, it just analyses the data at the time. The approach to these questions depends a lot on circumstances; again the distinction between a test set-up (which has to be flexible) and a real run (which has to be efficient in collecting and recording data) is important.

4.3 PROBLEMS OF TEST SET-UPS

Let us consider a test set-up, say, to check out some new chamber. We are not worried about speed or data recording, so a program like the example will do very well. On the other hand, we want to change our minds a lot and build in various different tests as our understanding of the system develops.

If you think back to working on a large computer, you will see that program preparation involves essentially three elements:

- a) storage for the source;
- b) editing facilities for the source;
- c) compilers, loaders, etc.

On more traditional systems the source is stored on cards and editing is done off-line. This approach is not practical for minis since punched card equipment is more expensive than the average mini, as well as being unreliable, noisy, large and power-consuming.

On more sophisticated systems, sources are stored on disk files and edited interactively from terminals. The problem with this approach is that to provide each mini with its own disk is expensive. It has other disadvantages too: organizing library updates and archiving user files for a large number of separate systems is not easy.

4.4 INTERPRETERS

One way out for test set-ups is to use an interpreter. The most well known is BASIC (Ref.5); this is a FORTRAN-like language, but with every statement numbered. As each statement is typed its syntax is checked, and mistakes can be corrected immediately. Each correct statement is stored in the computer memory in source form, errors found at execution time can quickly be corrected; typing a new line will simply add it in the appropriate place, deleting any old line which may exist with the same number.

This is a very easy system to use, much appreciated especially by people with little
computing experience. By providing straightforward CAMAC routines like those in the example, a powerful tool is immediately available to the physicist or engineer. The interpreter itself can be on paper tape or cassette since it is not necessary to read it in very often; the individual programs can also be stored on these cheap but slow media since this is essentially only needed for archiving. It is a very popular solution in many environments, but it has some disadvantages which we shall return to later. In particular it is slow, because statements are interpreted every time they are obeyed.

4.5 LINKS TO LARGE COMPUTERS

Another approach is to have a link between the small computer and a large one. The file and editing system of the large computer is then used to prepare programs, and they are loaded down the data-link into the mini. This is an attractive system in many ways especially if you have an interactive terminal for the large system at your mini. It also centralizes libraries and other facilities, avoiding unnecessary duplication and simplifying book-keeping and management. The cost of links is not prohibitive (though cable costs may be important) and the central facilities are shared by large numbers of users so their cost is spread out. In addition, once such links exist they can be used for other purposes, such as passing samples of data for analysis on the large computer, recording them centrally, etc. This kind of approach leads to the establishment of computer networks offering many facilities to the mini user (Ref.14,19). A problem is that cross-compilers etc. must be provided, and that the system relies heavily on a single critical installation, so overall reliability may suffer.

With a set-up like this the program preparation facilities of the host computer can be used. Often this has an interactive system; the user can then sit at a terminal at his experiment and edit, compile, etc., on the host. Having prepared the programs, the host makes up a tape-load which can then be sent down the link into the mini, which is then ready to start work.

I have spent some time on the problems of program development, but this illustrates an important point. Once you have a computer in your system you must start to take seriously the problems of programming and provide adequate facilities and manpower; hardware costs are only a small part of the total cost of a small computer system.

4.6 REAL DATA ACQUISITION SYSTEMS

Let us leave program development problems now and turn to the other question: what about data recording? Our simple example left it out completely. We could just add some routines allowing the programmer to write on magnetic tape, and that would be adequate for a test setup. But in fact we have a more serious problem. If you look back to the rates of data-taking in real experiments, you will see that the computer is too slow to handle them by the kind of program I have been describing, even if it is efficiently coded rather than interpreted. In particular for a normal experiment:

a) The data must be buffered to avoid dead-time and inefficient use of the computer. (In the example, the program idled when there was no data, and once the data was read it was analysed immediately, maybe missing the next event if it arrived before the computer was ready to read again. This gets even worse if we add magnetic tape writing.)

b) The computer cannot normally monitor every event in real time, but must be satisfied with recording them all and only analysing a sample.

c) The computer cannot normally even input and output the data directly by program; it must set up hardware-controlled data-transfers, which can then operate more or less independently.

Consider the first two points. We wish to buffer the data; all the buffer organisation can be made more or less the same for all experiments and in any case we do not want it cluttering up the monitoring program. In addition we really want to separate data acquisition from monitoring. Now is this done?

We do it by removing the CAMAC handling (which I had just told you was so powerful) from the program and putting it somewhere else. Essentially we have two programs, one organizing the data flow from CAMAC to buffers and out onto magnetic tape, and the other taking data from the buffers and monitoring it. We can, in fact, extend this to have several different monitoring programs if we like.

You will notice that this is what is usually called a multi-tasking system; various tasks operate more or less independently of one another. For instance, one may read CAMAC into the buffers, one may write buffers onto magnetic tape, one may analyse data. Each of these processes can go on more or less independently of the others; it depends only on the availability of buffers and the state of the external equipment. Essentially we want to adopt an interrupt-driven approach.

Let me remind you how interrupt-driven systems work by a simple example. Suppose you want to output a message to a teletype, you call some system routine (by FORTRAN WRITE, for instance) and then you can carry on with calculations without having to wait for the message to be physically output. What happens? The system routine (device "driver" or "handler") saves the message, sends the first character to the teletype, and returns to the main program which gets on with its work. When the teletype has printed the character (maybe 10 ms later, enough time for approx. 10,000 executed instructions!) it sends a signal to the computer, which interrupts what it is doing at the time, and goes to a routine which gets the next character and sends it out; then it carries on as before. Thus calculation and output proceed in parallel.
and no time is lost. How does this apply to our CAMAC system?

Clearly, if we can make the trigger LAM cause an interrupt we can then read the data immediately, whatever the computer was doing at the time. Thus analysis would normally operate as a "background" task; when a LAM appears, a separate data acquisition task reads data from CAMAC into the buffer. In a similar way when a record has been written to magnetic tape an interrupt causes the next one to be fetched from the buffer and output.

Evidently, the details of communication between these tasks can be quite complex, but the principle is easy to grasp. What would our simple monitoring program look like now? Here it is:

```c
DIMENSION DATA(3)

C GET NEXT EVENT FROM BUFFER
1 CALL GETEV (DATA,IFLAG)
GETEV (IFLAG,ED) GO TO 1

C CONVERT READINGS TO CENTIMETRES
X1 = SCALE*(TMAX-DATA(1))
X2 = SCALE*(TMAX-DATA(2))
X3 = SCALE*(TMAX-DATA(3))

C CALCULATE DEVIATION, HISTOGRAM
DEV = (B*X1 + A*X3) / (A + B) - X2
CALL HIST (1,DEV)

C CYCLE FOR MORE DATA
GO TO 1
```

The program now loops if there is no data in the buffers, otherwise it gets the three words and analyses them. The routine GETEV is responsible for seeing whether there is any data. Meanwhile, new events flow in independently: when each LAM arrives it immediately causes the series of CNAFs to be sent out, and the data is put into the buffer and the pointers updated. We do not bother about this in our program. Of course the CNAFs to be sent out when the LAM is seen must still be specified somewhere; normally a list of them is provided data acquisition program to use. Clearly, this must correspond in structure to the event as analysed by the monitoring program; it is set up beforehand and incorporated in the data acquisition program for the experiment. In our example it might look like this:

```
LAM1: OPEN DMA
F0 1,1,0
F0 1,1,1
F0 1,1,2
F10 1,1,0
END
```

4.7 DMA (DIRECT MEMORY ACCESS) TRANSFERS

Let us go on to the third speed problem we mentioned. Since even interrupt handling takes time, we do not want to interrupt the computer for each word of data to be transferred in or out. A direct memory access facility is thus provided where the transfer of a block of data directly into memory is organized by hardware. The transfer is set up by program, and a memory address and word count specified. After that each data word is transferred directly into memory without the program being aware of it; the address and word count are stepped each time. This is often called cycle-stealing. Normally when the word count is empty a normal interrupt occurs (Ref.1).

This method can be used to write a record to magnetic tape, the controller hardware getting each word from memory as required. The program merely sets up the buffer address and length in the DMA and starts the transfer; then when the controller has finished writing the record and is ready for the next, it gives an interrupt. Similarly, in the case of CAMAC the buffer address and word count can be set up ready for the next event and then as the CNAFs are sent out the data flows automatically into memory. (Suppose it takes 100 usec to handle an interrupt, then if we interrupted the computer every word of a 1000 word record we use up 100 msec, but if we use DMA we use only about 1200 usec, assuming that one word can be transferred every 1 usec, which is a little optimistic in practice.

4.8 BLOCK TRANSFERS

In the case of magnetic tape, the situation is simple: the controller just has to fetch n words in sequence. It is written on a tape (or a cassette cartridge controller). The computer only sets up the sequence, sets up the DMA, and starts the transfer; the data then flows in automatically, whilst the computer gets on with something else. There are various ways of organizing such transfers; often the need arises for sequences of similar CNAF's (Our example above for instance had a short sequence of this sort.)

Let us take an example. Suppose we have a crate full of scalers; we want to send in turn

```
Cl,N1,A9,F0
Cl,N1,A1,F0
............
```

and so on, stepping A and then N until the whole crate is read in. The branch driver can generate this crate scan; the computer tells it the first CNAF, and usually the address to stop at, and starts it off. It then performs the whole sequence, sending a data word to the computer each time it gets Q=1. Normally it will signal the end of the sequence to the computer.

Similarly, if a given module generates a lot of data (MWPC readout, for instance), it may be necessary to automatically generate the same CNAF a number of times until a pre-arranged signal from the module indicates that no more data is available. Several of these repeat modes are officially defined in the CAMAC standards (Ref.3), and modules using them are readily available. Nowadays, people are even building microprocessors into branch drivers or crate controllers to extend these facilities.
5. SOFTWARE SUPPORT FOR EXPERIMENTAL TEAMS

I should like to discuss now some of the problems involved in supporting high-energy physics experiments in a large laboratory. Let me start with some background information to give you some idea of the problem. Remember that I am talking only about the application of small computers to experimental data-acquisition and related activities, not accelerator control or other applications. At CERN, there are about 100 minis in use either at experiments, or for testing equipment in beams or electronics labs. Most of these are Hewlett Packard 2100, PDP-11, or Nord-10 computers. There are other machines of various types belonging to visitors and not supported by CERN programmers. This is not an unusual situation; high-energy physics labs tend to be large, as they are built round expensive accelerators, and normally a wide variety of experiments and other activities will be going on at once.

Not only is the number of systems rather large, but the situation is dynamic. New experiments are always being set up, test beams may become available at short notice, problems with electronic equipment may need quick response. Demand for systems may therefore arise at short notice, and they are sometimes needed for only a few weeks; it is therefore essential to preserve flexibility in adapting the systems to new needs.

There are of course different approaches to this problem, although they share common features; it is one that has been tackled in a number of high-energy laboratories, and I shall say something later about the different approaches open. First however I will try to outline some of the essential features of the problem, with perhaps some bias towards CERN experience.

5.1 THE PROBLEM

What is needed, then, is to support a large number of systems with a reasonable number of programmers, in a way which is flexible enough to respond to new needs. We must be able to provide simply and quickly a data-acquisition system for any experiment or test set-up. This is done by making the hardware and software MODULAR; when a new experiment comes along, a preliminary discussion establishes what the needs are. A hardware configuration suitable for that experiment can then be put together from a pool of computers and peripherals. The software modules needed for a simple system can be provided immediately.

If the hardware is available, a physicist can often be working (reading data and recording it) in a few days after the first discussion. This simple system can then be added to and refined as experience is gained. Since the apparatus usually grows, the physicists appreciate the chance to get something working quickly in an early phase of the experiment; they can use it to test parts of the equipment whilst working on a more elaborate system for the final running conditions.

5.2 THE SOLUTION: STANDARDIZATION

This aim is achieved by standardization in two areas, hardware and software. The technical details of the solution used are less important than the basic ideas behind the approach.

In the case of hardware we standardize in two areas: the use of CAMAC for interfacing, and the use of the same computer by all groups. The use of CAMAC is extremely important; without it the kind of service I am describing would not be possible. It removes the need to design special interfaces for a new experiment (this can take many months); from our point of view it must not be forgotten that the programming effort needed to cope with a new or special interface is considerable. In addition, it is difficult to build up the experience of engineers which allows good reliability and fast troubleshooting if every system has different interfaces. Another important advantage of CAMAC is that it defines an interface between software and hardware where the programmer and the physicist or engineer are forced to talk about the problem in the same terms. This is very valuable as experience builds up in the laboratory both among physicists and programmers and it makes communication easier.

In addition to using CAMAC, all the physics groups should use the same type of computer where possible. Software is almost never transportable from one type of computer to another, especially where input and output are concerned. So we standardize to avoid such troubles many times, and again to allow experience to build up. Which computer you use is not very important; there are many which would do the job equally well provided some reasonable criteria are satisfied. Of course, not everyone has the same configuration; some experiments need more memory or peripherals than others. It is therefore useful if the computer itself is reasonably modular. A range of peripherals must also be chosen which can be offered, with standard interfaces and software, to groups needing them.

5.3 STANDARD SOFTWARE

Standard hardware is a fairly obvious idea, but what exactly is standard software? Usually all the same operating system should be used everywhere; this forms a framework within which the other software works. I shall say more about this later; we can go farther than this however. Essentially, we try to separate those problems which are common to all experiments from those which are special to a particular experiment. The common problems are solved once and for all by the support group of programmers, leaving the physicists free to
concentrate on the essential features of their experiment.

The criteria for a good flexible test system and an efficient data-taking system are very different, as has already been explained. The result of these conflicting requirements is that at most labs two different systems have been provided aimed at optimising the two extreme cases. For tests, an interpreter like BASIC with some CAMAC calls is generally used. The work involved in providing this is not too great, especially if the computer manufacturer (or someone else) provides the interpreter. All that has to be done then is to provide a way to drive the CAMAC interface from the BASIC program; I shall say a bit more about this later.

The data-acquisition system proper is a little more complex, so I will say more about it. What are the problems common to all experiments that we can solve? They include dealing with the details of input, output, buffering; responding to CAMAC and other interrupts, writing events to magnetic tape etc. In short all the essentials of data-acquisition and recording. In addition it includes a number of utility routines like histogramming and graphic display packages which are available for the physicist when he wants them. The actual monitoring of the data, on the other hand, is done by the physicist. For very large experiments the physics group may in fact need help in writing the monitoring programs, but the essential separation of the problem still stands.

The program in the computer thus consists of a mixture of modules provided by the support team and modules provided by the physics group. The physicist does not bother about details of input, output, interrupt handling, buffering and other technicalities. On the other hand, tasks involving the data itself are left to the physicist.

5.4 PROGRAM MODULES IN THE COMPUTER MEMORY

To show how things work in practice let us look at the contents of the computer memory in a very simple system. A typical memory map might look like (Fig.7). The data-acquisition system, provided by the support group, performs all the data-acquisition functions. It responds to event triggers (usually CAMAC LAMs), reads events into buffers, and writes them onto magnetic tape when the buffers are full. It is driven by the event trigger and can run more or less independently of what the monitor is doing.

The monitor program is written, typically in FORTRAN, by the physicist; I showed you an example earlier. It has a simple interface to the data-acquisition package via a few FORTRAN calls. (The most important are those to request an event for analysis, and to start and stop the run). Once the physicist has got the event he can do whatever he likes with it; normally he will collect at least some statistics, involving a few histograms. This is where the library routines come in. They are also provided by the support group and can be called in FORTRAN by the monitor program.

![Fig. 7 Memory map of simple data-acquisition system](image)

Notice that here the control of the experiment is left to the monitor; it can start and stop runs, display histograms etc., as it likes. A certain amount of standard dialogue is provided by the data-acquisition package for setting up run parameters, but this can be bypassed. Usually the support group provides a simple framework to get started, and the physicist carries on from there.

There is one more component of the system not yet described; that is the "CAMAC descriptor". This is the list of CAMAC operations to be performed by the data-acquisition package in response to various LAMs. Like the monitor it is experiment-dependent; usually it is set up in collaboration between the physicist and the support group at the start of the experiment and forms a module of the data-acquisition package. Normally it does not change as often as the monitor, which develops as physicists' ideas change. A simple example was given above.

The list is incorporated into the data-acquisition package; every time LAM 1 appears, the commands are executed. The first command turns on the DMA. Then the CNAFs to read the TDCs and clear the LAM follow. The end of the list is then signalled. Any data read goes automatically into the buffers and is available to the monitor by a GE7EVT call. Real descriptors are usually more complicated, with many different triggers and long sequences of CAMAC commands for each; special subroutines can be inserted if necessary. In the descriptor module we also define the number and size of the data buffers and any other experiment-dependent information needed by the data-acquisition package.

5.5 OTHER FACILITIES

Remember that the data-acquisition and monitoring go on independently. The data-acquisition task responds to interrupts generated by CAMAC (to read data) and by the
magnetic tape hardware controller (to write data). These operations have highest priority. The rest of the time is used for monitoring. In some circumstances it may be necessary to monitor every event. This is an option that can be selected at the start of a run; in this mode every event must not only be written to tape but is kept in the buffer until the monitor program has asked for it; only then can it be overwritten by new data.

Another useful facility is the ability to "replay" magnetic tapes. The monitor program is unchanged, but in this mode (again selected for the run) the data-acquisition package reads events from a previously recorded tape and passes them to the monitor via the GETEV call. Many other facilities are also needed in real systems, especially in large ones, but these are the most essential.

5. IMPLEMENTATION AND DESIGN CHOICES

I hope that by now I have explained the sort of software which is required for small computers in high-energy physics experiments. This type of software has been developed in a number of different laboratories (Refs.11-13). It has been developed in various forms for the EMR6130, HP2180, NORD-10 and PDP-11 computers. Examples of standardized support packages elsewhere are POLM at Pernilab, and O at Los Alamos (both for PDP-11's) and the Daresbury Data Acquisition system using small front-end computers linked to the IBM computer. In these implementations there are a number of design choices were made, and often different approaches taken. In what follows I should like to outline some of the options and the arguments for the various approaches.

6.1 MEMORY

Before going on to consider the various design choices open to the implementers of online systems, I should like to draw your attention to a general problem. Everyone familiar with minis knows it only too well, and it is probably the biggest single difficulty they have to face; the problem is memory space. There are two aspects of this, the total physical memory available on the system, and the logical address space; the latter is usually the most severe problem. The essential difficulty lies in the 16-bit wordlength of most minis. This is quite adequate for most data-transfer and manipulation, and often for calculations since multiple-precision operations are commonly available. What it makes very difficult is to design the architecture of the computer in a sensible way to deal with large programs or data areas.

The first problem for the computer designer is to get the instruction code, operand address, and various mode bits etc. into a 16-bit word. It cannot be done, so he has to rely on some combination of indirect addressing, indexing, paging etc. which allows a full 16 (or sometimes 15) bit effective operand address to be built up for all instructions which need it. Thus each instruction can 'see' a 64k (or 32k) space. This was the maximum physical memory size on early versions of most minis. Soon this became inadequate, and 'memory-mapping' was introduced; this uses a set of hardware registers to translate the 16-bit address into a longer one which allows access to a subset of a larger memory. This allows big minis to be built, but the essential problem usually remains that the task is restricted to 64k or 32k 16-bit words, which is not a big program.

This causes a lot of trouble; for instance to handle big buffers requires dynamic mapping logic with corresponding overheads and complications. Again, big programs must be overlaid or split into subtasks, with corresponding loss of efficiency and clarity. Often high-level languages cannot be used because of the referencing in a high-level language to dominate the work of the programmer directly or indirectly, to a greater extent sometimes than the real problem he is trying to solve. It will often influence the design of software, but on the whole it is something we have to live with. Still, I should like to suggest that for all except the simplest problems, a 16-bit computer is inadequate! Alternately the problem can be cut down to fit, and the rest of it solved elsewhere; I shall come back to this point later.

6.2 LANGUAGES AND PROGRAMMING SYSTEMS.

You will remember that one of the basic practical distinctions was between test setups and running experiments. In the former case it is important to let the user have as direct interaction with the CAMAC as possible; ease and flexibility of programming is important whereas efficiency often is not, and the typical problem is rather small. In the second case the CAMAC readout sequence tends to be more stable, efficiency is important, and programs to analyse the data itself become important; they can often grow to a considerable size. As I mentioned before, most people have tended to supply two different systems, aimed at the two ends of the spectrum, since the needs are so different.

6.2.1 SYSTEMS FOR TEST SETUPS

For the test setup an Interpreter is commonly used; its essential features were described earlier. Its attraction is the unification in a coherent language of all the system functions including editing, running, debugging, file-management, and the application program itself, combined with a user interface that is easy to learn and to apply (Ref.4). The most commonly used interpreter is probably some form of BASIC. The language is rather like FORTRAN in appearance, but it is limited in some respects. Usually it has a fixed number of variables of only one type, and a single scope. Using it one can access the CAMAC fairly easily via calls to input-output routines which communicate with the CAMAC interface (see eg. Ref.5). Since flexibility is important, attempts have been made to introduce new features into the language to facilitate this.

For instance, a standard for real-time BASIC now exists (Ref.4), which defines new statements for input-output, interrupt
handling, and declarations. The essential extension is the declaration of 'process-variables', which allow the program to refer directly to CAMAC registers. For instance if we have in our declaration part:

```
DEC A PRO CAMAC (1,2,10,15) (F0)
```

Then in the executable part of our program the statement B=A, where B is a normal BASIC variable, will cause the register at subaddress 15 in the module at Branch 1, Crate 2, Station 10 to be read with the function F0, and the result put in B.

Simple extensions also allow arrays and block-transfers to be dealt with. The values of Q and X following any CAMAC command are available for testing via special variables. In addition it is possible to define real-time 'events' corresponding to LAM's, and to interrogate them by WHEN statements, causing LAM-handling routines to be entered in real-time.

Similar features are provided for example in the CATY (Ref.7) and NODAL (Ref.8) systems; examples are shown in Table 1 for comparison. In general it can be seen that the extensions are of a rather simple kind, and the additional power and readability over explicit function or subroutine calls is not perhaps very great.

An important feature of most such systems is that error checks are normally built into the CAMAC routines: the overheads introduced are not excessive, and the checks are useful in the situations where BASIC is used. Contrast the descriptor in the example above, where for instance no check of Q or X is done, with the one in Table 1 it is trying to read!

6.2.2 SHORTCOMINGS OF INTERPRETERS

Interpreters are very convenient, but we cannot use them for everything. There are two major reasons for this. One is that they are normally slow, typically at least an order of magnitude slower than the execution of compiled code. This can be got around by compiling the whole program when the RUN command is typed, rather than line by line at execution time, but this can be expensive in memory since compiler, source and compiled code are all present at once. A more serious problem is that simple languages like BASIC are not suitable for larger programs; this is due to the restricted number of variables and variable types, and especially to the lack of any subroutine or procedure structure. A system allowing great ease of change and no way to structure the program is not what one wants for large programs; modern thinking agrees that these should be properly planned as a whole rather than growing organically.

6.2.3 LANGUAGES FOR DATA-AQUISITION SYSTEMS

Here the problem typically splits into two; the writing of the essential software for CAMAC access, buffer handling, communication etc., and the writing of the data-monitoring tasks themselves. The data-acquisition system proper must be efficient and handle input-output and communications with the operating system. It can be written in any language appropriate for this type of work; usually it is done in assembler or some intermediate-level language. (By 'intermediate-level' one usually means a language with a good structure, but with means to manipulate things not normally available to a high-level language (such as registers); it may however just be a more or less convenient shorthand for assembly-code.) PL/11 (Ref.9) is an example for the PDP-11, with features allowing easy access to CAMAC. This programming is usually done by a support team, and is relatively stable.

The analysis programs for the data will vary widely from experiment to experiment. On the other hand they communicate in simple ways with the data-acquisition system (to get an event, or start and stop the run), and they have little to do with input-output or real-time problems. They are thus conventional programs which may be written in any high-level language (consistent with memory limitations). FORTRAN is the most widely used, since it is understood by most physicists and supported by mini manufacturers. The programs, though larger than those used in simple hardware tests, are not large by any other standards these days, especially as memory restrictions place a natural limit on their growth! The possibilities provided in FORTRAN for a reasonably structured program are thus usually adequate.

6.2.4 TRANSPORTABLE SOFTWARE

In an environment like CERN, where several different types of mini are in use, the use of transportable software is attractive. FORTRAN and BASIC are useful in this respect, and attempts to standardize them and their interaction with CAMAC are valuable. Nevertheless, FORTRAN programs often prove difficult to transport in practice, due to incompatibilities between different compilers, and in some cases FORTRAN is not appropriate for a lot of system software. Normally intermediate-level languages are written for particular machines (like PL/11); an interesting experiment in providing one that is really portable is BCPL (Ref.10). It is more transportable than FORTRAN in that the same compiler is in principle run on all machines.

At CERN this has been used to write host-independent cross-assemblers for minis and micros; in addition it has been used to implement a portable Transport Manager which will run on various minis connected to the general packet-switching network CERNET (Ref.14). The main disadvantage is in efficiency, especially in terms of space, and this may bar its use in otherwise interesting applications.

6.2.5 PROGRAM-DEVELOPMENT FACILITIES

The sort of approach used for these data-acquisition and monitoring systems brings up again the whole question of
program development facilities; the traditional apparatus of source file and edit facilities, compilers, linkers, loaders etc. is required. This point has already been mentioned; the main choice is whether to do things locally or on a larger host. Here the choice is usually clear, in that very few experimental minis have adequate configurations to allow everything to be done locally, and even when they do it is usually not practical. We have some choice however: the extremes are

a) Do as much as possible locally, using links to access the host filing system for archiving the local file-base, accessing standard libraries etc., and perhaps for special compilers not available on the mini.

b) Do everything in the host, producing a core-load which is sent down the link to the mini for execution; even overlays or dynamically loaded tasks are fetched from the host when required.

An intermediate approach might be to use the host for all file and edit functions, but compile and task-build locally, keeping only commonly-used task-images, overlays etc. locally. The choice in practice will obviously depend on the size of configurations, availability of cross-software, speed, reliability of links etc., and may vary widely. Many examples of different approaches exist even at CERN.

6.2.6 GENERAL-PURPOSE MONITORING SYSTEMS

An alternative to the FORTRAN monitoring program exists; perhaps the physicist need not write a monitoring program at all! This is because the sort of thing done by the monitoring program tends to be fairly conventional; setting up, filling, and displaying histograms for instance. It may therefore be possible to provide a kind of 'black box' system, where the physicist can set up a terminal, often using a question and answer dialogue, the tests he wants to make, and can ask for displays when he wants them. Even when the physicist does write his own program, it tends to end up as a black box in the final data-taking phase, when the experiment may run for a long time in stable conditions, operated by people who know little about what goes on in the computer; we are just asking if this cannot be done in the same way for different experiments, as we have said the data-acquisition itself can.

This approach is used in the FOLM system (Ref.13) at Permilah. Most other systems I have mentioned require the user to actually write a program, even if it can be a very simple one. Naturally, the possibility exists even in such a system for the user to write his own routines and have them called as part of the monitoring procedure, but it is not necessary that he should.

What are the advantages and disadvantages of this? It is attractive for simple experiments, as the amount of work to be done before being able to run is very small. Again, it reduces the chance of error by keeping the general line of program flow and operator interaction in the control of standard (and presumably well-tested) software. In addition, it may reduce confusion if communication with the operator, run-control etc., is standardized for different experiments. The main price paid is lack of flexibility; it is difficult to foresee all the things physicists may want to do, and it may be very hard to provide sufficient generality within such a framework. In addition the communication and control aspects are those which directly affect the physicist as they run, and they may prefer to arrange things to their own taste. In general the trade-off will depend on the problem, with standard packages of this sort perhaps more attractive for smaller experiments.

6.2.7 THE CAMAC DESCRIPTOR

The last element of the data-acquisition system to be dealt with is the CAMAC descriptor, or list of CAMAC commands to be obeyed in response to various triggers. This is prepared in advance and used by the data-acquisition tasks. Essentially it is a rather simple 'program' consisting of trigger specifications followed by lists of CAMAC commands, with usually the possibility to make and test 'call simple data-processing routines. Normally data-transfer is implicit; it is assumed that all data generated by the CAMAC read commands will be put in the data-buffers. Sometimes a means is provided to specify which buffers to use if different possibilities are available. Usually the descriptor is set up either in a simple 'language' or via a question and answer dialogue. The 'language' approach is perhaps often used in the Q system (Ref.12). This includes the means to define names for modules, including their type and CNA address; there is a library of types so that the descriptor compiler can check for instance that the functions used in the descriptor are valid for that type of module.

For instance, we might have a definition

```
DEVICE THING XY001,1,1,0
```
defining a module called 'THING' of type XY001 in Crate 1, Station 1 Subaddress 0. Then the descriptor might contain the instructions:

```
RD16 THING IF Q.EQ.0 REPEAT
```

which would cause a block-transfer 16-bit read (F0) in Q-repeat mode to be made from this unit.

6.3 OPERATING SYSTEMS

For a test setup very little may be required in the way of operating systems. Many BASIC systems are 'stand-alone'; a single package including the interpreter and input-output routines is provided, which can be loaded into an end-of years computer. The input-output, which includes CAMAC, may not even be interrupt-driven. Even if real-time facilities are provided, they may be very primitive (like the INTR facility in CATY for instance). This is in keeping with the simplicity of the typical problem solved with these systems.
For real data-acquisition however, as explained earlier, some form of multi-tasking is needed. The next stage therefore is a simple type of 'single-user' system with device drivers to handle interrupts. Monitoring is done in the background, and normal I/O is initiated only from there. The data-acquisition program itself is written specially and grafted on to the basic system. It contains only the essential multi-tasking facilities which it needs to enable communication between the routines which handle CAMAC, buffers, and magnetic tape. This is quite a fruitful approach, which has been the basis of several successful data-acquisition systems (e.g. EMR, HP at CERN; POLM at Fermilab). It has the merit of simplicity, is small and fast, and well adapted to straightforward high-rate work. Such a system is often too simple to support program development, which is done somewhere else.

This approach has its drawbacks however if the complexity of the problem increases, and this is often difficult to avoid. Ad hoc solutions tend to be grafted onto it. Here are some examples of the kind of problems that may arise:

a) We may want to run more than one kind of monitoring program, sharing time and events between them in a controlled way. (This can often be got around in practice without changing the basic approach; the next two problems are less easy to deal with)

b) There may be more than one type of "consumer" of data from the buffers (e.g. magnetic tape plus link to a bigger computer).

c) Suppose the data-acquisition task wants to output a status or error message on a device that the background program is using; this causes potential conflicts.

To solve each of these problems as it arises can quickly become clumsy and error-prone; we need a more general multi-tasking system which deals in a consistent way with interrupts, I/O queuing, task-scheduling, semaphores etc. (Ref.15). It is then tempting to go to a general 'operating system', such as those provided by the mini manufacturers (RXS-Ill, RTE-III, SINTRAN-III etc.) This usually provides many other benefits, such as memory management, a full range of peripheral drivers, access to standard file systems, FORTRAN run-time system, utility libraries, debugging aids etc. There is a price to be paid for this: inefficiency (task-switching times >1msec and large resident system), and complexity (when bugs occur they can be very hard to find). In addition these systems often require a communication system; people therefore prefer to stay with the simpler kind of system described above where possible; if the problem gets too complicated we can perhaps send it somewhere else like the computer centre.

Sometimes an attempt has been made to compromise: to provide a system with the simplest and most essential multi-tasking features, but with nothing unnecessary for the kind of problem in hand. (An example is the SMO system at CERN (Ref.17)). This can be an attractive solution, but it is a difficult compromise to make in practice. The risk is that the system will be forced to grow to greater generality in an ad hoc way and end up with less efficient version of a more general system. It can usually be successful only in well-defined situations where the effect of the trade-offs can be understood from the start. (SMO has been very successful in gamma-ray and cosmic-ray applications (Ref.16)). A way to try to get the best of both worlds will be discussed in the next section.

6.4 MULTI-COMPUTER SYSTEMS

This brings us to the possibility of using more than one computer; maybe we can split the problem up, and solve different parts of it on subsystems tailored to different needs (distribution of functions). One method would be to have one computer running the data-acquisition and recording, linking them together during the monitoring (Fig.8a). Then we can have a simple, efficient system in the 'front-end' (No extra tasks will run on it, it has no complicated peripherals or users...). Meanwhile the monitoring computer which needs the flexibility to deal with disks and other peripherals, filing systems, multiple tasks etc., can run a big general, heavy operating system. Sampling rates are usually lower than recording rates, so the link need not be a bottleneck.

This type of approach also allows us to share systems between different experiments if we want to. We might do it in one of two ways; a big monitoring computer, with expensive peripherals, might be shared between different experiments with their own small front-end system. Alternatively if the experiments use shared detectors or other equipment, data might be fanned out to different monitoring computers where physicists could keep out of each others way (Fig.8c, Ref.18).

What are the disadvantages of such an approach? One is that CAMAC and magnetic tapes are not readily available from the monitoring computer which may be inconvenient. We could move the magnetic tape to the monitoring computer (Fig.8d), but that also transfers it to some of the time-critical work, and the link now carries heavy traffic and may become a bottleneck. The latter problem may be overcome by some more intimate connection, giving the front-end direct access to shared memory for instance (Fig.8e). Now the front-end runs only one task, and all the multi-tasking is back on the other machine; we really have a single machine configuration plus an intelligent CAMAC controller (Fig.8f). This is how the '4' system at Los Alamos works; the front-end is a microprogrammable branch-driver interfaced to the PDP-11 Unibus, and essentially running one task of the data-acquisition system, while the rest runs on the PDP under RXS-Ill. The normal functions of a branch-driver are thus combined with the interpretation of the CAMAC descriptor, and in addition nonstandard block-transfer modes can be implemented. However, it has not really solved our multi-tasking problem!

- 102 -
Of course, once we do this, we may ask ourselves if we have put the intelligence in the right place; maybe it should go in the crate controller or elsewhere in the CAMAC rather than the branch-driver. This would give us more possibility of parallel processing. We can thus relate the problem to the 'hardware filtering' approach (Ref.2); this type of system with multiple front-end processors to reduce the data before it gets to the slow computer is becoming more important. The people thinking about improvements and extensions to CAMAC have in mind the kind of flexibility which would allow this kind of system to be built up easily.

Since some of our problems seem to have been generated by the need to record on magnetic tape and use various other peripherals locally, we may ask a more radical question: why not send all the data down a link to a large shared computer at a computer centre and record it there? (This is an extreme case of one of the models just described) And if all the data is sent off in this way, then maybe the calculations can also be done for the most part centrally. So we end up with a controller hanging on the end of a link! (Not quite in fact: it must still be a multi-tasking system).

This approach is attractive in many ways, and has been used successfully at Daresbury for example (Ref.19). It avoids duplication of expensive peripherals; magnetic tapes are shared and essential disk space is cheaper. Peripherals also work better in the controlled environment of a computer centre than on the experimental floor, where dust, electrical noise and shared power supplies cause problems. It provides all the resources of the big computer for program development etc., and gets away from the memory problems of small machines.

What then are the disadvantages? They are mainly practical (or even psychological!); many users prefer to have 'their' computer, with all the drawbacks. They do not like to have to fight for computer time, or go through the complex operational procedures of a big computer centre to get things done in a hurry. They feel frustrated if the central computer goes down. In addition the extreme case I have described is not in fact practical; it is hard to manage without local printing, and in fact fairly sophisticated means of dialogue with the central computer are required, which complicate the local system again. More seriously perhaps, the central computer does become very critical; if it stops so does everything else in the laboratory. Such a system has to be more professionally designed and operated than one where systems are more decoupled and 'graceful degradation' becomes possible. The total link traffic becomes large too, and may put a big strain on the central computer, whose real-time response may not be its best feature.
Finally, an approach which does not lend itself to central recording is one where raw data is recorded locally at very high rates on special equipment. This would saturate the link system and central facilities; locally however it can be 'played back' when the beam is off and the experiment is not running. It gives some of the advantages of on-line filtering but with more flexibility to change 'the methods'. It relies heavily on special recording techniques, however, and is not likely to become widespread unless big improvements in technology occur (Ref.20).

I hope I have said enough to give you an idea of the wide range of choices available in implementing online systems for high-energy physics. The solution adopted in a given case will depend on many factors; at least if you have to design one yourself you will have plenty to think about!

REFERENCES.

Note: many of these references describe work not formally published, and are available only as internal documentation from various laboratories. Students wishing to learn more about current developments may, however, be able to obtain these notes by writing to the authors.

3. The essential documentation on CAMAC hardware standards is contained in the EURATOM reports EUR4100 (and supplement), EUR4400 and EUR 6100.
4. 'Real-time BASIC for CAMAC', ESONE report ESONE/RTB/02, 1977; see also 'Subroutines for CAMAC', ESONE/SR/01, 1978
12. S.Ghser et al., 'Data Acquisition Program Q', Los Alamos Scientific Laboratory, USA 1976.
Communication Systems

Dag Belsnes
Terje Noodt

Computing Center
University of Oslo
Norway

Abstract

This paper gives a survey of the field of data communication. The topics covered are: types of communication, protocols, communication control systems, communication equipment and techniques, and types of data nets. Further, some of the data nets in use today, and the techniques applied in their implementation, are described. The intent of the paper is not to give an in-depth analysis of the various data communication techniques; rather, we attempt to describe the principles and problems involved in the construction of a state-of-the-art communication system.

1. Introduction

The ability of man is closely related to the amount of information available to him. History shows an ever increasing utilization of information and information processing.

The handling of data (representing information) can be divided in three different categories:

- Processing of data.
  It is unusual that data give directly applicable information. Hence, data have to be processed.

- Storing of data.
  It is unusual that data can be processed directly. Hence, data have to be stored.

- Transportation of data.
  It is unusual that data are stored where they are needed. Hence, data have to be moved.

This shows that communication is a necessary part of data processing, and that the need for data communication is strongly related to man’s need for information. Note that the three components operate fairly independent of each other in most cases.

For instance, an algorithm to sort numbers can work regardless of whether the data originate from a card reader or a disk file.

In data processing (and also in other connections), we know that the number of relations and bindings between the components of a system has direct influence on the system’s complexity, stability, efficiency, reliability,... Thus, during system design it is important to make a decomposition with fairly independent components. Applying this principle to data communication gives the following rule:

when transferring data it is (or should be) of little concern how the data are stored or processed before and after the transportation.

The applicability of some of today’s communication systems is highly restricted because of violation of this rule.

Data can be moved in many ways. Examples in connection with non-automatic processing are: letters, books, telephone, telex, TV. The use of these types of communication has shown a rapid increase during the last century. Since the invention of the telephone in 1876, the number of installed units in the US reached 100 millions in 1970.

The electronic computing machines have since 1950 made a revolution in the area of automatic information processing. In the early days of computers (1958-1965), both data and persons using the computer had to be located in the computer room. This centralized usage and lack of automatic transportation of data were clearly a limiting factor on the range of applications. The communication media were cars and postal services which carried card decks, listings, and persons. Since transportation of data is a fundamental component of data processing, we now have a corresponding revolution in data communication, 15 years late.

The following figures illustrate the data transmission growth. In 1970 the number of terminals in the US was about 185000, and the amount of traffic 14 billion transactions. In 1975: 800000 terminals and 50 billion transactions. The estimate for 1980 is: 2.5 mill. terminals and 250 billion transactions. In addition, there is an increasing amount of data traffic between computers in computer networks.
Data communication is used in many applications, which can be divided broadly into a few categories according to some main characteristics:

- **Dialogue** or not; i.e., after sending a unit of information (a message), do we require a response?

- **Delay**; what delay are we willing to accept between the time a message is sent till it is received at its destination?

- **Capacity**; if we send many large messages, the transmission line must be fast in order to keep up with the traffic.

Based on these characteristics, we may list five types of application of data communication.

- **Message switching.**
  A message switching system is used for sending messages (letters) one way, in the same manner as we use the mail. The TELEX system is an example of a message switching system. This application requires no dialogue, and usually a delay of hours is acceptable. Unless the traffic is high, low capacity transmission lines can be used.

- **Interrogation.**
  Typical examples of interrogation systems are those for airline reservations and banking. The user has a relatively short menu of questions he can ask, or orders he can give, and he normally gets a short answer back. This application requires a dialogue of short duration, and the delay ought to be less than 5 seconds. High transmission capacity is usually not needed since the messages are short.

- **Interactive computation.**
  This is a service offered by timesharing systems. It requires a dialogue of some duration, in which delays must not be too long. Regular interactive programming requires but moderate transmission capacity, while for instance an interactive graphics terminal will require high capacity.

- **Remote batch and file transmission.**
  These two applications are in principle implemented in the same manner. Although we expect a response to the job submitted at a remote batch station, the answer is so much separated from the request that we will not consider this to be a dialogue. Delays measured in minutes are acceptable, but high transmission capacity is required, since large volumes of data are transmitted.

- **Computer network.**
  General resource sharing computer networks usually serve applications of all the four preceding types. Therefore, we require both dialogue, low delay and high capacity. A computer network enables the user to select the computer which suits his present computation, or the one with the lowest load.

2. **Basic concepts in data communication**

In this section we will define and discuss the main elements of data communication. Let us first consider the expression "data communication". It can be defined as:

Transmission of data from a data source to a data sink via a communication medium according to a protocol.

The definition explains:

- What is transmitted (data)
- Who communicate (data source and sink)
- How the communication is performed (via a communication medium according to a protocol).

The data source or sink can be a hardware device (interactive terminal, CRT, LP,...) or a software program (I/O-driver, user program, timesharing monitor,...). The main point in our context, is that the data source is something (a process) that produces data which are given to the communication medium for delivery to the data sink. The data sink is a process which consumes the data received from the communication system.

In many applications, the communicating processes both transmit and receive data. In such cases we can regard the processes as two sink/source pairs, or we can extend our definition to include a two-way (duplex) communication between the processes.

When two processes communicate, it is not enough just to deliver data to the other process. The processes also have to agree on the interpretation of the data. By a process-to-process protocol (p-p protocol) we mean a set of rules/agreements between the processes for *formatting of the data
* interpretation of the data
* synchronization of the communication
* control of the data flow
* control of abnormal situations.

Such a protocol can be very simple, e.g., an interactive terminal communicating with a
driver for the terminal, or it can be very
complex, e.g., communication between network
control programs in a computer network
(protocol for program addressing, error
correction, data packet sequencing and
assembly, data flow control, ...).

The **communication medium** moves the data
physically (or logically) from the data
source process to the data sink process. A
leased or dial-up telephone line is an
obvious example of a communication medium
which is commonly used to connect equipment
communicating over some distance. But a
communication medium can also be a radio or
satellite link, a private high-capacity
cable, a huge data network, or even a
buffer system within a computer. Since we
regard a communication medium not only as a
transmission line but also more composite
systems with control functions (e.g., data
networks), we will in the following use the
name **communication control system (CCS)**.

The processes do not communicate directly
with each other, but via the CCS. A process
communicates with CCS over the interface
process/CCS by the use of a local **access
protocol** (see figure 2.1). This access
protocol depends on the services offered by
the CCS.

The p-p protocol consists of those rules
given implicitly by the effect of the
access protocol, and of those rules
explicitly expressed in the messages
transmitted between the processes.

In turn the system shown in figure 2.1 can
be regarded by "higher-level" processes as
representing a new CCS with new, and
hopefully better, services. Thus, a CCS
often consists of many layers, each with
its own protocol. We will look into such
protocol hierarchies later.

The communication between two processes can
be divided into three phases:
1) establishment of the process connection
2) data transfer on the process connection
3) de-establishment of the process
connection.

A p-p protocol controls these three phases.
As mentioned, such protocols will range
from simple to very complex depending on
the needs, and on the quality of the
underlying CCS. For instance, the phases 1)
and 3) can be eliminated if there is no
need for variations of active process
connections. On the other hand, the control
can be comprehensive in a net of
inhomogeneous computers with different
process concepts and addressing methods.

The quality and usability of the CCS depend
on the following factors. These factors
also indicate the main problem areas in
data communication.

* Probability of introducing errors in data
  If the CCS is a bad transmission line, this
  probability is high. If the CCS
  contains error recovery procedures (e.g.
  HDLC), then the CCS may be considered
  error-free.

* Possibility of duplication/loss of data
  Due to error recovery (with
  retransmission), duplicate messages may
  occur in some CCSs.

* Out-of-order delivery
  In a packet switching network with
  alternate routing of packets, data can
  be delivered out of sequence.

* Simplex/duplex connection
  A protocol for a simplex (one way at a
time) connection is fairly easy to
  construct, but the efficiency of the
  connection will usually be low. A duplex
  connection can give higher throughput
  and shorter delays, but the protocol

---

**Figure 2.1**

---

**- 107 -**
then becomes quite complex.

* Multiplexing of many connections
Some CCSs connect only one source/sink pair, while others can handle many connections simultaneously.

* Addressing possibilities
If the CCS connects a fixed set of process pairs, there is no need for addressing, while a CCS connecting programs in different computer systems, should contain good addressing capabilities.

* Transit time for data inside the CCS
CCS transmission delay is the amount of time between submission at the source and delivery at the sink process. For a direct transmission line, the delay is short and constant, while in a CCS with many components and queues, the delay can be long and variable.

* Capacity of data transmission within the CCS

* Flow control
Connected processes often produce/consume data at different or variable rate. Then the CCS has to control the data flow between the processes and prevent overload of the CCS capacity.

* Restrictions on data formatting
A CCS may contain restrictions concerning the length of messages, legal characters, control messages, and interrupts. Such restrictions may cause few problems, or they can prevent an efficient solution to a given application.

* Treatment of abnormal situations.

3. Communication Equipment and Techniques

3.1 Communication Equipment

In this section we will briefly describe the equipment used in data transmission, and the main types of transmission methods. We can broadly classify the units which communicate in following groups:
- host computers,
- terminals,
- front-end computers,
- datanet nodes.

A host computer performs the major processing of the data, and can store data. It is usually a large computer which offers a broad spectrum of services, but it can also be a small computer (mini/micro) for a dedicated task. Almost all of today’s computers have systems for data communication.

A terminal is usually connected to a host computer, and is used to read in and write out data from the computer. There exists a variety of different terminals, but the two main classes are those used for interactive conversation with the host, and those used for batch processing.

The control of data transmission on many lines requires a lot of processing capacity. In addition, it involves time critical operations. In order to relieve a large (and expensive) host computer of this task, it is common to let a front-end computer take care of the communication to and from the host. This also facilitates a better decomposition of the total system (no complex communication software in the operating system of the host). The communication between the front-end and the host is simpler due to short distance and special hardware (e.g. direct memory access).

If many units communicate with each other, it is uneconomical to have direct physical connections between every pair. Instead, the units communicate through a data network. This network consists of node computers connected by transmission lines. The nodes (usually mini computers) have the task of routing the data through the network from the source to the destination.

The most common way to connect a terminal to a computer is by a transmission line, which in general can be an ordinary telephone line. A special interface equipment (modem) converts between digital signals and audible sound. Direct lines (current loop) can be used if the distance is short.

The transmission can be simplex (transmission in only one direction), or half duplex (both directions, but only one at a time), or full duplex (both directions simultaneously). Two different modes are used to transmit data. Asynchronous transmission means that each data character is sent as an isolated entity. In order to identify one (8-bit) character, eleven bits are transmitted (including start and stop bits). When a lot of data are transmitted,
a better line utilization is obtained by sending a whole block of characters as one long bit-string. Special synchronization characters are used to identify the start of each block (this also defines the start of each character throughout the block). This mode is called synchronous transmission.

3.2 Communication Techniques

Transmission lines are expensive. In many cases the whole transmission capacity is not utilized by one source/sink connection (e.g. asynchronous transmission). One method to increase the utilization of a transmission line, is to multiplex many logically independent connections on the same line. Two different ways to divide the transmission capacity are:
- time division multiplexing, i.e. the connections occupy different time intervals.
- frequency division multiplexing, i.e. data are sent simultaneously, but the connections occupy different frequency intervals.

The line is shared by a fixed number of users. However, if one user does not use his share for a while (time slot or frequency interval), the corresponding line capacity is lost. This disadvantage is eliminated in a third multiplexing method:
- each block of data includes an identification of the connection to which the block belongs (e.g. HDLC, X.25).

As mentioned above, when the number of communicating units is large, it is necessary with some sort of network in order to allow all possible requests for connection between the units. The telephone network consists of many switching centrals connected by multiplexed trunk lines. A call between two subscribers requires the establishment of a fixed path through some switching centrals using parts of the multiplexed lines. This technique is called circuit switching. It results in an extensive sharing of the communication network, taking into account the infrequent use of a large number of telephones and a distributed traffic pattern. It would require an enormous net to have capacity for all telephones to be used simultaneously.

A circuit switching network guarantees a fixed throughput on each connection, since a physical path is allocated. On the other hand, unused capacity is lost. In many applications of data communication, the throughput requirements fluctuate considerably during the lifetime of the call. This fact has lead to the implementation of packet switching networks. The basic operation of a packet switching network is as follows. The sending process delivers a unit of data (a packet) to the network. The packet is routed through one or more network nodes before it is given to the destination process. If packets are not sent on one connection, the network capacity is used to send packets on other connections. This enhances a better overall utilization of the network. The weaknesses of a packet switching network are: variable transmission delay/throughput on each connection, and avoiding overload situations (network congestion). A circuit switching network can avoid overload by simply rejecting new calls.

Two different transmission strategies in a packet switching network should be mentioned. In a datagram network each packet sent between two processes is treated independently by the network. Then the network does not need to keep information about each active connection, and there is no call set-up time. However, each packet (datagram) has to contain the complete network addresses of the two processes (data overhead), and the destination process must perform sequencing if needed (datagrams may arrive out of sequence).

A virtual call network tries to establish a situation similar to circuit switching. Before two processes can communicate, they perform a connection set-up. After a call is successfully established, data packets can be transmitted with only a reference number to the logical connection. The data packets are delivered to the destination in correct sequence. They can very well get out of sequence within the packet network itself, but they are brought back in sequence before delivery. A datagram network is ideally suited for short transaction-oriented traffic, while a virtual call net is better for transmission of large data files.

4. Description of some data networks

Data networks can roughly be divided in the following categories: specialized net, general net, computer manufacturer net, public net, and local net.
4.1 Specialized networks

A specialized data network is part of a system with specific and well defined communication requirements, and therefore the network can be designed and tuned to meet these particular requirements.

Typical examples of specialized nets are military control and command networks, banking systems, and airline reservation systems. Already in the early 1960's American Airlines installed, in cooperation with IBM, a large network of this type. The objectives of the system, called SABRE, were:

- maximise the utilization of available seats on the air flights.
- give exact and updated information to prospective customers, agents, the operations personnel, and the management.
- increase the control and efficiency of the daily operations.
- provide statistics and data for long range planning.

The application has a set of characteristics and requirements which highly influenced the design of the SABRE communication system:

- real-time inquiry/response
- real-time data base update
- large number of terminals (several thousand)
- distribution of terminals from one to several hundred per location
- large centralized data base
- less than 3 sec. response time
- relatively short message length in both directions
- 24 hour operation
- high availability.

The topology of the SABRE network is a tree structure where all communication goes between the central computer (the root) and the terminals at the agent offices (the leaves). The main reason for this structure, with one data processing centre, is that the data base is updated in real time from many locations simultaneously. The centralization makes it easier to avoid conflicts (same seat is reserved by two persons). However, the whole system becomes vulnerable for errors in the centre. To fulfill the requirement of a 24-hour uptime, nearly all components in the processing centre of SABRE are duplicated.

Many terminals are situated a long distance from the centre. To connect each terminal by a direct line becomes prohibitively expensive, and the requirement of less than 3 sec. response times rules out a solution with dial-up lines. Instead, all terminals in one area are concentrated by a multiplexer which in turn is connected to the centre by a leased line. Again, in order to save line costs, many multiplexers are connected to the same transmission line (multidrop). This is a good example of how a distinctive feature of the application (a large number of terminals which infrequently send short messages) is expressed in the net design (extensive sharing of the transmission lines).

To achieve an efficient and reliable transmission, the line protocol uses

- 6-bit character code (even though the main computer is IBM)
- Cyclic redundancy check
- Full duplex transmission.

The two last features are used in most modern line protocols, but terminal systems with less efficient alternatives are used even today (DCT2000, IBM3270, CDC200UT, ...).

The end-to-end protocol in SABRE also takes advantage of the fact that a human at the terminal can control the communication. The protocol rules are simple and efficient compared to other protocols originated at that time, but they would not be sufficient for machine to machine communication.

4.2 General networks

These networks consist of terminals and computers of many types. The transmission network is able to handle different types of traffic patterns.

A typical example of a general network is the ARPA NET. This network, which became operational in 1970, connects various computers in more than 50 universities and research institutes in the US. The SABRE net was designed for the connection of distributed terminals to a central computer, whereas the ARPA NET links a large number of advanced data processing systems.

The main objective of the ARPA NET was to allow persons and programs in one research centre to get access to data, equipment, and program systems in other locations. In addition, an important aspect was to develop new and better methods and techniques in the data communication field.

The computers, called hosts, are connected through a transmission subnet consisting of mini computers (called IMPs). Each IMP is
connected to other IMPs (at least two) by high speed transmission lines (50 Kb/s). Europe is also included by a satellite link. Most of the hosts have their own local terminal network, and the terminals get access to the rest of ARPANET through their own host. A modified version of the IMP (called Terminal IMP (TIP)), allows direct connection of terminals.

The communication between the hosts goes as follows. The host gives its local IMP messages including an address to the remote host. In the IMP the message (which can contain many thousand characters) is split into shorter packets. These packets are then transmitted through the subnet to the remote IMP. The path each packet follows depends on the status of the subnet (current load and error situations). The packets are reassembled into a whole message before the message is given to the remote host. This reassembly has caused some problems due to the possibility of deadlock in connection with buffer allocation in the IMPs.

The transmission subnet of ARPANET is very reliable. It is estimated that a complete disconnection between two arbitrary IMPs does not occur more than half a minute per year on the average. This is achieved by
- at least two different routes between each pair of IMPs
- error checking (24 bits CRC) and error recovery (retransmission) on the transmission line between any two IMPs
- robust design of the IMPs.

Flow control in the transmission net is obtained by not allowing any two IMPs to have more than a few messages in transit at any time. This control mechanism is fairly conservative, and has shown to reduce the capacity on file transfers, in particular between distant hosts. In some cases (many remote IMPs sending to the same IMP) this flow control does not prevent network congestion.

In the ARPANET there are many layers of protocols, each representing a new CCS (see section 2). The low level protocols deal with communication problems; the high level protocols are more related to user functions.

- The IMP-IMP protocol provides reliable communication among the IMPs. The protocol handles transmission error detection and correction, flow control, and routing.
- The IMP-HOST protocol controls the communication between a host and its local IMP. Through this protocol a host can establish virtual connections to other hosts.
- The HOST-HOST protocol contains the set of rules whereby the hosts control and maintain communication between processes running on remote computers. Thus, this protocol level establishes a CCS suitable for process to process communication.
- The application which currently dominates ARPANET activity is the remote use of interactive systems. The TELNET protocol gives the users this capability. A user at a terminal, connected to his local host, can communicate with a process in a remote host as if he were a local user of the remote host. The TELNET protocol initiates and terminates, on the request of the user, connections between the local and remote hosts, controls the data exchange between the hosts, and unifies the access to the different host systems.
- The File Transfer Protocol (FTP) is used for moving files between the ARPA hosts. The FTP also includes operations such as list remote directory, rename remote file, and delete remote file.
- The RJE protocol provides remote job submission. The RJE is constructed on top of the TELNET and FTP protocols.

In addition, there exist a number of special user defined protocols. One of the more difficult problems in implementing such protocols in a general network of diverse computers and operating systems, is that of dealing with all types of incompatibilities. For instance, the FTP protocol has to interface to a variety of different file systems. Even more difficult would it be to define a protocol for remote access to files (a program performing I/O on a remote file).

There exists a lot of literature about ARPANET, and it is obvious that the design of this network has had a great impact on later work in data communication. The ARPANET has also had a positive effect on reducing the communication gap between distant groups of research workers. It has been possible to immediately utilize software which has been under development at other institutions, and scientific
papers and new results have been available
to others long before being published
elsewhere.

Another packet switching network is the
CYCLADES network in France. 16 research
institutions were connected in 1974. The
net design has much in common with the
ARPA NET, but an important difference is
that the communication between the hosts is
based on datagrams instead of virtual
calls.

4.3 Computer manufacturer networks

Computer network consisting of equipment
from one vendor. These nets have aspects in
common with nets of both type 1) and type
2). They are specialized because the design
of the transmission net is influenced by
special properties common to this type of
computers, and because it is usually
difficult to connect other types of
equipment. They are general networks
because they aim to cover a broad range of
applications (vary from one customer to
another).

Most manufacturers of computers have, or
are making, systems to interlink machines
of their kind. CYB ERNET is a collection of
about 20 large CDC CYBER machines. These
computers offer interactive or batch
services to a large number of terminals
throughout the world. The transmission
network is designed mainly for terminal to
computer communication, but one host may
transmit data to another host by simulating
a terminal station (CDC 200 UT).

The main advantages of this kind of network
are:
- to increase the availability of a CYBER
system. If one computer is out of order,
the customer can use other systems.
- to balance the load on all systems. For
instance, the peak hours differ on the
east and west coasts of USA.
- to increase the utilization of computers
which are specialized for a particular
application.
- to reduce line costs. A customer makes a
local call to the nearest switching
centre. From that point the
communication to/from the selected
computer is multiplexed on leased high
speed lines.

CYBERNET has recently been interfaced to a
large net of IBM-computers (SBC CALL/370),
and may also be classified as a general
network.

DIGITAL has developed a system, called
DECnet, to allow inter-system
communications between their computers
(PDP-8, PDP-11, PDP-15, and DECSYSTEM-10).
The machines can be connected by different
types of transmission lines and speeds
(asynchronous or synchronous lines, serial
or parallel lines, full or half duplex
lines). There are few restrictions on the
topology in which the computers can be
inter-connected.

DECnet consists of program modules which
are integrated in the operating systems of
the computers. This allows for a general
communication between programs in remote
computers. It is used for

* Device sharing
  Terminals on one system can get access
to another system, and peripheral
equipment (disk station, line printer,
etc.) at a remote location can be
controlled and used as if it were
locally connected.

* File sharing
  Files can be transferred between
  systems, and programs can access remote
  files as if they were local (e.g.
  record-wise access).

* Program sharing
  The ability to send loadable programs to
  another system for loading and
  execution. This process allows for the
  construction of networks with satellite
  systems that do not need mass storage
  capabilities, but are able to run a
  variety of applications by requesting
  programs from other systems in the
  network.

* Control sharing
  The general program to program
  communication (open a data path between
  remote programs) allows for distribution
  of the control of a large task. A
  program can be divided and distributed
  into smaller units to be executed on
different systems within the network.

To achieve the functions listed above, and
to allow easy integration into existing
operating systems, DECnet has been designed
using a layered structure approach. Each
layer performs a distinct set of functions.

* DIGITAL Data Communications Message
  Protocol (DDCMP) handles the link
  traffic control and error correction
  problems within DECnet. Characteristics:
  - operates over a wide variety of
    hardware types
good utilization of full duplex channel capacity and of lines with long signal propagation delays (satellite links)
- allows transmission of all data types (including binary)
- uses 16 bits cyclic checksum for error detection, with correction by retransmission.

* Network Service Protocol (NSP) builds on an errorfree communication between the computers (task of DDCMP). NSP handles the control of logical connections through the network. It makes it possible for two programs on different machines to establish a logical communication channel, and to exchange data using this channel. Characteristics:
  - dynamic creation/deletion of logical channels between remote tasks
  - exchange of data and interrupts on these channels
  - control of data flow and route selection within the network
  - dynamic management of the network topology. Each node is kept informed of the current state of nodes and links in the net.

* Data Access Protocol (DAP) enables programs on one node of the network to utilize the I/O services available on other network nodes. Each operating system in DECnet provides facilities for translating its own unique I/O calls into the DAP standard. Characteristics:
  - remote file access on sequential and random files (open, read, write, close, delete,...)
  - remote device access
  - virtual terminal support - allowing an interactive terminal to operate as though it were connected to another system within the network.

DIGITAL's systems are influenced by the interactive usage based on a character by character communication. This causes some problems in connection with data network.

- Each message going through the network must contain a certain amount of network control information. This implies that the effective utilization of the transmission lines becomes bad for short messages.

- The treatment of echo (response on typed characters) both complicates the virtual terminal protocol and causes extra traffic on the lines. (The problems concerning echoing and the solution to these are interesting from a theoretical viewpoint.)

Norsk Data has implemented a system, called NORD-NET, for connecting NORD computers with the SINTRAN III operating system. NORD-NET is similar to DECnet both concerning the capabilities offered to the users and the technical design.

Since the machines and operating systems are very similar, and since they are modified under control of only one firm, it is possible in a manufacturer network to obtain a strong integration of all components of the system. Seen from the user, any change in the location of the resources he is using, requires just minor modifications of commands and programs. On the other hand the user is not offered the same variety of capabilities as in an inhomogenous net, and he meets problems if he wants to connect equipment from another manufacturer. The conformity and integration is a special property of manufacturer networks compared to general networks. The rest of the world tries to achieve the same advantage through standardizations, but the progress is slow and painful.

4.4 Public networks

These nets are built and administrated by the PTTs in each country. Telephone, telex, telegraph, (and post) can be regarded as systems for transmission of information. Each country has developed extensive transmission networks for these services. In some countries the services are offered by commercial companies, but in most cases the network is administrated by a governmental institution. By the use of special interface equipment (modems), the telephone networks have to a great extent been used also for transmission of information between computers. In 1976 about 3000 data terminals were connected to the telephone system in Norway, and the telephone administration has established a special branch to take care of the problems with this type of use.

However, the telephone network is not designed for data communication, and thus the quality and the capabilities are not ideally suited for this type of use. Furthermore, it is undesirable that various users develop incompatible data networks based on telephone lines. It costs a lot to implement such data nets, it becomes difficult to inter-connect them, and it is
hard to obtain a good utilization of the transmission resources. Therefore many countries have started to build transmission networks specially designed for communication between data processing equipment (e.g. DATAPAC in Canada, EPSS in England, TRANSPAC in France, CTNE in Spain), Denmark, Finland, Sweden, and Norway have in cooperation started the development of a public data network for the Nordic countries. It is estimated that this net will serve about 60,000 subscribers in 1985, and the hardware cost is assumed to be about $150 mill.

The public data networks are pure transmission nets, and the connected equipment (called DTE, Data Terminating Equipment) is regarded as a subscriber outside the net. The DTEs can vary from small terminals to large data processing centres, and they are connected to the net through locally installed subscriber-connectors (called DCE, Data Circuit terminating Equipment). The DTE transmits data and requests for connection setup or special services by sending signals, standardized in a CCITT recommendation (X.21), to the DCE. This gives a well defined logical and physical interface between the transmission network (the responsibility of the PTT) and the subscriber equipment (responsibility of the user).

The public networks are based on the work of CCITT (Comite Consultatif International Telegraphique et Telefonique). Through the recommendations from CCITT, various international standards within telecommunication are developed. This standardization work is most useful and timely, since the computer vendors seem to be unable to make their systems more compatible. The CCITT work is also moving into the area earlier considered as belonging to the computer people. The CCITT recommendation X.25, which defines the interface between a packet mode DTE and the DCE, also implies an end-to-end protocol for packet exchange between computers, and CCITT recommendation X.29 defines procedures for exchange of control information and data between an interactive terminal and a computer.

4.5 Local networks

In many institutions there is a lot of data processing equipment spread out in a local area. The equipment range from powerful general-purpose computers to dedicated mini (micro) computers, special I/O devices, and terminals. To get a good utilization of the available hardware and software, it is often highly desirable to connect the equipment by a local data network. If everything is delivered by the same vendor, it is natural to use the network of the manufacturer, or if the institution has a very specialized data processing, a special purpose network can be implemented.

However, it is often the case (e.g. universities, research institutions) that the institution's need for data processing involves many applications (with different requirements to the communication) and that a variety of incompatible equipment is in use. Consequently, it may be impossible to connect two units directly together (for instance by a public network), and a major task for a local network is to bridge such incompatibility gaps. The local network should interface each unit in a way to make them compatible as far as data communication is concerned. On the other hand, this makes the network itself more complex. A local network is also influenced by the fact that the equipment in a local area may be physically connected in different ways. The computers can share memory or an I/O-bus, and transmission lines can be of various types and quality. The variations in transmission speed make it important that the local network has effective procedures for handling the flow control problem.

Typical examples of local data networks are: CERNET at CERN, NPL Data Communications Network at National Physical Laboratory, MININET at Polytechnic of Central London and Bologna University, and BRUNET at the University of Oslo.

References

* Jensen, W., "Elements of Data Networks", Report no. 32, Dept. of Computer Science, Univ. of Tromsø 1977.


* Woodt, T., Belsnes, D., "Data Communication" (in Norwegian), Lecture notes, Univ. of Oslo 1977.


DESIGN OF EXPERIMENTS

D. Drijard
CERN, Geneva, Switzerland

Introduction

This course is mostly devoted to simulation problems. The part which concerns detector optimization was essentially treated during the School in a seminar about the Split-Field Magnet (SRM) detector installed at the CERN Intersecting Storage Rings (ISR). This is not given in the written notes since very little of general use can be said about this subject, unless very trivial. We wanted rather to describe in a detailed way the tools which allow such studies to be made.

The notes start by a summary of statistical terms. The main emphasis is then put on Monte Carlo methods and generation of random variables. The last section treats the utilization of detector acceptance, which will be one of the most important parts to optimize when designing a detector.

1. Statistical terms

In this section we summarize the statistical material which we need in this course, so that no a priori detailed knowledge is assumed.

1.1 Continuous random variable (r.v.)

1.1.1 Range:

For simplicity of the expressions, we will use in the following a unidimensional r.v. and use explicitly its range $x \in [m,M] \subseteq [-\infty, +\infty]$.

1.1.2 Probability density function (p.d.f.):

$$f(x) = \lim_{\Delta \to 0} \frac{\text{Prob}(x - \frac{1}{2} \Delta \leq X < x + \frac{1}{2} \Delta)}{\Delta}.$$  

From its definition, $f(x)$ is a positive function of unit integral, i.e. $\forall x \in [m,M]: f(x) \geq 0$ and $\int_m^M f(x) \, dx = 1$. When it is not explicitly necessary we will not mention the range of integration; that is, we will have the implicit statement $\forall x \in [m,M]: f(x) = 0$ for the p.d.f. although the function $f(x)$ which represents it in its range may take non-zero values outside this range. The abbreviations r.v. and p.d.f. will be used throughout.

1.1.3 Cumulative distribution:

$$F(X) = \int_m^X f(y) \, dy \quad \text{hence} \quad F(m) = 0 \quad \text{and} \quad F(M) = 1.$$  

1.1.4 Conditional distribution:

For multidimensional r.v., we may consider subspaces. Let $f(x,y)$ be the joint p.d.f. in the r.v. $x$ and $y$ (which may still be multidimensional). Then integrating over one r.v., we find the marginal p.d.f. in the other r.v. $y$ like $g(y) = \int f(x,y) \, dx$. Finally, we define the conditional distribution $h(x|y) = f(x,y)/g(y)$. It represents the p.d.f. of the r.v. $x$, given a definite value of the variable $y$. Notice that these functions are still normalized to unity, as p.d.f. should be $\int f(x,y) \, dx \, dy = g(y) \, dy = h(x|y) \, dx = 1$.

1.1.5 Expectation value:

For an arbitrary function $g(x)$, we define the expectation value of $g(x)$ over the p.d.f. $f(x)$:

$$E_f(g) = \int g(x) \, f(x) \, dx.$$  

1.1.6 Variance:

$$V_f(g) = E_f(g^2) - [E_f(g)]^2 \geq E_f([g - E_f(g)]^2) \geq 0.$$  

The second expression exhibits the important positivity condition, which comes trivially from the integral of a positive function.

1.1.7 Change of variable:

We start from a r.v. $x$ whose p.d.f. is $f(x)$. We study a new variable $y = g(x)$ which is a r.v., since it is a function of a r.v. Consequently it has a p.d.f. $h(y)$ which is certainly defined completely by the knowledge of both $f(x)$ and $g(x)$ functions:

$$h(y) = \int f(x) \, \delta[y - g(x)] \, dx = \sum_i \frac{h(x_i)}{|g'(x_i)|},$$  

where the sum $\sum$ runs over all roots of the equation $y = g(x_i)$. For example, $y = x^2$ gives

$$h(y) = \frac{1}{2|x|} \left[ f(x) + f(-x) \right] = \frac{1}{2\sqrt{y}} \left[ f(\sqrt{y}) + f(-\sqrt{y}) \right].$$  

1.1.8 Moments:

$$\mu_n = E[(X - E(X))^n] \quad \text{is the n^{th} central moment; \quad \mu'_n = E(x^n) \quad \text{is the n^{th} algebraic moment.}$$  

- 116 -
The following notations are often used:

\( \mu \) is the mean value;

\( \sigma = \sqrt{\frac{1}{N} \sum (x - \mu)^2} \) is the standard deviation.

1.1.9 Characteristic function:

The Fourier transform of a p.d.f. is called a characteristic function. It is most useful in the study of sums of r.v.:

\[
\phi_x(t) = E(e^{itx}) = \int e^{itx} f(x) \, dx
\]

and

\[
f(x) = \frac{1}{2\pi} \int \phi_x(t) e^{-itx} \, dt.
\]

The main properties are the following:

i) \( \phi(0) = 1 \) and \( |\phi(t)| \leq 1 \).

ii) \( \phi_{x+y}(t) = \phi_x(t) \phi_y(t) \) if \( x \) and \( y \) are independent r.v. (that is, if the joint p.d.f. of \( x \) and \( y \) is the product of the individual p.d.f. of \( x \) and \( y \)).

iii) \( \phi_{ax+by}(t) = e^{ibt} \phi_{x}(at) \) if \( a \) and \( b \) are constant numbers.

iv) \( \mu'_t = \frac{1}{i} \left[ \frac{d}{dt} \right] \phi(t) \mid_{t=0} \).

v) \( \mu'' = \frac{1}{i} \left[ \frac{d}{dt} \right] e^{-it\mu} \phi(t) \mid_{t=0} \).

1.1.10 Normal law:

Usually designated by \( N(\mu, \sigma^2) \)

p.d.f. \( f(x) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left[ -\frac{(x-\mu)^2}{2\sigma^2} \right] \), with \( x \in [-\infty, \infty] \).

The mean value and standard deviation are explicit in the expression of \( f \):

\[
E(x) = \mu, \quad V(x) = \sigma^2.
\]

The characteristic function is \( \phi(t) = \exp \left[ it\mu - \frac{1}{2} t^2 \sigma^2 \right] \).

It is useful to remember some numbers about the probability content outside ranges centred about the mean value \( \mu \):

- Prob. \( (|x-\mu| > \sigma) \approx 0.52 \)
- Prob. \( (|x-\mu| > 2\sigma) \approx 0.046 \)
- Prob. \( (|x-\mu| > 3\sigma) \approx 0.0027 \).

It is customary to present experimental results by quoting estimators \( \bar{x} \) and \( \bar{\sigma} \). If the p.d.f. of the r.v. analysed were a normal law, we could deduce information on the probability that the true \( \mu \) value lies in some range about \( \bar{x} \), expressed in units of \( \bar{\sigma} \). It turns out that, in a low statistics experiment, the p.d.f. is far from being a normal law and, consequently, \( |x-\bar{x}| > 3\bar{\sigma} \) may often be much more probable than the 3\% expected from a normal law. As far as we are concerned in this course, we shall see reasons why we expect asymptotically normal laws, but this must be checked in practical cases. We shall mention (4.8) a simple check. At this point mention is made of a semi-analytical check:

There is a one-to-one correspondence between p.d.f. and characteristic function. Thus we may check this last one, or equivalently the set of moments. It is usual to study quantities based on 3rd and 4th moments, adjusted to be zero for a normal law (and dimensionless):

\[
\gamma_1 = \frac{\mu_3}{\mu_2^{1/2}} \quad \text{is the coefficient of skewness,}
\]

\[
\gamma_2 = \frac{\mu_4}{\mu_2^2} - 3 \quad \text{is the coefficient of kurtosis.}
\]

These coefficients are fairly easy to estimate (they are usually available in histogramming packages) and constitute something of a security for normality, although evidently not a certainty. As an example, a \( \chi^2 \) distribution with \( n \) degrees of freedom [sum of \( n \) r.v. with independent normal distributions \( \mathcal{N}(0,1) \)] has \( \gamma_1 = 2\gamma/n \) and \( \gamma_2 = 12/n \). It is considered reasonably normal for \( n > 30 \), since \( \gamma_1 \) and \( \gamma_2 \) are less than 0.5.

1.1.11 Uniform law:

\( \forall x \in [m, M] : f(x) = \frac{1}{M-m} \); otherwise \( f(x) = 0 \).

\( E(x) = \frac{1}{2} (M+m) \) and \( V(x) = \frac{1}{12} (M-m)^2 \).

We shall later refer to this law by the notation \( U(m,M) \).

1.2 Discrete random variables

1.2.1 Range

Using a one-to-one application, a discrete set can always be mapped on a set of consecutive integers. It will thus be assumed that the r.v. has a range \( r \in \{m, m+1, \ldots, M, M+1\} \).

1.2.2 Probability generating function:

\( p(r) \) is the probability that the r.v. takes the value \( r \).

All definitions given for continuous r.v. are valid for discrete r.v. provided integrals are replaced by discrete sums. For example,

\[
\sum_{m} f(x) \, dx = 1 \rightarrow \sum_{r=m}^{M} p(r) = 1.
\]

1.2.3 Binomial distribution

\( B(r | p, n) = \frac{n!}{r!(n-r)!} \, p^r (1-p)^{n-r} \)

with \( r \in \{0, 1, \ldots, n\} \) and \( p \in [0,1] \).

\( E(r) = np \) and \( V(r) = np(1-p) \).
1.2.4 Poisson law

\[ \mathbb{P}(r = \mu = \frac{1}{r}) \quad \text{with} \quad r \in \{0, 1, \ldots \} \quad \text{and} \quad \mu > 0, \]

\[ \mathbb{E}(r) = \mathbb{V}(r) = \mu. \]

2. Central limit theorem

Let \( x \) be a continuous r.v. whose p.d.f. is \( f(x) \). Then the arithmetic mean \( \bar{x}_n = (1/n) \sum_{i=1}^{n} x_i \) of \( n \) such independent r.v. is again a r.v. of p.d.f. which we note by \( f_n(\bar{x}_n) \). We can in principle compute \( f_n \) from the knowledge of \( f \), but this is in general a non-trivial calculation \( [\text{from 1.1.9, } \phi_n(t) = [\phi_x(t)]^n \text{ but the inverse Fourier transform is normally not simple}]. \) However, we may state asymptotic properties (as \( n \to \infty \)) under fairly general conditions. If we restrict ourselves to the simple case which we study here, a sample of \( n \) r.v. (independent) issued from a common parent distribution, then: if \( V_f(x) < \infty \) (i.e. finite),

law of large numbers \( \bar{x}_n \xrightarrow{n \to \infty} E_f(x) \),

central limit theorem \( f_n(\bar{x}_n) \xrightarrow{n \to \infty} \mathcal{N}(E_f(x), \frac{1}{n}V_f(x)). \)

As we will see later, these theorems are the basis of Monte Carlo integration. The first theorem is not very surprising, one expects \( \bar{x}_n \) to get close to \( E_f(x) \) when \( n \) increases; but the second is less trivial since \( f_n(\bar{x}_n) \) converges towards a definite law, normal law, independently of the parent distribution \( f(x) \).

The only condition required \( [V_f(x) \text{ finite}] \) seems harmless \( (a f o r t i o n i c h f o r p h y s i c s a p p l i c a t i o n s w h e r e t h e r a n g e o f r . v. o b s e r v e d i s a l w a y s f i n i t e ] \), but it should be noticed that the theorem says nothing about the rate of convergence (how large \( n \) should be?). We can get a feeling by the study of a few examples.

2.1 Normal law

If \( f(x) \) is a normal law, then \( f_n(\bar{x}_n) \) is a normal law as well, whatever \( n \). Indeed, using the characteristic function (1.1.9) in the special case of normal law (1.1.10), then

\[ \phi_x(t) = \exp \left[ itu - \frac{1}{2} t^2 \sigma^2 \right]. \]

If \( y = \frac{1}{n} \sum_{i=1}^{n} x_i \), then \( \phi_y(t) = \exp \left[ itu - \frac{1}{2} t^2 \sigma^2 \right]. \)

Finally \( \bar{x}_n = (1/n) \gamma \), then \( \phi_{\bar{x}_n}(t) = \phi_y(t/n) = \exp \left[ itu - \frac{1}{2} t^2 (\sigma^2/n) \right]. \) It then follows that \( f_n(\bar{x}_n) = \mathcal{N}(u, (\sigma^2/n)). \)

2.2 Cauchy law

\[ f(x) = \frac{1}{\pi} \cdot \frac{1}{1 + x^2} \quad \text{with} \quad x \in [-\infty, \infty]. \]

This is the usual Breit-Wigner formula after a change of variable such that the r.v. \( x \) is centred and rescaled. It is easily seen that the mean value \( E(x) \) is undefined and the variance \( V(x) \) is infinite. Let us restrict the range \( x \in [-A, B] \) with \( A, B > 0 \). Then

\[ f(x) = \frac{1}{\pi} \cdot \frac{1}{\arctan A + \arctan B} \cdot \frac{1}{1 + x^2}, \]

\[ E(x) = \frac{1}{\pi} \cdot \frac{\frac{1}{2} \ln \left( \frac{1 + B^2}{1 + A^2} \right)}{\arctan A + \arctan B} - 1. \]

If now we let \( A \) and \( B \) tend towards \( \infty \), then \( E(x) \) will depend on the ratio \( B/A \), and is thus undefined. We may cure this difficulty if we impose, say, \( A = B \), then \( E(x) = 0 \), but inevitably \( V(x) \) is infinite. So we are in the case where our asymptotic theorems should fail. Indeed, using again characteristic functions, we can show that

\[ \phi_x(t) = e^{-|t|} \quad \text{(no derivative at } t = 0, \text{ hence no moments).} \]

Then if \( y = \sum_{i=1}^{n} x_i \), \( \phi_y(t) = e^{-n|t|}. \)

Finally for \( \bar{x}_n = \frac{1}{n} y \), \( \phi_{\bar{x}_n}(t) = e^{-|t|}. \)

Thus \( f_n(\bar{x}_n) \) is identical to \( f(x) \). As in the preceding case, the type of law is not modified by the arithmetic mean process but, now, the width does not decrease with \( n \). This may seem an academic problem since the range is always finite in real life, but the rate of convergence towards a normal law will be slow. To get an order of magnitude, we may study the kurtosis \( \gamma_2 \) as a function of the size \( n \) of the sample and of the parameter \( A \). Using the moments expansion of the characteristic function given in 1.1.9, we find, without explicit reference to the Cauchy law in fact, \( \gamma_2 = 1/n (\gamma_2) \).

Then for \( A > 10 \) we may approximate \( \arctan A \) by \( \pi/2 \), and we get \( \gamma_2 = (nA/6) - 3. \) Consequently, the larger \( A \), the larger \( n \) will be necessary to have, say, \( \gamma_2 < 0.5. \) Notice that \( \gamma_2 = 0 \) for \( A = 6 \), which shows that \( \gamma_2 = 0 \) is not so good a test of normality.
2.3 Uniform law

If \( x \) follows a uniform law \( U(-0.5,0.5) \), then the r.v. \( u_n = x_n / \sqrt{n} \) follows asymptotically a normal law \( \mathcal{N}(0,1) \). This property is often used to generate r.v. according to a normal law, and \( n = 12 \) is frequently employed, which saves the calculation of a square root. One can compute directly 
\[
(y_2)_n = -1.2, \quad \text{and from the preceding paragraph} 
\]
\[
(y_2)_n = (1/n) (y_2) = -0.1. 
\]
The range of \( u_n \) is \([-6,6]\), which is large enough for most applications.

3. Monte Carlo integration

If we want to estimate an integral \( I = \int \frac{M}{m} g(x) \) dx, we may use standard techniques, such as the Simpson rule. If the integral is to be performed in multidimensions, then this method must be abandoned. Even more elaborate methods, such as Gauss methods, cannot be used for, say, more than five dimensions, and indeed in particle physics integrals of dimensions larger than 10 are often encountered. Such methods require sampling the function on a regular grid; so, for example, \( n = 10^d \) points if we require in \( d \) dimensions 10 points per dimension. This evidently becomes rapidly prohibitive. In addition, the error made on the integral varies asymptotically like \( n^{-2/d} \) for the Simpson rule. The Monte Carlo method, which we study now, does not suffer from such difficulties; in particular, we will see that the error on the integral varies like \( n^{-1} \), whatever the dimensions.

3.1 Conventional method

Let \( \{x_1, x_2, \ldots, x_n\} \) be a set of \( n \) independent r.v. which follow a common uniform law \( U(m,M) \), then a new r.v. \( \hat{I} = \frac{1}{n} \sum_{i=1}^{n} g(x_i) \) is constructed. By construction \( \mathbb{E}(\hat{I}) = I \) and \( \mathbb{V}(\hat{I}) = \frac{1}{n} V(g) \). From the central limit theorem [apart from rare cases where \( V(g) \) is infinite], the p.d.f. of \( \hat{I} \) tends for large \( n \) towards a normal law \( \mathcal{N}(I, (1/n) V(g)) \). The (asymptotic) normality allows us to make probabilistic statements about the closeness of \( \hat{I} \) and \( I \). In particular, there is about 95% probability that 
\[
|I - \hat{I}| < 2\sigma (1.1.10), 
\]
where \( \sigma^2 = (1/n) V(g) \). As mentioned earlier, \( \sigma \propto n^{-1} \). The same conclusions are obtained in multidimensions, simply \( x_i \) is now a set of \( d \) uniform r.v. (instead of one) and, in the expression of \( \hat{I} \), one must replace \( (M-m) \) by the volume \( \Omega \) spanned.

3.2 Importance sampling

We covered the region of integration with a uniform density of points (since uniform r.v. were used), but better results can be expected if the function \( g(x) \) is sampled with higher density in the regions where it is important. Let us recast the problem using an arbitrary p.d.f. \( f(x) \) instead of uniform law.

Let us define the "weight" \( w(x) = g(x)/f(x) \). Then the original integral is 
\[
I = \int g(x) \, dx = \int w(x) \cdot f(x) \, dx. 
\]
Our estimate will be 
\[
\hat{I} = \frac{1}{n} \sum_{i=1}^{n} w(x_i), 
\]
and, from central limit theorem, the p.d.f. of \( \hat{I} \) will tend towards a normal law \( \mathcal{N} \left[ I, \frac{1}{(1/n) V(w)} \right] \). The aim is now to adjust \( f(x) \) in order to minimize \( V(w) \) which governs the precision of the estimation. We first study this optimization problem; then in the next section we shall describe methods of generating r.v. according to some p.d.f. \( f(x) \).

The ideal case would be \( V(w) = 0 \), which implies 
\[
g(x) = I \cdot f(x). 
\]
Evidently, it would not even be necessary to generate a single r.v. since we can pretend to know explicitly the integral of \( g(x) \) [\( f(x) \) is of unit integral]. We address ourselves to the cases where such calculations are not possible, or at least not known.

At the other extreme, we may find \( V(w) = \infty \), in which case our theorems would not apply. This is apparently another academic problem but it happens in some careless applications, and stays unnoticed since no infinite number appears in the calculations. Assume that in some region of the space of integration the function \( f(x) \) is equal to 0 [but not the function \( g(x) \) to integrate]. Then in this region \( w(x) \) is infinite and consequently \( V(w) \) is infinite. However, this will not show up in the calculations since there will be no occurrence of the r.v. \( x \) in this region. Simply \( g(x) \) will be integrated in a restricted range, hence wrongly estimated; \( \hat{I} \) infinite simply reflects the absence of knowledge of a part of the integral.

Practically, one will search for a function \( f(x) \) reasonably close to \( g(x) \), in order to minimize \( V(w) \), but at the same time one must have a way of generating r.v. according to \( f(x) \). This will be the subject of the next section.

4. Random variable generators

4.1 Uniform law

This is the basis of all r.v. generators out of which they are constructed, as we will see in the following paragraphs. Most computers have a uniform r.v. generator, so we will not study this problem.
Simply, it should be remembered that the quality of such a generator (true uniformity, absence of correlations) will reflect itself on subsequent generators; hence one should not adopt a new generator without elaborate tests.

4.2 Inverse cumulative distributions

This is something of a brute force method -- that is, it does not require much finesse. Unfortunately it works only for simple cases. Let $x$ be a r.v. of p.d.f. $f(x)$ and cumulative distribution $F(x)$ (1.1.3). Then one can show that the r.v. $y = F(x)$ follows a uniform law $U(0,1)$. Reversing the statement, if $y$ follows $U(0,1)$ then $x = F^{-1}(y)$ follows the original $f(x)$. Thus the method requires a knowledge of the inverse cumulative distribution. This is rarely possible in an analytical way, but, at least in the one-dimensional case, we often construct a table of $F(x)$ by numerical integration and later solve the inverse equation by interpolation.

As an example, let us take an exponential law $f(x) = e^{-x}$ with $x \in [0,\infty]$; then $y = F(x) = \int_0^x f(u) \, du = 1 - e^{-x}$, and finally $x = -\ln(1-y)$. So, using for $y$ a uniform r.v., we get for $x$ an exponential p.d.f.

4.3 Change of variable

We have seen (1.1.7) how to compute the p.d.f. of a function of r.v. If we start from a uniform law, in one dimension, then the method is identical to the preceding one [i.e. if we find a change of variable $x = h(y)$ such that, when $y$ follows a normal law, $x$ follows the p.d.f. $f(x)$ in which we are interested, then this change of variable is certainly $x = F^{-1}(y)$]. However, in multidimensions this may not be so trivial. The following example shows how to generate r.v. according to a normal law (it is standard on most computers).

Let $\theta$ be an angle which follows $U(0,2\pi)$, and $r$ follow $U(0,1)$. Then $x = \sqrt{-2 \ln r} \cdot \cos \theta$ and $y = \sqrt{-2 \ln r} \cdot \sin \theta$ both follow normal laws $N(0,1)$, and they are uncorrelated. Compared to the method mentioned earlier, the average of $n$ uniform r.v., the present method is exact but a priori more time consuming.

4.4 Special tricks

If $r_1$ and $r_2$ both follow a uniform law $U(0,1)$, then $x = -\ln (r_1 \cdot r_2)$ follows $f(x) = xe^{-x}$. It can be seen by $f(x) = \int_0^x \delta \, dr_1 \cdot dr_2 \cdot [x + \ln (r_1 \cdot r_2)]$. This method is used, in particular, to generate transverse momentum distributions of the type $d\sigma/dp_T^2 \sim e^{-dp_T}$.

4.5 Keep or reject method

Assume we want to generate $x$ according to $f(x)$ with $x \in [m,M]$. Let $\alpha = \max f(x)$. Then we may use the following algorithm:

1) generate $u$ according to $U(m,M)$;
2) generate $r$ according to $U(0,\alpha)$:
   if $r > f(u)$, restart the procedure from point 1,
   if $r < f(u)$, keep the r.v. $u$.

The set of r.v. $u$ generated by such a method have $f(u)$ as p.d.f. on the average, i.e. at the limit of a large number of trials, the efficiency of the method is $e = \frac{1}{\alpha \cdot (M-m)}$, which represents the ratio of the number of accepted generations to the number of times the algorithm was executed [i.e. the ratio of the integral of the curve to the area of the rectangle where points $(r,n)$ were uniformly generated]. Notice that it is mandatory to redo both steps 1 and 2 when the test fails. This method may be easily generalized in multidimensions, but its efficiency may then become too small. Even in one dimension the method may be useless when $f(x)$ exhibits sharp peaks, i.e. of small width with respect to the range.

4.6 Weighting method

Assume that we found a random generator $g(x)$, close enough to the p.d.f. $f(x)$ which we want, buy not identical. We have now the choice between two methods. Either we keep the r.v. $x$ thus generated but every subsequent calculation must be done using the weight $w(x) = f(x)/g(x)$, or we use a keep or reject method. We shall study how to choose between these two methods, given the precision required, by comparing the time spent (computer time nowadays).

Let $\tau_g$ be the time necessary to generate the r.v. $x$, and $\tau_u$ the time necessary to utilize it. This last time may vary in very large proportions. As extreme cases, the shortest time may be to simply record a dichotomic variable, such as a forward backward asymmetry study; the longest may be the complete analysis of a high multiplicity event. Practically, $\tau_u$ may vary from 10 ms to 5 s (in the CDC 7600 computer).

4.6.1 With weights

From the foregoing, it can be deduced that, loosely speaking, a weighted event is worth $E^2(w)/E(w^2)$ and requires a time $(\tau_g + \tau_u)$. Thus the
time required for an "equivalent event" is
\[ \tau_1 = (g + \tau_u) \cdot E(w^2)/E^2(w). \]

4.6.2 Without weights

The keep or reject method is applied to the function \( w(x) \) (it was described in Section 4.5 for a p.d.f., but it need not be the case). So we must find an estimation of the maximum weight \( w_{\text{max}} \). In practical cases, it is rarely obtained by analytical methods but rather on a large enough number of trials, by keeping track of the largest weight observed. Then, in subsequent applications, for each r.v. \( x \) generated with a weight \( w \), we generate \( r \) according to \( U(0, w_{\text{max}}) \) and keep the generated value \( x \) if \( w > r \) (otherwise we restart \( x \) and \( r \) generation). The r.v. thus accepted are then given a unit weight (or any arbitrary but fixed value). Reformulating the efficiency given in Section 4.5 for a function \( w(x) \) whose integral is not unity, one finds \( e = E(w)/w_{\text{max}} \). Finally, the time needed for an "equivalent event" is \( \tau_2 = (g \cdot \frac{w_{\text{max}}}{E(w)} + \tau_u). \)

4.6.3 Example

We have just obtained the times \( \tau_1 \) and \( \tau_2 \) for the two methods of generating and utilizing one equivalent event; that is, corresponding to a common statistical precision (asymptotically). We can see how different these times may be in the case of high-energy, high-multiplicity events \( [\sqrt{s} \sim 50 \text{ GeV}, E(m) \sim 20], \) where \( m \) is the total multiplicity.

Known algorithms which allow complete events to be simulated exist with the following characteristics:
\[ \begin{align*}
\tau_g & \sim 0.1 \text{ ms/particle}; \\
E^2(w)/E^2(w) & \sim 0.1; \\
E(w)/w_{\text{max}} & \sim 0.01.
\end{align*} \]

We will assume that the events generated are saved and later used many times for different studies (setting up of the analysis chain, for example). An order of magnitude may be 100 utilisations at the cost of 25 ms per particle. Then we get
\[ \begin{align*}
\tau_1 &= (10^{-4} + 2.5) \times 10 = 25 \text{ s/particle,} \\
\tau_2 &= (10^{-4} \times 10^2 + 2.5) = 2.5 \text{ s/particle.}
\end{align*} \]

A sample, rather meager, of 1000 events of multiplicity \( \sim 20 \) was used in such conditions but, although these events have been used over three or four years, we were certainly legitimated in adopting the "no weight" method in view of the factor of 10 in time spent. Evidently this conclusion depends on specific cases, and we may be led to use the first method when the utilization time is low enough.

4.7 Johnson distributions

The Johnson family of distributions provide a wide range of shapes which we may use to represent empirical distributions of which we do not a priori have the p.d.f., or as a first step followed by a "weight or no weight" method. These distributions result from a change of variable applied to a r.v. which follows a normal law \( N(0,1) \). The interested reader should consult: G.J. Hahn and S.S. Shapiro, Statistical models in engineering (John Wiley and Sons, New York, 1967).

4.8 Remarks

The preceding statements, based on mean values, imply asymptotic limits, i.e. large samples. It is most useful to study the finite sample behaviour. The simplest way seems to be the following.

Every study that can be made is eventually summarized by a few numbers, such as, for example, the width of a mass distribution when resolution is analysed. If it is decided to generate \( n \) events for this study, then, say, 10 independent studies could be done on samples of size \( n/10 \). Then the distribution of the results could be compared with their otherwise estimated variance.

4.9 Conclusions

A serious pre-study of the problem in hand is extremely useful, especially when large computer time is needed:

- invest brains in r.v. generation, factors of 10 in time spent are not rare;
- choose "weight or no weight" method depending on subsequent use;
- not only may computer time be wasted by careless study but, in addition, stupid results may be obtained (that is, very imprecise).

5. Event simulation

In most cases, we want to generate exclusive events, according to some physics hypothesis. Sometimes the study may be done on the basis of the four-momenta of the particles. However, on many occasions, we may want a more refined study which takes into account the characteristics of the detector. In this case we must simulate completely the response of the detector both in terms of its
capacity to record information and in terms of background generated.

We must in general follow the particles in a magnetic field using Runge-Kutta methods or pieces of helix. Then global effects must be simulated, multiple scattering and energy loss, and discrete effects such as decays, secondary interactions, $\delta$-rays, etc. Eventually the same procedure must also be done for secondary particles. The formulae for such processes exist in classical text books; only for special studies do we need to search in the recent literature. Eventually the detector response must be simulated in terms of multiwire proportional chambers, drift chambers, calorimeters, etc.

In some cases, we must artificially enhance an effect. For example, in an experiment which looks for prompt electrons, we must study sources of background, such as $\delta$-rays. A priori, the experiment was designed in such a way that the particles cross a small thickness of matter, so that few $\delta$-rays are produced. A straightforward analysis would give rise to a small number of such background electrons (hence poor statistical precision from Poisson law or binomial law) at the price of a lengthy analysis of many events. So the idea is to enhance $\delta$-ray production. For example, at the level of individual particles, we may compute the probability that a $\delta$-ray was produced in the thickness traversed, attach it as a weight to the track, and generate a $\delta$-ray in this thickness according to the relevant exponential law but with certainty (i.e. probability 1). Thus we save the generation of the track and the calculation of its trajectory in the magnetic field for the vast majority of cases where a $\delta$-ray would not be produced. When two or more electrons should be studied in the same event, then the problem becomes more complicated (since one wants to preserve the right multiplicity of $\delta$-rays).

6. Acceptance

Given an event, defined by the four-momenta of the particles produced, we observe in general only a subset of these particles, with four-momenta slightly different. The imperfection of the detector has two components (although one goes continuously from one to the other): blindness in some regions of phase space, and imprecision in the observations. The concept of acceptance is used in a loose sense to measure this imperfection. To be definite, the p.d.f. of the observations $f(o)$ results from a convolution integral between the p.d.f. $\phi(p)$ of the four-momenta $p$ defined by physics, and the acceptance $a(o/p)$ of the detector

$$ f(o) = \int a(o/p) \phi(p) \, dp . $$

The object of experimental physics is to estimate $\phi(p)$ from observation of $f(o)$ and calculations of $a(o/p)$. In full generality, for non-trivial cases, the problem cannot be solved since it is ill-conditioned. As is often the case, we have two methods: the first one is the good one but it is difficult to apply; the second one is not very good but easier to apply, hence widely used.

6.1 Distorting physics

We start with an assumption about the underlying physics $\phi(p)$. Then we estimate the expected results through the convolution integral mentioned earlier, and compare it with the actual observations, with possible estimates of some free parameters in $\phi(p)$ (using classical statistical methods). In the case of clear disagreement, we must modify the assumption $\phi(p)$, but in the case of agreement we may infer only compatibility, not rightness of hypothesis (the problem is indeed ill-conditioned). The main difficulty of this method is that on the one hand we must have a complete description of physics through $\phi(p)$, and on the other hand the calculations must be done with a statistical precision at least as good as that of the experiment proper.

6.2 Undistorting observations

Let us pretend that $o \equiv p$, at least on the average, for this phase of calculations. Then we may say that an observed event is worth a $^{-1}$ produced events ($a \leq 1$ by definition). This method is often the only one feasible but, unfortunately, it has many drawbacks. This weighting method is not statistically sound as we will see later, but at least it deserves some care. In these calculations we have lost track of the difference between the variables $o$ and $p$. So, useful information such as the resolution must be estimated independently. In the phase-space regions where the detector is blind, infinite weights should be applied to absent observations. This difficulty may be partly solved by symmetry arguments. For this, we need factorization $\phi(p) = f(x) \cdot g(y)$, where, say, $x$ is one variable and $y$ represents all the other variables. In addition, we need to know the function $f(x)$. Finally, for all regions spanned by $y$, there must be a region spanned by $x$ where the detector is not blind. Then, for each $y$ value, we will compute an over-all factor.
\[ n(y) = \int_{\omega(y)} f(x) \, dx / \int f(x) \, dx, \] where the first integral is restricted to the x region \( \omega(y) \) where the acceptance \( a(x,y) \) is non-zero. Finally, we use a weight \( \left[ n(y) \cdot a(x,y) \right]^{-1} \) for each \( (x,y) \) observation. The x distribution will have the right shape in the region \( x \in \omega(y) \), scaled up by \( n^{-1} \) to take into account the blind region. Typically, \( x \) may be the azimuthal angle around unpolarized beams, in which case \( f(x) = 1/2\pi \). If these conditions are not satisfied, then the experiment must be restricted to the observable \( y \) region.

6.3 Exclusion of large weights

When events of different weights are considered, we can estimate the number of equivalent events by \( n_{eq} \sim (Depsilon)^2 / D\sigma^2 \). If \( n \) is the number of events, then we can easily verify that \( n_{eq} \leq n \),

\[ \frac{n}{n_{eq}} = 1 + \frac{1}{n} \sum_{i=1}^{n} \left( w_i - \frac{1}{n} \sum_{j=1}^{n} w_j \right)^2 \geq 1. \]

Coming back to the first expression, if one event has a weight much larger than the weights of the other events, then \( n_{eq} \ll n \). For example, one event of weight \( 10^3 \) and 999 events of weights 1 give \( n_{eq} \sim 4 \) for \( n = 10^3 \). Since the statistical precision is governed by \( n_{eq} \) (uncertainty varies like \( n_{eq}^{-1/2} \)), we may question the use of that one event which ruins the sample.

Assume that we want to estimate a parameter; that is, the function \( \tilde{g}(p) \) which describes physics has one free parameter. Then we may compare the two methods in terms of the variance \( \sigma^2 \) of the parameter estimated. If we use the first method ("distorting physics"), then we can show that every event contributes individually to decrease \( \sigma^2 \) (the information of the sample about this parameter, \( 1 \sim 1/\sigma^2 \), is the sum of the information of each event, each piece of information being a positive number). If now we consider the second method, which uses weights, the situation may be very different. In statistical terms, this method is not optimal and \( \sigma^2 \) may increase when a large-weight event is added to the sample contrary to the intuitive feeling.

This is of fundamental importance. One is naturally not inclined to discard events, but such a conservative attitude is sometimes opposite to one's interest! For a more detailed discussion see: W.T. Eadie et al., Statistical methods in experimental physics (North-Holland Publ. Co., Amsterdam, 1971).
1. Super-computers

1.1 Use and classification of super-computers

Up till quite recently the highest speed commercially available computers such as the IBM 7090, the CDC 6600 and the IBM/360-91 in the 1960's, and the CDC 7600 and IBM/360-195 in the early 1970's had a few attractive characteristics which were taken for granted:

i) They represented the most cost effective way of providing computing (if you could afford them); Grosh's "law" that the price only grows with the square root of the speed (within the same generation of computers) was conveniently observed by the computer manufacturers' pricing manuals.

ii) The software systems provided were general purpose: COBOL, FORTRAN, multiprogramming systems, sophisticated data set handling, terminal support, etc.

iii) The speed gains were transparent to the end-users who simply saw their programs running 50 times faster on the CDC 7600 than they did a decade earlier on the IBM 7090.

For economic and technological reasons these qualities of very high speed computing are all practically non-existent today. The fastest general-purpose computers available from CDC and IBM are the CDC Cyber 176 (about the same speed as the CDC 7600 first delivered in Autumn 1969) and the IBM 3033 which is somewhat slower than the IBM 360-195 (first delivered in 1970). The Amdahl 470/V7 and the Fujitsu M-200 are reputed to be somewhat faster than the IBM 3033, but even so, there has not been any significant increase in the speed of the largest available general-purpose computers for about eight years. Most computer users have found this quite tolerable and have covered their needs for additional capacity by installing multiple mainframe systems. With today's software and hardware facilities allowing sharing of data sets, and remote access from terminals and batch stations, it is not too difficult to provide a good service on such systems and the effect of one (out of three) mainframe going out for repair or software tests is (mostly!) easy to bear compared with having your one and only super-computer going down for the rest of the prime shift.

The interest in other high speed computing than this sort of multiple mainframe installation comes mainly from two types of computer users:

a) those with a total need for computing power five times and beyond a CDC 7600 or with very large problems which cannot be split and have to be solved within a fixed time frame (e.g., weather forecasting);

b) those with a relatively modest general-purpose installation, but with a need for real high speed computing at a low acquisition price.

1.2 The super-computers of recent years

For the first category of users mentioned above there have been four different computers marketed over the last five to seven years:

- the CDC String Array processor (STAR-100);
- the Texas Instruments Advanced Scientific Computer (ASC);
- the Cray Research Inc. computer (CRAY-1);
- the Burroughs Scientific Processor (BSP).

These computers are all capable of producing floating point results ten times faster than a CDC 7600, given the absolutely right problem, and have been costing $10-15 million for a system complete with front-end computers and indispensable peripherals. Another common "feature" is that less than 10 of each have been built till now. Here, however, the similarity stops and although some of the technical differences are discussed in detail later, it seems useful to make a few remarks on each one here to situate them better.

1.2.1 CDC STAR-100 \(^{1)}\) (Figs. 1-13). CDC started to design the STAR-100 in 1964 and delivered the first machine in 1974. Altogether three systems have been installed outside CDC so far [Lawrence Livermore Laboratory (2), NASA Langley Research Centre]. The STAR-100 has a multiprogramming operating system, virtual memory, and handles peripherals and terminals using rather independent "station computers" linked to the main computer via high speed channel-to-channel links. The STAR-100 runs on a 40 ns CPU cycle and has up to 1 Mwords (64 bits
plus parity) of core memory (1.28 us cycle time). The scalar part of the CPU works mainly on a set of 256 general-purpose registers using a three-address order code. The vector part of the CPU typically streams one or two operand vectors from memory through a floating point "pipe" generating a result vector in memory. The elements of the vectors all have to be consecutive in the memory owing to the fact that the relatively long access time of the core memory is compensated for by having it organized in superwords of 512 bits each. Once such a "stream" is started up, however, a 64-bit result may be generated every 40 ns. Since there are two floating point pipes and they can be used in overlapped mode, and since each pipe can generate 32-bit results at the rate of two per cycle, we get a top speed of one floating point result per 10 ns -- or 100 Millions of Floating Point operations per Second (MFLOPS).

1.2.2 **TI ASC**. Texas Instruments started the ASC project in 1968 and delivered the first machine in 1973. Three systems have been installed outside of TI (Geophysical Fluid Dynamics Laboratory in Princeton, Systems Development Corporation, and Naval Research Laboratory) and TI terminated the project in 1976. Several machines are in use within the TI service bureau specialized in seismic data processing. The ASC has a complete multiprogramming operating system somewhat similar to the IBM 6500 MVT and it uses a mixture of direct channels and eight integrated peripheral processors (somewhat akin to those of the CDC 6600) to control the input/output equipment.

The ASC runs on an 80 ns CPU cycle and has up to 1 Mwords (32 bits plus error correction bits) of semiconductor memory (256 bits/chip) with a 160 ns cycle time. The scalar part of the CPU works with 16 base address registers, 16 general arithmetic registers, and eight index registers. The vector part of the CPU typically streams one or two operand vectors from memory through a general-purpose "pipe" generating a result vector in memory. ASCs may have one pipe, two pipes or four pipes with either one instruction processor or, in the case of a four-pipe machine, two instruction processors each tied to two pipes. The elements of the vectors have to be spaced regularly in memory but one is not limited to an increment of 1. The actual vector instruction is a rather complex affair that can handle indexing corresponding to three nested FORTRAN DO-loops, provided the inner loop has an increment of 1 (or -1!). Each pipe generates a 32-bit result every 80 ns cycle, thus achieving a maximum of 50 MFLOPS on a four-pipe machine.

1.2.3 **CRI CRAY-1**. (Figs. 14-27). Cray Research Incorporated was established in 1972 with the initial goal of designing a large scientific computer, the CRAY-1. The first CRAY-1 was delivered in March 1976 to Los Alamos Scientific Laboratory. At present about six CRAY-1s are installed and several more are on order. Among the more recently installed systems the following sites can be mentioned: National Centre for Atmospheric Research (Boulder, Col., USA), United Computing Services (Kansas City, Miss., USA), Magnetic Fusion Energy Project (Livermore, Calif., USA), European Centre for Medium Range Weather Forecasting (Reading, UK).

The CRAY-1 runs under the control of a simple multiprogramming operating system and is limited by software to using only high speed disks as its direct peripheral equipment. It requires a general-purpose "front-end" computer with a channel-to-channel link to pass to it jobs, permanent files from disks and magnetic tape files, and to absorb the various output files. All these external files are copied to and from the CRAY-1 disks before and after job execution; this copying operation is normally referred to as "staging".

The CPU cycle time of the CRAY-1 is 12.5 ns and its memory can be up to 1 Mwords (64 bits plus 8 error correction bits) of 50 ns semiconductor memory (1024 bits/chip). The scalar part of the CPU works in three-address mode with eight general-purpose arithmetic registers and eight address or index registers. A second level of register memory, only available for transfer instructions to the above-mentioned registers, provides 64 64-bit and 64 24-bit registers. The vector part of the CPU operates on eight "vector registers" each of 64 words (of 64 bits). These vector registers are loaded from memory and restored to memory when necessary and the typical vector instructions use two of them as input and one as the destination for operations dealing with vectors of up to 64 words. Longer vectors must be broken down into pieces for processing. The memory locations which are brought into a vector register need not be consecutive but must have constant spacing.

In vector mode the CPU can produce a result every cycle, or 80 MFLOPS, but, owing to parallel instruction execution and a large degree of autonomy between the scalar part of the CPU and the vector part, certain inner loops have achieved up to 158 MFLOPS or beyond.

1.2.4 **Burroughs 3SP**. (Figs. 28 and 29). The Burroughs Scientific Processor was announced in
early 1977 and deliveries are only expected by 1979 at the earliest. It is conceived as an integrated "back-end" processor for a Burroughs 7800 (or 7700) with which it communicates almost uniquely in terms of files (or rather "chapters" of files) which reside on a file memory of up to 64 million 48-bit words. This file memory is constructed using 64 Kbit charge-coupled device (CCD) technology and is accessible both from the B7800 and the BSP. There is no other input/output device for the BSP and it runs a single user program at a time.

The BSP CPU consists of a scalar processing unit (SPU) which operates, using an 80 ns cycle, on 16 48-bit general-purpose registers. The SPU also uses 16 120-bit wide vector description registers. The parallel processor contains 16 Arithmetic Elements (AEs), which execute floating point operations in "lock step" (i.e. each element is executing the same instruction but on different data) at the speed of one result per 320 ns, giving a top speed of 50 MFLOPS. The vector instructions for the AEs are microprogrammed and very complex; up to five operands in one instruction, several different operators, field manipulation and editing operators and, even, special instructions for FORTRAN FORMAT conversion!

The BSP is the only announced super-computer which is based on the ILLIAC IV 3) experience. ILLIAC IV was conceived in the early 1960s at the University of Illinois as a machine with four "quadrants" of 64 identical processors each of which would execute in lock step on the same memory. Only one machine was built and with only one quadrant which has been in use at the NASA Ames Laboratory (Calif.) for several years 4).

1.3 A couple of recent array processors

For the second category of users mentioned in Section 1.1 there is a reasonably important number of "attachments" available for specialized high speed computation. The relative inexpensiveness (ranging from $100,000 and upwards) of such "array processors" normally results in a high degree of specialization with either very limited applications in mind or very special programming requiring a substantial investment. Seen from an architectural point of view we can split them into two classes:

i) those connected via a channel interface to the main computer,

ii) those which access the main memory of the host computer directly.

Each of these types will be discussed under the heading of an existing processor.

1.3.1 Floating Point Systems Inc., Array Processor AP-120B 5) (Fig. 30). In the mid-1960s, almost every large scale "scientific" computer such as the CDC 6000 series, IBM System 360 and UNIVAC 1100 series could be equipped with an "array processor" from the manufacturer to speed up simple floating point operations such as vector inner products. These processors were in considerable use for certain types of calculations such as, for example, seismic data processing. As the speed of large central processors has increased without the channel speeds increasing correspondingly it is now, however, getting too slow to send a couple of vectors out over a channel for an inner product operation; on a normal 1.5 Mbyte IBM S/370 channel it takes over 5 μs per pair of single precision numbers to send them out; much slower than to perform the operation on an S/370-168.

But, at the same time, semiconductor technology and microprogramming architectures have made it very attractive and cheap to build small, specialized processors which can generate 5-10 million results per second. The results have been that such array processors are now offered to perform relatively complete calculations in the form of, for example, a FORTRAN subroutine call: matrix inversion, complex to complex Fast Fourier Transformation, etc. Attached to a minicomputer this can be a very cost effective way of performing quite extensive calculations, in particular in the general area of "signal processing".

The FPS AP-120B is such an array processor of which several hundred units have been installed since 1975 and which is available with a channel interface to many minicomputers as well as to IBM S/370. It runs with a cycle time of 167 ns and has two independent arithmetic units in the CPU: A floating point multiply unit and a floating point adder. Both units are fully pipelined (i.e. can start a new operation every cycle) and can finish in three and two cycles, respectively. The operand memory is available in up to 500 Kwords (38 bits) with 167 ns cycle time, and the separate program memory is 50 ns cycle time, 4 Kwords of 64 bits each; a micro-instruction which controls both arithmetic units in a zero address mode needs 64 bits. There are 64 floating point registers and 16 operand address registers. An additional "table memory" of up to 64 Kwords of 38 bits is composed of read only memory and normal memory of 167 ns cycle time. When both arithmetic units are active the FPS AP-120B delivers two results every 167 ns or a
result rate of 12 MFLOPS -- quite an acceptable performance for a "box" which in a small configuration is comparable in price to that of a large minicomputer.

1.3.2 International Computers Ltd. (ICL), Distributed Array Processor (DAP) [8,9] (Figs. 31-35).
Another approach to make "array processors" cost effective with today's large-scale general-purpose computers is to have them operate directly on the main memory and thus to avoid the overheads involved in transmitting the data across the input/output channels. An important problem with this design is that of memory access conflicts: either the array processor has to be able to handle a delay in operand access thus greatly complicating the design and diminishing the efficiency or a way has to be found whereby the operand memory is left to the exclusive use of the array processor when it is active.

International Computers Ltd. have, since 1972, worked on such an integrated design inspired by the SOLOMON concept [10] and revived by J.K. Iliffe: the Distributed Array Processor (DAP). DAP was announced in early 1978 and will be available on the ICL 2900 series from 2960 and upwards. So far, one DAP is scheduled for installation at Queen Mary College, University of London, in 1979.

DAP consists, basically, of a matrix of 64 × 64 1-bit processing elements of which each has access to 4096 bits of storage. Each processing element can communicate with its four nearest neighbours (N, E, S, W) with a choice of "geometry" around the edges of the matrix. DAP is integrated into the host computer as 16 Mbits of normal memory, and it is perhaps conceptually easiest to view it as a box where each main memory word has its bits allocated to different, consecutive processing elements and where each processing element has a column of 4096 bits available representing bit number n in 4096 different 32-bit words. The DAP has to be programmed to perform floating point arithmetic either by having, for example, 32 consecutive processing elements working together on a main storage word, or by having all 4096 processing elements working on words stored in the columns. All processing elements execute in lock step (i.e. the same instruction), but can have different offsets within their "column" of 4096 bits and can be "inactivated" individually. Efficient algorithms exist for the "transposition" of numbers from columns to rows and vice versa.

DAP executes at a cycle time of about 200 ns, which makes it capable of doing 32-bit floating point arithmetic in times between 135 and 330 µs (addition and division) per processing element, corresponding to 12-30 MFLOPS. On table look-up or scanning operations it is very, very fast as it can be considered as an associative store in such applications. The interface between a FORTRAN program and DAP is on a double subroutine level: the user program calls subroutines written in DAP-FORTRAN (operating only on data in COMMON blocks which are allocated to the DAP memory) which in turn is practically interpreted using a large set of general-purpose DAP routines written, laboriously, at the level of the individual processing elements.

2. The Architecture

2.1 Technology, software and other limitations

What are, then, the problems in providing cost effective "super-computing" and what are the solutions proposed by the various pioneering vector computers currently available? The three large categories of problems are:

- the technology improvements are no longer sufficient to improve computing speed by significant factors -- a CDC 7600 delivered in 1969 had a cycle time of 27.5 ns and a redesign in today's technology would probably find it very hard to get below 10 ns. Hence, to get speed improvements of factors 5 and 10 one needs to explore new architecture.

- When the architecture cannot be that of a family (e.g. CDC Cyber series or IBM S/370) then the new computer needs a completely new operating system and compilers. The market for these supercomputers is relatively small; even of the CDC 7600 and the IBM S/360-195, which were general-purpose super-computers, only about 60 and 20, respectively, were installed. Hence one needs to find ways of reducing dramatically the cost of developing the operating system and the compilers.

- Within the CPU itself so many levels of buffering and "pipelining" have to be explored that interruptions such as context switches or simply strictly unvectorizable sequences of instructions can represent very significant overheads. In certain designs where vector operations are executed at the rate of 5-10 times that of a CDC 7600, scalar work may be 2-5 times slower than on a CDC 7600, such that the time of the previous inner loop becomes negligible in comparison with, for example, the task of formatting some data for output in graphical form.

In the following sections these various problem areas are discussed and an attempt is made to analyse
the solutions adopted in order to isolate significant trends.

2.2 CPU and memory architecture

2.2.1 Memory speeds and sizes. The progress in memory technology has been most decisive for getting higher speeds at a reasonable cost and reliability. Today, the CRAY-1 is delivered with up to 1 Mwords (64 bits) of 50 ns bipolar memory and it will probably be possible quite soon to reach 4 Mwords using 4096-bit memory circuits. With 16 independent memory banks, the memory bandwidth is 320 million 64-bit words per second -- quite sufficient to feed the CPU and keep the input/output going without conflicts.

On the CDC STAR-100 of much older conception a memory bandwidth of 200 million 64-bit words per second on sequential accesses is achieved by using very long words (512 bits) in the memory itself and by having 32 banks. The core memory cycle time is 1.28 μs, making it very slow just to access one operand at random. In both of these machines other limitations prevent the CPU from fully exploiting the memory bandwidth under normal circumstances.

In order to treat large problems efficiently on vector oriented machines it is necessary to have very large memories for the vectors treated, the intermediate result vectors and the logical condition vectors which often have to be used as a substitute for branches in order to gain speed. Hence a memory size of 1 Mwords has to be seen as the strict minimum of what is necessary today.

In order to increase the memory sizes to four, or even 16 Mwords without increasing the cost too much it will probably be necessary to go to Metal Oxide on Silicon (MOS) memories with a cycle time of between 100 and 200 ns. There will be no problem with bandwidth if there are enough banks and the problem of access time is not very severe owing to all the overheads which are involved anyway in getting a word, checking it, and storing it in a register; a total of 137.5 ns is needed on the CRAY-1 to access a word. So even with a memory of, for example, 150 ns cycle time, or three times slower, the real access time would only increase by 73%, to 237.5 ns.

The preceding discussion has concentrated on the various main memories, but it should not be forgotten that they all have a hierarchy of memories ranging from the operand registers and instruction buffers in the CPU through the main memory to larger backing memories which are still mostly disks, but where the BSP CCD memory of 1 ms access time, 75 Mbytes per second transfer rate and 64 Mwords maximum size may be the best indicator of what will happen in the future. It is not impossible either that the concept of a relatively small main memory (~100,000 words) might re-appear almost to be treated as a register memory by the program and that the "main memory" would be situated "behind" that one with a few million words.

2.2.2 Vector processing

A basic common design assumption for the supercomputers discussed is that a large proportion of the algorithms to be executed can be expressed as "vector operations", i.e. relatively simple operations to be executed (conceptually) simultaneously on a number of elements with fixed spacing in memory to allow prefetching of operands. Thus a single instruction may execute the equivalent of simple FORTRAN DO LOOPS like:

\[
\text{DO } 1 \text{ I}=1,N \quad \text{or even} \quad \text{DO } 1 \text{ J}=1,M
\]

\[
1 \text{C(I)} = \text{A(I)} \times \text{B(I)} \quad \text{DO } 1 \text{ J}=1,M
1 \text{C(I,J)} = \text{A(I,J)} + \text{B(J,J)}
\]

By expressing such operations in one instruction one eliminates the (large) overheads involved in all the instruction issues, decoding and branches (~15% of instructions executed in many programs) normally

| Table 1 |
| Memories |
| ASC | BSP a) | CRAY-1 | STAR-100 | CDC 7600 |
| Cycle time (ns) | 160 | 160 | 50 | 1280 | 275 |
| Word length (bits) | 256 | 48 | 64 | 512 | 60 |
| Banks | 8 | 17 | 16 | 32 | 32 |
| Max. size (Mbytes) | 4 | 48 | 8 | 8 | 0.5 |
| Bandwidth total (Mbytes) | 1600 | 600 | 2560 | 1600 | 270 |

a) Parallel memory, only for vector operands.
necessary for such a loop and one tells the hardware well in advance where the operands are to be found. This predictability allows the vector processors to prefetch operands from memory and thus achieve the rate of tens of millions of results per second.

The two major techniques used in processing operands of a high rate in an arithmetic unit are:

- streaming operands through a pipelined arithmetic unit capable of producing a result every CPU cycle, and
- making multiple processors work on the same instruction (lock step) but on different operands.

Streaming. The method most commonly adopted today for the processing of vectors uses as short a cycle time as possible in the CPU and streams operands through generating about one, two or four results per cycle, depending on the computer, in the straightforward case of element-by-element multiplication. The CPUs are constructed such that, although a floating point multiplication may take many cycles (seven on the CRAY-1), each step is independent of the previous one allowing a new set of operands to be passed to the CPU every cycle ("pipe-lining").

An advantage of this approach is that it is conceptually simple to think of it as a sequence of scalar operations and that it can be implemented in such a way that work on short vectors or even scalars can be handled at a relatively small loss in speed -- on the CRAY-1 scalar work is only 4 (addition) to 7 (division) times slower than work on long vectors.

An important disadvantage of the approach taken by the ASC and STAR-100, where the operands are fetched from memory and the results stored back into memory, is that it requires a significant "set-up" time of each vector instruction to get going -- tens of CPU cycles normally. This is partly due to the complicated memory access hardware which has to make sure it can guarantee a pair of operands per cycle out of the memory once the execution of the instruction starts, and the result is that top speed is only obtained on very long vectors and that multiple pipes, like on the ASC, become inefficient to use on relatively short vectors (the break-even point is around 100 elements). Since many programs spend a lot of computing time in calculations on short vectors this represents a major handicap.

The CRAY-1 achieves good performance on short vectors by making the vector operations register-to-register (8 vector "registers", each of 64 words, are treated exactly like accumulators on conventional computers). This is an interesting approach since it also limits the length of vector operations to a point where one does not need to interrupt them because of input/output activities but can wait for completion instead. Interrupts in the middle of the execution of one instruction create very significant logic problems on a machine like the STAR-100 but cannot be avoided owing to its virtual memory architecture. The weakness with the CRAY-1 approach is, of course, that peak performance in sequences like:

- load first operand vector (≤ 64 words) to a vector register
- load second operand vector (≤ 64 words) to another vector register
- multiply into a third vector register
- store the result vector into memory

require four machine cycles per result rather than one on the STAR-100 or the ASC. But in many cases several arithmetic operations can take place in the registers without having to store the intermediate results. In such cases, and provided the instructions are consecutive, the subsequent operations start as soon as the first element of the result vector is available in the result register, thus minimizing the overheads by a "chaining" process.

Another point to be remembered about pipelined CPU designs is that none of them currently provide for a fully pipelined division; a drop in speed of a factor of four compared to multiplication is more or less the rule. The CRAY-1 provides a fully pipelined approximate reciprocal (14 cycles) which has to be "improved" by one more explicit iteration and the other designs use subgroups of iterations within the pipeline design.

Multiprocessor. The BSP provides for 16 general-purpose arithmetic elements which execute in lock step. The DAP provides 4096 1-bit arithmetic elements executing in lock step -- the equivalent of about 850 of the 48-bit BSP arithmetic elements. This approach allows the duplication of relatively slow hardware, 320 ns to perform an arithmetic operation on a BSP arithmetic element, 200-300 µs on a DAP one, and still achieves very high performance on vector operations.

The major advantage of this approach is that future speed improvements still seem possible both by having more elements and by technology improvements, since we are in a range of technology where the speed of signal propagation is not yet the limiting
factor. This is a different situation from that of pipeline processors, where one is now close to physical limits (12.5 ns cycle time for the CRAY-1) and where speed improvements will require architectural changes resulting in a significant amount of software development and conversion costs.

The most striking disadvantage is that the speed of calculations which refuse to be "parallelized" will be relatively very slow: a factor 16 on the BSP and orders of magnitude on the DAP. This will certainly limit the immediate applicability of these machines compared to a CRAY-1, which is already more than two times faster than a CDC 7600 on completely scalar problems. On the other hand, unless new superminiaturized technologies become commercially available for the development of large computers the only way to provide significantly faster computers in the future will be the effective use of multiple processors on single problems.

2.2.3 Scalar processing. It is easy to see that the ultimate performance of a super-computer on a real program depends critically on the speed at which it processes the parts of the code which are non-vectorized. Table 2 compares hypothetical supercomputers with the CDC 7600 as a function of the performance gain on vector and scalar processing and the percentage of the code execution time which can be vectorized.

<table>
<thead>
<tr>
<th>Vector/scalar speed relative to the CDC 7600</th>
<th>Degree of vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50%</td>
</tr>
<tr>
<td>15/0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>15/0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>15/1</td>
<td>1.9</td>
</tr>
<tr>
<td>15/2</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Even for typical super-computer applications it can be hard enough to achieve more than 50% vectorization on complete programs and it is extremely rare that one can get to 90%. The importance of fast scalar processing in achieving a significant performance improvement over a large general-purpose computer like the CDC 7600 is thus quite clear.

In order to situate the numbers used in this table a little bit better it should be noted that the speed factor of 15 for vector processing is quite optimistic for the super-computers of today; but it is not a very critical parameter. On the scalar performance the STAR-100 is generally thought to be quite close to 0.2, the CRAY-1 about 2 and the ASC about 0.5. It is difficult to give an estimate for the SSP yet, but it seems as if its specialized scalar processor might bring it into the general area of about half a CDC 7600 on scalar work. All these numbers are very rough estimates based on a few applications only.

The way of achieving best possible scalar performance is to have effective instruction buffering and separate hardware for scalar and vector processing so that vector and scalar instruction execution can proceed in parallel.

The instruction buffering is advanced enough on most super-computers to allow several separate but small loops to be kept in the buffers simultaneously but since many loops are quite big and also because context switching invalidates the contents of instruction buffers it is important that the buffers are filled rapidly from the main memory. It is worth noting that the CRAY-1, which is the only super-computer currently to exceed 7600 scalar performance, has its widest memory path to its four independent instruction buffers: 16 full words are transferred in one 50 ns memory cycle. Branch look-ahead with instruction decoding of several possible sequences like on the IBM 3033 is not used in any of these computers; generally instruction issue and decoding stops as soon as one instruction cannot be issued to an operand access or other conflict.

The question of separate hardware for scalar processing on a vector-oriented computer is a particularly difficult one, since it is necessary to decide the level at which the separation between vector and scalar processing takes place: at the one extreme the AP-120B and DAP are array processors only, and scalar work is expected to be handled in the host computer. At the other end of the spectrum the CRAY-1 processes vector and scalar instructions in any mixture using both some common registers and some common functional units; reservation hardware is implemented to make sure that instruction issue stops if there is a conflict. The BSP has a separate register-oriented scalar processor which can handle real calculations and help to prepare the vector instructions for the 16-element array processor.

The trend seems to be in the direction of relatively well separated processors within the same
super-computer for the different sorts of processing required. This will certainly impose some additional burden on the compiler writers and, most likely, on the end user who will be encouraged to structure his calculations to separate scalar and vector work already at the source code level.

2.3 Input/Output

When considering input/output for a super-computer there are three important problems which need to be resolved:

i) The production of the software necessary to handle general-purpose input/output (i.e. disks, tapes, terminals, card readers, remote job entry, printers, mass memories) is cumbersome and very expensive.

ii) The amount of interrupts ("context switches") generated by a large volume of input/output and the scalar-type code in the operating system needed to deal with them represent a danger to the ultimate performance of the super-computer.

iii) The aggregate input/output rate ("bandwidth") has to be very high owing to the large computing potential.

2.3.1 Software and costs. The total market for computers of the category we are discussing here is, currently, less than 10 per year distributed over several manufacturers. With their very special architectures made necessary by the emphasis on speed it is not possible to develop super-computers to be able to use the same operating system, compilers, etc., as other computers produced in large quantities. It is, on the other hand, necessary to develop a sophisticated FORTRAN compiler which recognizes and vectorizes DO-loops, gives informative hints on code restructuring which would allow more efficient execution, etc. It should perhaps be noted that all manufacturers have, so far, developed FORTRAN compilers rather than pushing new languages.

The only way then to cut costs and limit the time needed for the production of software of a super-computer is to restrict the variety of input/output devices and record formats available. Although both the STAR-100 and the ASC have been delivered with a rather full complement of input/output devices and can support all the classical peripheral devices and a variety of access methods, this may prove impractical in the future. Already the CDC 7600 needs a general-purpose front-end computer for all other input/output than tapes and disks and the CRAY-1 only supports high speed disks. The BSP has its CCD memory as the only input/output device (which is shared with the B 7700/7800 front-end) and the DAP is integrated as a piece of shared main memory in its ICL 2900-series host. The software effort for monoprogrammed machines like the BSP and DAP can then be restricted almost only to FORTRAN cross-compilers (executing on the respective hosts) and FORTRAN run-time input/output routines. The CRAY-1 is in an intermediate position with a multiprogrammed system and software to communicate with a front-end as well as disk support -- but the less than 50,000 lines of code is still very acceptable cost-wise, compared to the millions of lines in a system like MV.

2.3.2 Context switching. The overheads involved in context switching and input/output processing can be illustrated by the example of the CDC 7600 installed at CERN which is running a general scientific workload in terms of input/output frequencies. A study in 1976 11 gave the following figures: 3 ms of CPU time between GET/PUT operations and 250 μs average CPU time in the supervisor to deal with a GET/PUT.

For one particular application, FORTRAN compilation, there was a GET/PUT request for every 360 μs of CPU time. Thus we find an average of about 330 context switches per second, and about 8% of the CPU time spent on input/output in the operating system. If we extrapolated these figures to a CRAY-1 which is generally rated to be five times faster than the CDC 7600 we would get 1650 context switches per second (assuming an unchanged length of the average GET/PUT). An "exchange jump" takes about 1.5 μs on the CDC 7600, but because of the almost 700 registers in the CRAY-1 CPU an exchange jump could easily have been 10 times slower thus creating a potential performance problem. The design solution adopted on the CRAY-1 is to do only a partial context switch in the hardware (612.5 ns) the exchange jump instruction exchanges the contents of the 16 scalar registers, the P-counter and a few other indicators, and it is left to the operating system to exchange the contents of the vector registers, the extra buffer registers, etc., if it wants to give control to another user program. The operating system can be programmed to use only the scalar registers (or save/restore any other registers it wants to use) and thus the overhead for an input/output interruption is kept down to a minimum.

On monoprogrammed machines like the BSP or, in some sense, DAP, the time spent in context switching for input/output processing will be small but the overheads will not disappear entirely: setting up the jobs, creating a structure where certain routines are
completely vectorizable, waiting for the host computer to provide the data in memory or on the CCD file, etc.

2.3.3 Maximum speed input/output. The total input/output bandwidth for the super-computers under discussion is, in all cases, very impressive when measured as the maximum number of bytes per second which can be passed between the channels and the main memory (or between the CCD file and the main memory on the BSP). Thus for computers like STAR-100, CRAY-1, ASC and even the CDC 7600, total input/output memory can be sustained at much higher peak rates than the peripheral devices can supply data. The peripherals are, of course, generally disks which operate at a peak rate of about 4 Mbytes per second but which have latencies of the order of 50 ms). The limitations on peak interrupt rates furthermore restrict the number of parallel disk streams to three on the CDC 7600 and eight on the CRAY-1 for instance. This corresponds to a maximum of about 32 Mbytes per second on the CRAY-1 where the central memory access input/output rate would allow 20 times this rate. The BSP uses a CCD file for input/output which is designed to achieve 75 Mbytes per second transfer rates with 1 ms latency -- this high speed disk replacement is certainly a very interesting feature of the whole BSP design.

3. The Programming

3.1 Vectorization, parallel processing

To write a program which executes fast on an array processor it is necessary to "think parallel" in the first place; algorithms have to be chosen in such a fashion that the bulk of computing can be expressed as identical operations on many elements. If we use the FORTRAN DO-loop as an example one has, basically, to avoid IF statements, not to call EXTERNAL FUNCTIONS or SUBROUTINES (which may have side effects), use simple indexing representing equally spaced elements in memory, and avoid formulations where an iteration depends on the results of the previous one [i.e. A(I) = f(A(I - 1))].

Special limitations on certain array processors are:

- STAR-100 demands a spacing of one between elements,
- CRAY-1 must produce a vector (not a scalar) as the result in the inner loop (i.e. the matrix multiplication \( C_{ij} = A_{ik} B_{kj} \) must not be written with \( K \) as the index in the innermost DO-loop),
- DAP requires two-dimensional matrices to be of the dimension 64 \( \times \) 64,
- ASC demands that the inner DO-loop increment in the case of a 3-level deep nest is +1 or -1 etc.

There are also particular strengths of some of the computers which can give rise to very efficient code generation:

- STAR-100 has single instructions for such complicated calculations as polynomial evaluation, square root, difference calculations and even for the transposition of an \( 8 \times 8 \) matrix,
- the BSP compiler is planned to be able to vectorize both loops with subscripts of the form \( A(I(J)) \) and to resolve iterations of the form \( A(I) = f(A(I - 1)) \) into expressions in \( A(1), A(17), \) etc. Here advantage is taken of the multiple processor approach which is more flexible than pipelining for these cases.

3.2 Vectorizing FORTRAN (Figs. 36-42)

All the manufacturers of super-computers discussed here have chosen to invest considerable effort in developing FORTRAN compilers which are capable of recognizing DO-loops (and IF-loops) which can be translated to vector instructions. The ASC compiler\(^2\) was definitely the pioneering effort and it compiles excellent code if the FORTRAN is written in a way "friendly" to the computer. Vectorizing compilers for the CRAY-1 and STAR-100 are also very well developed now.

Another approach which is being tried, partly in order to ease the transition from a non-vector machine, is to have a (FORTRAN) program which analyses all loops and uses a comprehensive set of vector subroutines, about 75 in all, to output another FORTRAN program where loops are, as far as possible, replaced with calls to these subroutines\(^3\). Implementation of the subroutines is then done in machine language on such target computers as STAR-100, CRAY-1, CDC 7600, ILLIAC IV. This approach allows the development of a very sophisticated general analyser program which is independent of the target computer; on the other hand some inefficiency is introduced by going via SUBROUTINE calls in all cases.

Whichever approach is used to vectorize, the FORTRAN program which is written for efficient execution on a vector-computer is very hard to read. In order to avoid tests within loops, "state-vectors"
of 0's and 1's have to be generated which are then used within the arithmetic to generate the correct results. Non-ANSI intrinsic functions are provided on the CRAY-1, for example, to avoid the specific generation of state vectors. A primitive example:

```fortran
DO 10 I=N,M
  X(I)=A(X)
  IF(B(I).GT.C(I)) X(I)=D(I)
10  CONTINUE
```

is to be rewritten as follows to become vectorizable:

```fortran
DO 10 I=N,M
  X(I)=CMVM(D(I),A(I),C(I)-B(I))
10  CONTINUE
```

The CMVM function will, after compilation, only be called once per 64 elements (the maximum number of elements handled in a CRAY-1 vector instruction), will do the test of C(I)-B(I) and the subsequent merge operations with a few vector instructions and a temporary state vector with one bit per element.

Thus we notice that FORTRAN has a number of drawbacks specific to its use for the expression of algorithms amenable to vectorization:

- all SUBROUTINE and EXTERNAL FUNCTION calls may have side-effects;
- purely syntactical choices, such as the sequence of indexing in nested DO-loops, may prevent vectorization in some cases;
- paucity of standard built-in functions for the handling of vectors and matrices;
- no auto-indexing expression (e.g. A(*) in PL/1);
- no encouragement to write code which is easily vectorized -- one "thinks" in terms of IF's and DO-loops.

If parallel (or vector-oriented) machine architectures are to become more widespread, it will be necessary to stop using a sort of distorted FORTRAN when trying to express parallel algorithms and bring a new language into use. The major reason for this is linked to the fact that the area of parallel machine architecture still is a research topic and that each new computer contains significant innovations and changes. In such an environment the importance of transportability and readability of programs is not to be demonstrated if the end user has to avoid repeated and costly program conversions resulting in discouragement with the whole approach of parallel computing. Since FORTRAN is poor in features to express parallel algorithms without information loss users are forced to use coding tricks and assembly language in order to solve problems efficiently. The end result is that the FORTRAN programs become non-standard, non-transportable and difficult to read. In such an environment it would be better to give up the illusion that one creates transportable programs for these new architectures by writing them using FORTRAN syntax when the semantics is hopelessly inadequate.

4. SUMMARY

The super-computers of today are becoming quite specialized and we can no longer expect to get all the state-of-the-art software and hardware facilities in one package. In order to achieve faster and faster computing it is necessary to experiment with new architectures, and the cost of developing each experimental architecture into a general-purpose computer system is too high when one considers the relatively small market for these computers. The result is that such computers are becoming "back-ends" either to special systems (BSP, DAP) or to anything (CRAY-1).

Architecturally the CRAY-1 is the most attractive today since it guarantees a speed gain of a factor of two over a CDC 6600 thus allowing us to regard any speed up resulting from vectorization as a bonus. It looks, however, as if it will be very difficult to make substantially faster computers using only pipelining techniques and that it will be necessary to explore multiple processors working on the same problem. The experience which will be gained with the BSP and the DAP over the next few years will certainly be most valuable in this respect.

Bibliography


References


2) W.J. Watson and H.M. Carr, Operational experience with the TI advanced scientific computer, AFIPS - Conference Proceedings, Volume 45 (1974).


4) C. Jensen, Taking another approach to supercomputing, Datamation, pp. 159-172 (February 1978).


7) Floating Point System, P.O. Box 23489, Portland, OR 97223, USA.


9) P.M. Flanders, D.J. Hunt, S.F. Reddaway and D. Parkinson, Efficient computing with the distributed array processor, in Kuck et al., (see [B] in Bibliography).


11) H. Lipps, DD Division CERN, unpublished study.


**CDC STAR-100**

**Development started:** 1964  
**Delivery:** 1974  
**5 produced:** Lawrence Livermore Lab. (2)  
NASA Langley  
CDC (2)  
**Timing:** Maximum 100 MFLOPS (32 bits) but vector start-up can be a significant overhead for short vectors  
Approx. equivalent to a CDC 6600 on scalar work.

![Fig. 1](image)

**STAR-100**

**CPU**  
40 ns clock period  
256 64-bit registers  
Two floating point pipelined execution units, one string unit  
Macro instructions  
64-bit, 32-bit, 8-bit, 4-bit, 1-bit vector instructions  
100 MFLOPS maximum (32 bits).

![Fig. 2](image)

**STAR-100**

**Core memory** 524,288 64-bit words (plus parity) in 32 banks  
Second memory possible  
1.28 µs cycle time for 512 bits  
(25 Mwords/s)  
3R,2W doubleword trunks (25 Mwords/s each)  
Virtual addressing, two page sizes, 16 associative registers compared in 40 ns, subsequently 20 ns/entry

![Fig. 3](image)

**STAR-100**

**INSTRUCTIONS**

32-bit or 64-bit  
Vector instructions have an implied stride of 1  
Sparse vector format: 0001101001 order vector 5, 6, 8, -4 data  
Scalar is 3-address register-register  
Unusual instructions:  
Transpose an $B \times B$ matrix in place  
Compare and generate order vector  
Sparse dot product  
$\sum A_N = A_{N+1} - A_N$, $C_N = (A_N + B_N)/2$, $C_N = (A_N - B_N)/2$, $CN = \sqrt{A_N}$  
Polynomial evaluation  
Translate A per B into C  
Floor, ceiling, absolute, truncate exponent  
Altogether 231 instructions

![Fig. 4](image)

**STAR-100**

1/0  
4-12 16-bit I/O channels  
5 Mbyte/s maximum channel speed  
Intelligent stations attached only

![Fig. 5](image)

**Context switching**

Invisible package (16 words)  
Register file (256 words)  
Monitor + Job: Exit force instruction  
Job + Monitor: Exit force  
Channel interrupt  
Storage access interrupt  
Illegal instruction  
Interval timer = 0  
Virtual addressing only in job mode

![Fig. 6](image)

**STAR-100**

**MCU**

Computer with disk, line printer, operator display system autoload, microcode load  
4 16-bit counters for performance monitoring  
For example:  
No. of branches in stack  
No. of CPU memory requests  
No. of vector instruction < 64 words  
No. of times a particular instruction  
No. of minor cycles delay due to operand  
Result conflicts  
Dew-point, freon pressure, temperature monitoring, compressor, power failure monitoring

![Fig. 7](image)

**CDC STAR-100, STRING ARRAY PROCESSOR**

![Fig. 8](image)

- 135 -
STAR-100

CPU OVERVIEW

MEMORY

VIRTUAL ADDRESS TRANSLATION 16 REGISTERS

DATA

WRITE BUFFER 16 WORDS

READ BUFFER 16 WORDS

INPUT/OUTPUT 64 WORDS

I/O CHANNELS

128 WORD BUFFER

128 WORD BUFFER

INSTRUCTION STACK 22 WORDS

REGISTER LOGICAL AND SHIFT UNIT

INTERUPT COUNTERS

LOAD/STORE UNIT

MICROCODE

STRING UNIT

FLOATING PIPE 1

FLOATING PIPE 2

ALL PATHS ARE DOUBLEWORD WIDE

Fig. 9

STAR-100

FLOATING PIPE 1

+, -. ± *, /, TRUNCATE, ADJUST EXPONENT,
CONTRACT, EXTEND, COMPARE; ADDRESS OPERATIONS

Fig. 10

STAR-100

FLOATING PIPE 2

+,-, *, /, TRUNCATE, ADJUST EXPONENT,
CONTRACT, EXTEND, COMPARE
REGISTER DIVIDE, BINARY/BCD CONVERSION
SQUARE ROOT, VECTOR DIVIDE, VECTOR
MULITPULY

Fig. 11

STAR-100

STRING PROCESSING

EDT (PACKED BCD)
LOGICAL (OR, AND, STROBE, INHIT, PIERCE)
BINARY ARITHMETIC (+, -, ±, /)
MOVE, COMPARE, MERGE, PACK, UNPACK

Fig. 12

STAR-100

INSTRUCTION BUFFERING

8 WORDS

8 WORDS

8 WORDS

1 WORD / 20 ns LOAD SPEED
CLEARS ON A BRANCH OUT
CONSECUTIVE ADDRESSES

Fig. 13
CRI CRAY-1
Development started: 1972
Delivery : 1976
7 orders: Los Alamos Scientific Laboratory
National Center for Atmospheric Research
European Centre for Medium Range Weather
Forecasting
US Department of Defence (2)
United Computing Services
Lawrence Livermore Laboratory (MFE)
4 "chip types", no capacitors, 3 in. wires,
8 gates/cycle, quite small.

Fig. 14

CRAY-1
CPU
12.5 ns clock period
585 64-bit registers, 72 24-bit ones
12 fully pipelined specialized functional units
80 Mips maximum
80 - 138 - 250 MFLOPS maximum (64 bits)
2's complement, no floating point divide, no
fixed point multiply.

Fig. 15

CRAY-1
Memory
1,048,576 64-bit words (plus seced) in 16 banks
50 ns cycle time, 137.5 ns access time
80 Mwords/s block transfers, 320 Mwords/s
to instruction buffers
Addressing offset by one fixed base address
register (set to zero in monitor mode)

Fig. 16

CRAY-1
Instructions 1
16-bit or 32-bit
Scalar is 3-address register-register
Vector is 3-address register-register (can use an
S register as one of the input streams),
VL determines number (1-64) of elements
No bit, byte, decimal facilities
Vector loads/stores are fixed strides, but not
only 1
VM register (64 bits) control merges of two vec-
tors (X-V, S-V, O-V, 3-V)
VM register is set from S or from one vector
(0, ≠ 0, pos, neg).

Fig. 17

CRA-Y-1
Instructions 2
Unusual features
Vector chaining
Some constants (0.5, 1, 2, 4)
Approximate reciprocal and 2-SG*SK instruction
Population count
Branch addressing to parcel (24 bits)
180 instructions

Fig. 18

CRA-Y-1
Delays
Functional units: only vector instructions
Registers: Operand and result register avail-
ability
(Note vector chaining however)
Only one result path to A, B, S and T
registers
Block memory copies
Vector load/store 8 or 16 word incre-
ment
(Note it prevents chaining)
V, B or T load/store wait for I/O to
stop and other block transfers to
finish
B and T register block copy stop
issue for 5, 6, or 14 CPs, cannot
issue themselves if there is an A
or S register reservation
Memory Scalar or I/O references conflicting
in rank B of the C, B, A memory
access network.

etc.

Fig. 19

CRA-Y-1
I/O
12 16-bit I channels and 12 O channels
One 64-bit register per channel
Maximum rates:
one channel 80 Mbytes/s
six channels 160 Mbytes/s
24 channels 640 Mbytes/s
Disk controllers or intelligent stations
attached only.

Fig. 20
CRAY-1

Context switching
Exchange package (16 words) 50 CPs
Monitor job: normal exit
Job monitor: error exit (000000)
normal exit (SVC)
memory error
floating point error
RTC interrupt
operand/program range
I/O interrupt
console interrupt
Note B, T, and V registers not saved

Fig. 21

CRI, CRAY-1

Fig. 22

CRAY-1

CPU OVERVIEW

MEMORY

FUNCTIONAL UNITS

Fig. 23
CRAY - 1
FUNCTIONAL UNITS 1

Fig. 24

ADDRESS
ADD
MULTIPLY

SCALARS
ADD
LOGICAL
SHIFT
POP COUNT

FLOATING POINT
ADD
MULTIPLY
RECIPIROCAL

CRAY - 1
FUNCTIONAL UNITS 2

VECTORS
ADD
MULTIPLY
RECIPIROCAL

FLOATING POINT
VECTOR (CONTROL : VL (VECTOR LENGTH)
VM (VECTOR MASK))

CRAY - 1
INSTRUCTION BUFFERING

MEMORY
4 WORDS EVERY 12.5 ns

- NO BRANCH IS 2 CP's.
- BRANCH IS 5 OR 16 CP's,
  EXCEPTIONALY, 25 CP's
  WITH INSTRUCTION SPLIT
  OVER TWO BUFFERS.
- 4 BUFFERS USED IN A CIRCULAR
  FASHION.
- EACH BUFFER STARTS ON A
  PARCEL ADDRESS DIVISIBLE
  BY 200
- EACH BUFFER IS INDEPENDENT
- NO LOOKAHEAD

Fig. 25

Fig. 26

- 139 -
AP-1208 (Floating Point Systems)

"ARRAY PROCESSOR"

Cheap, 167 ns cycle, 3 cycle MPY, 2 cycle add
Specialized with separate instruction and operand memory, double instructions for both pipes.
Scalar/vector no difference within the AP
500 Kwords data, 4 Kwords instruction memory,
6 Mips, 1-2 (compiler) - 8 (hand) -
12 (theory) MFLOPS.

Fig. 30
**ICL DAP**

*Development started:* 1972  
*Delivery:* 1979 (64 x 64)  
*Timing:* 200 ns cycle time, 4096 processors  
32-bit floating point:  
* + 135 µs approx. 30 MFLOPS  
* * 280 µs " 15 MFLOPS  
* * * 140 µs " 30 MFLOPS  
* * * * 350 µs " 11.5 MFLOPS  
* * * * * 200 µs " 20 MFLOPS  
*Memory:* Two storage modes, variable word length.

**Fig. 31**

**ICL DAP, DISTRIBUTED ARRAY PROCESSOR**

**Fig. 32**

**Fig. 33**

**Fig. 34**

**Fig. 35**

- 142 -
VECTORIZATION

FORTRAN or Assembler in practice

FORTRAN:
- complicated compiler analysis necessary to restructure. Even so depends on specific
  high speed computer
- Gains: 25-50% vectorization often easy; 50-70% often possible, hard work; > 75% exceptional

Fig. 36

VECTORIZING A FORTRAN LOOP

OBSTACLES AND TOOLS (1)

1. Lack of knowledge
   - Call subroutine
     - External function
     - Unknown index: M = J(I)
     - R(M)...
     - Tools: Fork and join (?)

2. Logical structure
   - GO TO
     - IF
   - Tools: Avoid GO TO's
     - Replace IF's by "state vector" type constructions ("DO for all except when")

Fig. 37

VECTORIZING A FORTRAN LOOP

OBSTACLES AND TOOLS (2)

3. Use of variables
   - Scalars used as temporaries
     - Vector results overlap with operands
     - Inner loop result not a vector
   - Comment
     - This is getting specific to the CRAY-1
     - Features on ILLIAC IV, ASC, STAR-100 would allow vectorization in some of these cases
   - Tools
     - Restructure using temporary vectors
     - Reorder the sequence of calculations
     - Split off the hopeless parts of loops

Fig. 38

THE VECTORIZER

(Research Development Associates)

FORTRAN - FORTRAN using calls to

DYADS:
- S = SUM(U(*)+V(*))
  - R(*) = S/U(*)
  - R(*) = U(IV(*))
  - R(*) = U(*) + V(*)
  - etc.

TRIADS:
- R(*) = SUM(U(*)*V(*))
  - R(*) = U(*) - V(*)
  - etc.

MONADS:
- S = SUM(U(*))
  - R(*) = S
  - R(3) = COS(U(*))
  - etc.

MERGES:
- TEST(W(*).GT.0).R(*)=U(*) OR V(*)
  - TEST(V(*).NE.0).R(*)=S OR -U(*)
  - etc.

Fig. 40
0. Introduction

The System Performance Optimization has become and important and difficult field for large scientific computer centres. Important because the centres must satisfy increasing user demands at the lowest possible cost. Difficult because the System Performance Optimization requires a deep understanding of hardware, software and workload. The optimization is a dynamic process depending on the changes in hardware configuration, current level of the operating system and user generated workload. With the increasing complication of the computer system and software, the field for the optimization manoeuvres broadens.

In the three hour lecture it is of course difficult to cover all aspects of the System Performance Optimization. First of all it was necessary to talk in the Chapter 1 about the hardware of only two manufacturers IBM and CDC. Chapter 2 contains the description of four IBM and two operating system. The description concentrates on the organization of the operating systems, the job scheduling and I/O handling. The performance definitions, workload specification and tools for the system stimulation are presented in the Chapter 3. Chapter 4 is devoted to the description of the measurement tools for the System Performance Optimization.

In this Chapter I am going to present software, hardware and hybrid monitors. The results of the measurement and various methods used for the operating system tuning will be discussed in the Chapter 5.

Unfortunately it was not possible to cover during the lectures theoretical aspects related to the System Performance Optimization. Therefore the author intends to cover the subject of the computer models and simulation in the separate publication.

1. Hardware Overview

In this Chapter I would like to present the examples of hardware used for the large scale scientific computations /batch and interactive/. The examples will be taken from CERN and INR computer centres, which use IBM and CDC equipment. It is well known, that scientific computations are almost monopolized by these two manufacturers especially in the field of physics. Assuming the elementary knowledge of computer structure, I will concentrate on the hardware aspects, which are crucial for performance and programming. Since the IBM and CDC computers differ very much in the organization, it is necessary to describe them separately.

1.1. IBM Hardware

Figure 1 shows IBM 370/168-3 computer, which is installed at CERN. The central processing unit is equiped with 3 megabytes memory, byte and block multiplexer channels, controllers and devices. Up to 16 controllers can be linked to a shared channel, and therefore the reconfiguration of the controllers linked to block multiplexer channels can be easily done manually. The 3333 and 3350 disk storage can be accessed from two independent storage controllers. Such feature is called dual access, and it causes a complicated routing of disk requests through the system.

The central processing units of higher models from IBM 370 line include a buffer storage. The buffer storage can sharply reduce the time required for fetching currently used sections of main storage. On the Model 165, for example, the CPU can obtain eight bytes from the buffer in two cycles /160 nanoseconds/, and a request can be initiated every cycle. This compares with 18 cycles /1440 nano-
seconds/required to obtain eight bytes
directly from main storage. On average,
the high-speed buffer storage operates
to make the effective system storage cycle
time one-third to one-quarter of the
actual main-storage cycle time. Buffer
operation is handled entirely by hardware
and is transparent to the programmer,
who doesn't need to adhere to any
particular program structure in order to
achieve close-to- optimum use of the buffer.

Very important hardware feature of
IBM 370 is the dynamic address translation
/DAT/. This feature is essential for
virtual storage operating systems. The CPU
can operate with their virtual storage
features disabled /Basic Control mode/ or
enabled /Extended Control mode/. For ease
in storage management, virtual storage,
real storage, and direct access storage
used to contain virtual contents, are
devided into contiguous fixed-length
sections of equal size. Virtual storage is
devided into 64K-byte segments. A maximum
virtual storage of 10,777,216 bytes,
therefore contains 256 segments. Each
segment of virtual storage is divided into
4K-byte pages. A page frame is a 4K-byte
block of real storage, that can contain one
page at a time. An equivalent of frame on
the direct access storage is called a slot.

In a virtual storage system,
a mechanism is required to associate
virtual storage addresses of data and
instruction with their actual location in
real storage. This function is performed
by DAT. To translate the addresses, DAT
uses tables in real storage. These tables,
which are maintained by the control
program, are the segment table and the
page tables. One segment table and
a corresponding set of page tables exist
for each address space in the system
/see Figure 2/. There is one page table
for each segment in the address space.
The page table indicates which pages are
currently in real storage and the real
storage location of those pages. DAT
translates the virtual storage addresses
contained in a instruction during execution
of the instruction.

First, DAT obtains the address of the
appropriate segment table from a system
control register. To this segment table
address, DAT adds the segment address bits
to obtain the segment table entry. Next
DAT obtains the page table address from
the segment table entry and adds the page
address bits to it in order to obtain the
page table entry. Finally, DAT forms the
24 bit real storage address by appending
the displacement to the page frame address.
To reduce the amount of time required for
address translation, DAT retains up to 128
previously translated addresses in a
translation lookaside buffer /TBL/. Prior
to performing a translation using segment
and page tables, DAT searches the TBL for
required address.

A program interruption occurs during
address translation, if DAT attempts to
translate a virtual storage address to
a real storage address and the required
page is not in real storage. This
interruption, called a page fault, alerts
the control program that the page must be
loaded from external page storage into a
page frame of real storage. The transfer of
a page into real storage is a page-in. The
page-in process is shown in Figure 3. First,
when a needed page is not in real storage
/indicated by a bit in page table entry/,
storage management automatically goes to
the corresponding entry in a external page
table. The external page table entry gives
the slot location for page.

Next, storage management selects
a frame in real storage to hold the
required page. To do so, it refers to the
page frame table, which indicates, which
frames are allocated. Storage management
finds an available frame and brings in the
required page from its slot in external
page storage. To complete the page-in
process, storage management updates the
appropriate page frame table entry and
page table entry.

In order to keep a supply of frames
available for page-in, the control program
removes pages from real storage that have
not been recently referenced. Prior to
removing a page from a frame, the control
program determines whether the page contents were modified during processing. If so, storage management performs a page-out. Otherwise an exact copy of the page already exists in external page storage. A page-out copies the modified page from its real storage to a slot. The slot need not be the one that contains the old version of the page. Storage management need only update the external page table entry to designate the new slot.

At the end of this section I would like to quote some characteristics of the Model 165. The Central Processing Unit has Basic Machine Time of 80 nanoseconds. Storage Cycle Time is 2 microseconds with the 8 byte Storage Access Width and four-way interleaving. High-Speed Buffer Storage can have 8192 bytes or 16384 bytes.

Block Multiplexor Channels are buffered to a width of 16 bytes for communication with storage. The maximum data rates for Block Multiplexor Channels is 3 million bytes a second.

Byte Multiplexor Channels are not buffered and they are used for slow peripheral devices. Finally, the 3300 disk has the average access time 30 ms, the average latency 8.4 ms and the data transfer rate 806 Kbyte per seconds.

1.2. CDC Hardware

In this section I will describe the architecture of CD 6000, CD 7000, CYBER 70 and CYBER 170 computers. The above computer families consists of central processors, central memory, peripheral processors, channels, controllers and devices. In comparison with IBM computers, peripheral processors form additional layer in the computer architecture introducing a high degree of multiprocessing. Peripheral processors perform a majority of system tasks, which usually take a lot of CPU time on computers with the standard architecture.

Figure 4 shows the configuration of CYBER-73 at the Institute of Nuclear Research. The configuration consists of central processor unit performing 1.2 million instruction per second, 96K central memory /60 bit words/ and fourteen peripheral processors with 4K memory /12 bit words/. The peripheral equipment may be attached to 24 channels with the maximum transfer rate 2 millions characters per second. Two types of disks are used at INR.

The characteristics of the disk are the following:

<table>
<thead>
<tr>
<th>Disk</th>
<th>CD 841</th>
<th>CD 844</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of units</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Unit capacity in millions of characters</td>
<td>36</td>
<td>118</td>
</tr>
<tr>
<td>Transfer rate in thousands of characters</td>
<td>179</td>
<td>461</td>
</tr>
<tr>
<td>Access Time /miliseconds/</td>
<td>135</td>
<td>55</td>
</tr>
<tr>
<td>Maximum</td>
<td>75</td>
<td>30</td>
</tr>
<tr>
<td>Average</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>Minimum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Rotational Latency</td>
<td>12.5</td>
<td>8.3</td>
</tr>
<tr>
<td>Dual access is provided for CD 841 disks.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Five Low Speed Batch Terminals and seven TTY are connected to Local Communications Controller through modems. Four PDP-11/45 minicomputers are linked to 6671 Multiplexer. Slow peripheral devices include two printers, Card Reader and Punch Paper Tape Reader/Punch, and Plotter. The devices form two separate lines linked to channels by means of controllers. Four 659 Magnetic Tape Units /9 track/ and one 657 Magnetic Tape Unit /7 track/ are served by one controller and channel.

Central Memory is organized in banks of 4,096 word each. The storage cycle in one microsecond, however the Central Memory address and data control mechanisms permit to move a word to or from Central Memory every 100 nanoseconds.

The Central Processor is interrupted by means of the Exchange Jump. This operation may be initiated by a peripheral processor or by the Central Processor. The effect of this operation is to interrupt the currently active central program and to initiate another program. To initiate the operation a PPU executes the Exchange Jump referring to the beginning of the
Exchange Jump Package. This package contains the 24 operating registers /X, A, B/, program address /P/, the relative address /RA/, field length /FL/ for Central Memory, and the Monitor Address /MA/.

If the CPU is "Monitor State" it may not be interrupted. In this state the Central Processor may set up and initiate jobs or tasks in a direct manner via the Central Exchange Jump instructions. When this instruction is executed the CPU state is switched to "program state". A user may also initiate a Central Exchange Jump instruction, however he is not allowed to set up the exchange package. The lower CYBER computers use for the Central Processor much more simpler interruption mechanism then IBM computers and the execution of peripheral processors can not be interrupted at all.

In the second part of this section, I would like to describe some features of CD 7600, CYBER 76 or CYBER 176 computers. Figure 5 shows the CD 7600 configuration at CERN. The computer has a 64 K small core memory /SCM/, and 512 K large core memory /LSCM/. Six high speed channels are used to connect the six 817 disk units, and another channel is used to connect the processor to eight 844 disks, four other channels are used to connect the maintenance control unit /MCU/, the first level instrumentation peripheral processor /FLIPP/ and the CDC front-ends. The RIOS stations are connected to the CDC 6500 and approximately 70 INTERCOM terminals can connect to either front-end.

In March 1978, Control Data announced a new model CYBER 176. The main difference between CYBER 176 and CD 7600 is the additional CYBER 170 Peripheral Processor Subsystem, which replaces MCU on CD 7600. Therefore CYBER 176 has two different groups of peripheral processors. High Speed Peripheral Processors are dedicated to the control of the 817 or 819 disk subsystems and Peripheral Processor Subsystem handles the peripheral equipment, which was on CD 7600 attached to the front-end.

The CYBER 173 Central Processor has a 36.3 MHz frequency clock which provides an internal minor cycle, or clock period, of 27.5 nanoseconds. The CPU contains a 12 words instruction stack with a two look-ahead feature plus the ability to execute contiguous or non-contiguous instruction loops which reduces the need to reference central memory in order to access the instructions.

Instructions are issued from the instruction stack at maximum rate of one per clock period to any of the nine functional units. Each functional unit is independent of others and all functional units are segmented, so that although it may take several clock periods for a given operation to be completed but several separate operations can be at various stages of completion within a single functional unit.

Unlike other CYBER models, the CYBER 176 contains an interrupt system designed specially to ease the handling of high speed I/O but at the same time, minimizing the system memory requirements. This interrupt capability is invoked primarily by the HSTP's handling transfers to the 819 disk subsystem.

CYBER 176 Central Memory /SCM/ consists of bipolar semiconductor memory ranging in size from 131 K to 262 K words of 60 bit memory. Associated with each 60 bit word in an additional 8 bit SECDED field and logic associated with this field permits the correction of single bit errors and the detection of double bit errors. Memory is interleaved 16 ways in order to minimize the occurrence of bank conflicts. The bank cycle time is 82.5 nanoseconds for read operations and 165 nanoseconds for write operations, and this together with 16 way interleaving provides for maximum transfer rate of up thirty-six 60 bit words per microsecond.

The CYBER 176 Extended Memory /LSCM/ is composed of ferrite core with integrated circuit control logic. The memory size ranges from 524 K to 2097 K. Extended Memory is equipped with SECDED logic. The memory is made up from individual banks of 262 K words with
a bank cycle time of 1.76 microseconds. Associated with each memory bank is a 16 word bank register, and this feature together with bank interleaving permits a maximum transfer rate of one word per clock period on the one or two million word configurations. The Central Processor can also directly reference elements in Extended Memory on a single word basis.

The next element of the CYBER 176 I should like to describe is the Input/Output Multiplexer. The multiplexer provides the interface between High Speed Peripheral Processors and the peripheral processor subsystems on the one hand and Central Memory on the other. The single path to CM can handle data up to a speed of 18 million character per second. The interface to the Peripheral Processor Subsystem is via a 60 bit port and allows each of the ten PPs access any area of Central Memory. The maximum data transfer rate across this 60 bit wide path is also 18 million characters per second. The High Speed Peripheral Processors are each connected to the I/O Multiplexer by 12 bit channels.

A basic CYBER 176 includes four channels, and four HSPPs, for handling the 819 disk subsystem but up to a maximum of 14 such channels can be connected to the I/O Multiplexer. Unlike the Peripheral Processor Subsystem, data transferred across one of these channels is directed to a dedicated circular buffer area in Central Memory. Management of these buffers is accomplished with a central processor interrupt system. As data flows into or out of the buffer, thresholds at midpoint and end-buffer cause a central processor interrupt to occur permitting the processor to remove or provide new data parallel with the continuing data flow across the channel. These channels can operate at data rates up to six million characters per second.

The Peripheral Processor System has the memory cycle time 500 nanoseconds. The other characteristics are the same as for CYBER-73.

The High Speed Peripheral Processors have a 4 K twelve bit memory with a 275 nanoseconds cycle time. The timing in these processors is controlled by 27.5 nanoseconds clock period.

The characteristics of disk subsystem include a very high transfer rate of just over six million characters per second, which is very closely matched to the actual six million char/sec rate of one of HSPP channels. A Peripheral Processor Subsystem channel can only handle a maximum rate of four million char/sec and hence be unable to fully exploit S19 capabilities. The S19 disk drives use a twenty platter non-interchangeable pack that has capacity of 412 million characters.

From the above hardware examples it is obvious, that the various manufacturers use different approaches to achieve the high computer performance. IBM applies the concepts of the High Speed Buffers, paging. CDC uses the multiprocessing with peripheral processors, instruction stacks and pipeline concept.

2. Operating Systems

In this chapter I am going to present an overview of the operating systems, which are most commonly used in the scientific computing centres. As in the first chapter, only IBM and CDC operating systems will be presented. The features of the operating system influence to large extent overall system performance. From user point of view it is also important, how the operating system treats various classes of jobs. In the description of the operating systems I will concentrate on the general organization, utilization of system resources and job scheduling. Two relatively old IBM systems MPT and MVT are presented in order to show the progress in the operating system design. VS2 is an example of virtual storage operating system. VM/370 represents the virtual machine super operating systems. The description of two CDC operating systems SCOPE and NOS should help us to understand the problems connected with the
multiprocessor systems.

2.1. OS/MFT

MFT is the name of a control program working in frame of IBM System/360 Operating System. MFT can control a fixed number of tasks concurrently for the computers with the storage size no less than 128 K byte.

Figure 6 shows the structure of main storage under MFT. An installation may define up to 52 partitions, however only 15 may be problem-program partitions. The boundaries between partitions are established at the system generation time, but the operator may redefine the partition sizes during operation. MFT is a spooling system, which can read up to 3 input streams and produce up to 36 output streams /Figure 7/. Job scheduling consists of assigning jobs to partitions. For this purpose jobs are divided into 15 classes from A to O and each program partition may process up to 3 job classes /Figure 8/. Inside a class, jobs are scheduled according to the job card priority.

Immediately after a job ends in a partition, the initiator program is loaded and it selects a new job according to the above rules. The initiator allocates also the requested data sets and I/O devices. Relocating loader program loads job step relatively to the beginning of the partition. The loader is a nucleus program, which has access to the whole storage. Finally, the job can start execution and at the completion, the initiator places the completed job on the output queue.

The task dispatching algorithm used by MFT is called highest-static-priority-first-served /HSFS/. The priority decreases with the partition number. The highest priorities are assigned to the system tasks residing in the nucleus. The tasks in the partition with high numbers /low priority/ can get access to CPU only during I/O operations of the tasks with low partition numbers. Therefore it is important to schedule I/O bound jobs into lower-numbered partitions. In the reverse situation, CPU bound jobs "screen" I/O bound jobs, which get access to CPU during infrequent I/O processing.

The MFT system nucleus occupies a fixed area in a main storage area at least 34 K bytes and it contains the resident partition of the control program that performs the following control functions:
- Task management routines
- Data management routines
- Job management routines
- The system queue area

The I/O requests are scheduled by the I/O supervisor residing in the nucleus according to first-come-first-served /FCFS/ rule.

Because of the static partition sizes, it is difficult for MFT to avoid the storage fragmentation. Also indirect control of job-dispatching priorities leads usually to low CPU utilization. The design of MFT reflects the state of art in system programming of early sixtieth, when the performance was a secondary objective. Further developments of IBM operating systems cope with the above deficiencies.

2.2. OS/MVT

Another version of IBM control program is OS/MVT, which is devoted to processing variable number of tasks. The processing of Job Steps, by an MVT Control Program is presented on Figure 9. A new feature of MVT is a possibility to execution in parallel several tasks in the frame of one Job Step. The second feature is dynamic memory allocation scheme. Unlike in MFT, the MVT user assigns the parameter /EEXGEN = mmm/ indicating its main storage requirement for each Job Step. MVT supplies only this amount of storage to the Job Step, but not a partition of prespecified size for the entire job.

Figure 10 shows the organization of main storage under MVT. The Link Pack Area is reserved for routines that can be used concurrently by the control program and by any jobs. These include access
method routines, storage management routines and job scheduling in conjunction with MVT initiator program. The Master Scheduler provides the main communication link between the operator and the operating system.

Job Steps, Readers, Writer, Initiators are scheduled to the dynamic region area between Nucleus and Master Scheduler. Jobs are devided into 15 job classes using the same scheme described for MFT. The operator loads an initiator for specified job classes to a region of 52 K bytes. The initiator requests a region for first job step, after selecting the job from its first class with the highest priority. If the appropriate region is not available, the initiator goes into the wait state. After a contiguous block of storage appears in the system, the initiator is loaded into it. The I/O devices are allocated to the job step and a routine in the Link Pack Area attaches the job step as a user task in the allocated region. All programs related to this task are loaded and the execution of the task begins. After completion, the region is released and a new initiator is loaded. Immediately after releasing the I/O devices of the first job step, the initiator can proceed to the second job step if any. Minimum region size can not be less than 52 K, unless some reentrant routines of the initiators are moved to the L.P.A. The maximum number of initiators can not exceed 15, however the degree of multiprogramming is variable and depend on storage requirements of jobs steps in the execution.

Some privileged regions, like teleprocessing jobs, may increase their size during job step execution. In this case they roll-out other jobs, which must come back to the same region to complete the execution. The task dispatcher in the systems nucleus controls the task residing in the storage according to HSFS rule. The priorities are taken from JOB statement or from EXEC for each job step in the last case. It is possible than to assign high priority to the I/O bound job steps and low priority to the CPU bound job steps. Among the tasks with same priority, FCFS rule is used.

The main advantages of MVT in comparison with MFT are: the reduction of storage fragmentation, the elimination of fitting jobs to partition sizes and the elastic dispatching scheme. The last one however assumes the user knowledge and the good will to dispatch job steps correctly. Some options of MVT allow to eliminate this doubtful assumptions.

The first is the time-slice option, which handles a selected dispatching priority according to round-robin scheduling rule. The other dispatching priorities are using the HSFS rule. The round-robin scheduling for one dispatching priority protects at least partly from monopolized of CPU by a CPU bound task and it is helpful to improve response time for short jobs.

The second option is the Houston Automatic Spooling System /HASP/, developed by a group of users. HASP includes the heuristic dispatching option, which is able to monitor some characteristics of job step. A job step is classified as I/O bound, if it will use only a part of time slice. In the opposite case it is classified as the CPU bound. The I/O bound tasks are scheduled first and if all are in wait state, the CPU bound tasks are served in round-robin scheme. The division between task classes in dynamic.

HASP can also establish a desired mix of I/O and CPU bound task by changing the time slice at installation specified intervals. The above features HASP are an example of the efforts toward the full automatization of the operating systems.

2.3. OS/VS2

Further development of IBM operating systems is connected with concept of virtual storage. The system control program OS/VS2 is known in two versions. Release 1 provides one virtual address range. Jobs are assigned regions from this just as jobs are assigned regions in real storage of MVT. Release 2 provides
multiple virtual address space: each user receives its own copy of virtual storage, minus space used for certain system function /e.g. the nucleus/.

Figure 11 shows the virtual storage overview for Release 1. The abbreviation LSQA stands for the local system queue area. Basically the concept of Realise 1 is close to OS/MVT except of course the virtual storage. Therefore we will concentrate on the description of Release 2.

Figure 12 shows the virtual storage lay-out for Release 2. Each address space is divided into system area, user area and the common area. The system area contains the nucleus, which is fixed in real storage then mapped in low addresses of virtual storage. The System Queue Area contains tables and queues relating to the entire system. The Pageable Link Pack Area contains supervisor call routines, access methods and any reentrant read-only system and user programs, that can be shared among users of the system. Common Service Area contains data for communication among the private user address spaces.

On the right part of Figure 12 we see the structure of the user address space in virtual storage. The LSQA contains tables and queues associated with the user's job and address space. Scheduler Work Area contains control blocks and tables created during JCL interpretation and used by the initiator during job step scheduling. Subpools 229 and 230 are used for control blocks, but can be obtained only by authorized programs. The remainder of private address space is available to its user, with space being allocated from low address up.

Figure 13 shows the VS2 Release 2 control program overview. After the system is initialized and the job entry subsystem is active, jobs may be submitted for processing. To schedule a batch job, the job entry subsystem issues a START command for an initiator. To schedule a time sharing job, a user issues a LOGON command. As a result of START and LOGON a new address space is required.

The address creation routine notifies the system resources manager that a new address space is to be created. The system resources manager decides, based on factors like priorities and the number of already existing address spaces, whether or not a new address space is advantageous. If not the new space will not be created until the system resources manager finds conditions suitable. If the new address space is acceptable, the address space creation routine involves virtual storage management /VSM/ to assign virtual storage and set up addressability for the address space.

VSM builds an LSQA and sets up a segment table, page tables and external page tables in it. VSM also creates control blocks to operate the control task for the address space.

Then the region control task /RCT/ receives control. The RCT builds control blocks that further define the address space, then attaches the started task control routine /STC/.

Next, STC uses an initiator as a subroutine to select the job. The initiator passes, the job ID to the job entry subsystem. The job entry subsystem invokes the interpreter to build scheduler control blocks in the scheduler work area /SWA/ for the address space. Upon return from the job entry subsystem, the initiator performs allocation and it issues an ATTACH for the task related to the address space: the terminal monitor program /LOGON/, or any started program /START/. If the START command is for an initiator, the initiator asks the job entry subsystem for a problem program job that is ready for execution. The job entry subsystem calls the interpreter to build the scheduler tables in SWA. When the initiator receives control again, the problem program is attached.

Task management is performed by the supervisor. Basically, the supervisor controls the use of the CPU, real storage, and virtual storage. All supervisor activity begins with an interruption. The supervisor interruption handler saves critical information necessary to return control to the interrupted program after
the interruption is processed. In most cases, the interruption handler passes control to one of the following routines to process the interruption:
- Task supervisor, which performs services /such as attaching and detaching a subtask /requested by tasks and allocates CPU among competing task.
- Contents supervisor, which locates requested programs, fetches them to virtual storage if necessary, and schedules their execution.
- Real storage manager, which directs the movement of pages between real storage and external page storage.
- Auxiliary storage manager which handles external page storage including virtual I/O data sets.
- Virtual storage manager, which services GETMAIN and FREE MAIN by allocating and deallocating storage within the virtual address space.
- Service manager, which improves system response through a new dispatching techniques that allows internal system function to run enabled, unserialized, and in parallel on a multiprocessing system.

In VS2 Release 2 an installation can specify, in measurable terms, the performance that any member of any subset of its users is to receive, under any workload conditions and during any period in the life of a job. The system resources manager is responsible for tracking and controlling the rate at which resources are provided to users in order to meet the installation’s requirements.

The installation defines:
- Performance groups - subsets of users that should be managed in distinguishable ways.
- Performance objectives - distinct rates, called service rates, at which CPU, I/O, and real storage resources are provided to users in a performance group at certain workload level in the system.

Service rate is the number of service units per second a user should receive. A service unit is a measure of processing resources. The system resource manager monitors the rate at which service is supplied to a user in order to ensure that the installation performance specification is met.

To take advantage of the system resources manager, user simply identifies the performance group in which he is to be included, as prescribed by the installation.

Service units are used to measure the amount of processing resources provided to each address space. They are computed as a combination of the three basic processing resources:

\[
\text{Service units} = A \times \text{CPU} + B \times I/O + C \times \text{frames/}
\]

The coefficients A, B, C are supplied by the installation.

When an installation specifies performance objectives, it specifies one or more service rates, how many service units per second a user should receive. The installation is not specifying any particular amount of the individual resources that a user is to receive. It is assumed that different users will use CPU, I/O and real storage in different proportions. However, by supplying the coefficients, the installation can adjust the relative importance of CPU, I/O, or real storage resources within service definition.

The purpose of performance groups is to group user transactions that the installation considers to have similar performance requirements. Basically, a user transaction in a batch environment is a job or job step and in a time-sharing environment, an interaction. The installation can define as many as 255 performance groups, each identified by a distinct performance group number.

Each performance group can be divided into as many as eight periods. By dividing a performance group into periods, an installation can associate different performance objectives with a different periods in the life of transaction. The duration of a period can be specified either as a number of real-time seconds or as a number of accumulated service units.

A performance objective states service rates, how many service units per second an associated transaction should receive under different system workload conditions.
The installation can define as many as 64 performance objectives, each identified by distinct number from 1 to 64.

Figure 14 shows an example of performance objectives.

The system resource manager tracks the service rates provided to users and the average workload level of the system. As the level increases or decreases, it adjusts the service rates to maintain the relationship between the performance objectives that the installation has defined.

User transactions are associated with performance objectives by means of performance group and periods within each performance group; each period of a performance group definition includes a performance objective number. Figure 15 shows, how a user is associated with a performance objective.

To manage system resources, the system resources manager serves as the centralized decision-maker. It monitors a wide of data about the condition of the system, seeking to control such key variables as:

- Amount of real storage allocated.
- Distribution of I/O load.
- Swapping frequency
- Level of multiprogramming
- Paging rate.

By centralizing the control of these variables, the system resource manager can better make decisions that will affect overall system performance.

VS Realise 2 is an example of a fully automated operating system, which can be tuned by large number of installation parameters. Such a tuning process involves also very complex measurements, which will be discussed in one of the following chapters.

2.4. VM/370

Virtual Machine Facility/370 is a system control program, that manages a real computing system so that all its resources are available to many users at the same time. Each user has at his disposal the functional equivalent of a real dedicated computing system.

While the control program of VM/370 manages the concurrent execution of virtual machines, an operating system must manage the work flow within each virtual machine. Because each virtual machine executes independently of other machines, each one may use a different operating system, or different releases of the same operating system.

The following operating systems can execute in virtual machines:

- DOS, DOS/VS, OS/PC?, OS/MFT, OS/MVT,
- OS/VS1, OS/VS2, OS-ASP, PS44, CMS, RSCS.

Figure 16 shows an example of the Multiple Virtual Machine. A virtual machine consists of the following components:

- Virtual system console
- Virtual storage
- Virtual CPU
- Virtual channels and I/O devices

As a virtual system control serves usually IBM 2741 Communication Terminal or an IBM 3277 Display Station. By entering commands at his terminal a user can perform almost all the functions an operator can perform on a real machine system console. He can load an operating system, stop and start virtual machine execution, and display and change the contents of registers and storage.

Each virtual machine has its own virtual storage space from 8 K to 16 million bytes. Control Program brings into real storage what ever part of virtual storage is needed for the virtual machine’s execution, but does not necessarily keep in storage those parts that are not needed immediately.

Control Program provides CPU resources to each active virtual machine through time slicing. The virtual CPU can execute in either basic or extended control mode. For example OS/MVT and OS/VS2 can execute in virtual machines.

A virtual machine supports the same devices as a real machine, Virtual devices are logically controlled by the virtual machine and not by VM/370. In most cases input/output operations, and any error
recovery processing, are complete responsibility of the virtual machine operating system.

Virtual and real device addresses may differ. CP converts virtual channel and device addresses to real channel and device equivalents and performs any data translation that are necessary.

All virtual devices must have real counter parts. A virtual disk must have a real disk counterpart, or a virtual tape must have a real tape counterpart. Some virtual devices, such as tapes, must have a one-to-one relationship with a real device. Others may be assigned a portion of a real device. For example, a virtual disk may occupy all or part of a real disk. In other words a real disk can be devided into several virtual minidisks.

Two operating system CMS and RSCS are considered as the part of VM/370. The Conversational Monitor System /CMS/ provides users with a wide range of conversational, time-sharing functions. The Remote Spooling Communication Subsystem allows users to transmit files to remote stations in the RSCS teleprocessing network.

The main advantages of VM/370 are connected with the operating system development, testing of complex system and interactive program development. Programs developed in a virtual machine can exceed the real storage size of the computer. Programmers can use debugging aids at their terminal, which are normally reserved for the computer operator. They can display and store into registers, stop execution at an instruction address and after normal flow of execution. CMS simplifies the creation and manipulation of source programs on disk, and allows the user to examine selected parts of program listings and storage dumps at this terminal.

The disadvantages of VM/370 are related with low execution speed of batch oriented operating systems. For example a virtual machine with OS/MVT might execute at only one half of its speed on real IBM 370.

2.5. SCOPE 3.4

From complicated organization of CD 6000 and CYBER computers, one can easily see, that it is a challenging task to write an effective operating system, which could fully exploit the multiprocessing capability of these computers. In the past many mistakes were done in the design of early versions of such operating systems.

The Supervisory Control Program Execution /SCOPE/ is the name of an operating system, which is now together with its successor NOS/BK most often used on CD 6000 and CYBER computers. SCOPE 3.4 and NOS/DE have basically the same organization. SCOPE has been primarily designed for the batch processing, but later on an interactive subsystem INTERCOM has been added.

Job and interactive command execution is controlled by SCOPE peripheral monitor /PPMR/ and central processor monitor /CPMR/. The activities of the monitors are peripheral supported by more than 300 routines. SCOPE uses some area of the central memory to store: the system tables and CPMR programs /CMR area/, the central memory resident library, and also the block addresses for files on mass storage /MBTA area/.

Figure 17 shows the usage of the central memory /CM/ for system and users. The user job in execution is assigned a contiguous area in CM and a control point number. A control point is a concept used to facilitate book-keeping. SCOPE permits up to 15 control points. CPMR programs do not run on the control points and only one control point is taken for the spooling subsystem JANUS. The others are used for user or operator jobs.

The communication between various processes running in SCOPE is rather complicated. When a user program is loaded and executed as the result of a control card call, the system must place any parameters specified on job card within the field length of the job. No central processor instruction allow a CP program to perform I/O, therefore a request must be send to the system, to load a PP routine.
to execute I/O. The request is placed in a register located at the reference address plus one /RA+1/, as it is shown on Figure 18. CPMTR will pick up the request inserting the control point number. Therefore the user's program should use the exchange jump instruction immediately after placing a call in RA+1. This will cause CPMTR to begin execution immediately. If CPMTR determines that the RA+1 call should be assigned to a PP, it will pass the call to PPMTR.

When a PP is available, PPMTR will write the call into its PP input register, in CMR. The PP resident is checking permanently its input register and when it sees the call, the appropriate routine will be loaded and executed.

SCOPE peripheral routines can not load and often execute without help of the monitors. In fact the monitors will be asked to perform such functions like: channel reservation, loading of peripheral, transient programs and overlays, sending of dayfile messages, changing of control point assignment and requesting another peripheral job.

When PP resident has a monitor request, it places a message into PP output register in the PP communication area. After making the request, PP resident waits for the first byte of the byte of the output register to be set to zero, signalling that the monitor has processed to request.

The PPMTR is in general control of the system. It is loaded into PPO at deadstart and remains there for the duration of system execution. Primarily, PPMTR controls and coordinates system activities to avoid conflicts between various system processors. It allocates peripheral processors, central memory to control points. During the execution of its main loop, PPMTR scans the CPMTR request stack /T.MTRRS/ for a CPMTR function call or a PP program request. Moreover PPMTR looks for the pending function coming from peripheral processors, it advances the system clock, and it checks RA+1 address of executing job. PPMTR upon completion of the four high speed functions will process one of slower functions. The list of slower function includes: advancing of control point, scanning of PP output register, processing of delay stack and some others.

CPMTR processes certain PP output register monitor functions, it checks user program RA+1 requests and it schedules CPU.

The CPU dispatching is based in earlier versions of SCOPE on one level round-robin rule. Recently a multilevel round-robin rule has been introduced. Active jobs from one class form a ring, which is served by CPU until all jobs disappear from the ring. Then a lower priority ring is served. The value of time slice depends on job priority and field length.

Input and output request processing depends upon the source of each request. Active user CM programs issue RA+1 request for I/O which are cycled through CPMTR. PP programs request I/O by placing a monitor request into their PP output register. CPMTR assigns the I/O request to CPCIO which in turn assigns it to the proper peripheral processor, CIO or ISP /see Figure 19/. CIO /ISP /processes requests for magnetic tape, teletype, a unit record equipment. ISP /stack processor/ processes all requests for mass storage I/O.

Before calling CIO, the program must set up circular buffer parameters and the CIO operation code in the file environment table /FET/ . The relative address of the FET is placed in the CIO call. Then the file is opened. CPCIO determines if the file is on allocatable or nonallocatable device.

If the file is in on allocatable device, CPCIO calls Stack Processor Manager /SPM/ to enter the request in the I/O Request Stack. SPM performs request scheduling and device optimization. Rotational Mass Storage I/O is performed by SPM selecting a stack request and assigning it to a ISP driver. The request is placed into ISP communication area. ISP comes up, it initializes the driver overlay appropriate to the specified RMS device /3SW/ for 841 disk or 3SY for 844 disk/. ISP performs the I/O requested, obtaining field access
as necessary and at I/O completion returns the request to SPM for termination processing.

If the file device code is for a non-allocatable device, CIO and its overlays will process the request. For example, if a user issues a request to read data from a file on a SCOPE standard format 7-track tape, CIO will call the overlay INT into its PP. INT will reserve one of the hardware channels connected to the equipment. It then issues the function code to connect the controller and tape driver. INT issues functions to transmit one PRU of data from the tape driver over the data channel.

When the entire PRU is transmitted or an end-of-record is encountered, INT picks up pointers to the circular buffer from FET and it transfers data from PP to the buffer. INT updates the PRU count in the file name table/FNT/, releases the channel, sets completion bits in the FNT and FET, and drops out.

The above description of SCOPE 3.4 I/O corresponds to level 430. Earlier versions of SCOPE 3.4 were using ISP driver for device optimization, instead of SPM. The device optimization algorithm schedules in both cases the requests for which the disk head displacements and the rotational latency is minimal.

SCOPE processes jobs in the system in the three independent stages: Input, Scheduling, Output. Jobs can be loaded into computer reading card decks into the system using the system package Janus. Alternatively, they may be input from tape with the tape loader /1LT/ or from a user terminal through INTERCOM /see Figure 20/.

As each job is read into computer, it is placed into input/preinput file. The preinput queue is formed for the jobs with magnetic tape requests. This jobs will be individually staged to execution by the operator.

Whenever one control point and 2000 B words of Central Memory are available, the Scheduler calls INT to initiate another batch job. INT scans the input queue and it calculates the input queue priority according to job card priority and job age.

Many installations, including INR, change the algorithm for the input queue priority to include the job field length and the time limit instead of card priority. A job with the highest input queue priority is assigned to control point and it starts the execution.

After initiation the job is under control of the Scheduler, which is a CPI/RR program running in user mode. The Scheduler is responsible for allocating control points and central memory.

The Central Memory queue /CMQ/ contains all jobs waiting to be run at a control point. If a job is in the CMQ, it must have all the resources that it currently needs to run except for Central Memory and control point.

The Device Queue is formed by jobs requesting a non-allocatable device. The program ITS is responsible for detecting, when the appropriate device is ready, and for calling the Device Queue manager /1DM/. 1DM will in turn call the Scheduler to put the job descriptor back into the CMQ.

The Permanent File Queue consists of all jobs waiting to attach a permanent file. If a job at a control point tries to attach a permanent file, the PP routine PFA is called. If PFA determines that the job cannot attach the permanent file because it is temporarily unavailable, PFA calls the permanent file queue manager /1PF/. Whenever a permanent file is detached, 1PF checks if there are jobs waiting for the file. If there are, 1PF will select the job which has been waiting the longest. 1PF will then call Scheduler to put the job into Central Memory queue.

Jobs which are waiting for operator action will be in the Operator Action queue. Then the operator enters an appropriate type-in the job will be put into the CMQ and eventually it will be rolled in and initiated at the control point.

Interactive INTERCOM jobs which are waiting for input from a terminal or waiting until output can be sent to a terminal are swapped out and put into the INTERCOM queue. Then the terminal I/O
completes, 1C1 will request the Scheduler to place the job in CMk.

The function of the CP Scheduler program is to decide which jobs should be run at any given time and to use CM efficiently. The Scheduler decides which jobs to swap-in and it then calls the swapper PP program to perform the actual swapping. Each job in the CMk or that is executing at control point has an associated priority called "queue priority".

A job card priority /JCP/ has a weighting effect on the queue priority. The Scheduler makes its decision based entirely on queue priority.

It will schedule in the highest queue priority job in the CMk, which will fit in the currently available memory and in the memory assigned to jobs of lower queue priority.

When a job is swapped into a control point, it is given a high queue priority. At the end of a period of time called a "quantum", the queue priority is changed to a lower value thereby making the job a more likely candidate for swap-out.

The quantum for a job is considered elapsed when \((X+Y/4) \cdot 64 \geq Bq\), where \(X\) denotes amount CPU time, \(Y\) denotes PPU time and \(Bq\) stands for quantum value.

Jobs in the system are divided into five classes:
1. Batch jobs using no non-allocatable devices.
2. Batch jobs using one or more non-allocatable devices.
3. INTERCOM /interactive/ jobs.
5. Express jobs.

A multiuser job is a program that runs under control of INTERCOM and that simultaneously processes several terminals in a serial manner /EDITOR/. An express job is a job for which the operator entered BSTOP, KILL or RERUN.

For each class there is an entry in a table in CM called the Job Control Area which contains parameters for scheduling of jobs in the class. These parameters include:
1. Minimum queue priority /MINQP/
2. Maximum queue priority /MAXQP/

3. Aging rate /AR/
4. Quantum priority /QP/ at control point.
5. Quantum length /Bq/

When a job has been swapped out and enters the CMk, it is assigned a queue priority equal to its base priority /BP/. The base priority is normally a combination of minimum queue priority for the class and the cob card priority /BP=MINQP+8 \cdot JCP/. The priority is incremented with time at a rate equal to the aging rate of the class. When the priority of a job in the CMk reaches the maximum priority of the class, its priority is no longer aged.

When a job is swapped into a control point, it is given a priority equal to the quantum priority plus eight times the job card priority. When the quantum for the job has elapsed, the priority of the job is set to the base priority.

The following tables gives the CDC standard set of Scheduler parameters:

<table>
<thead>
<tr>
<th>Class of Jobs</th>
<th>MINQP</th>
<th>MAXQP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>Dependent</td>
<td>200</td>
<td>1000</td>
</tr>
<tr>
<td>Interactive</td>
<td>1100</td>
<td>2400</td>
</tr>
<tr>
<td>Multiuser</td>
<td>2410</td>
<td>2510</td>
</tr>
<tr>
<td>Express</td>
<td>1000</td>
<td>3200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class of Jobs</th>
<th>AR</th>
<th>QP</th>
<th>3Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch</td>
<td>4</td>
<td>1400</td>
<td>200</td>
</tr>
<tr>
<td>Dependent</td>
<td>10</td>
<td>1400</td>
<td>200</td>
</tr>
<tr>
<td>Interactive</td>
<td>1000</td>
<td>2500</td>
<td>200</td>
</tr>
<tr>
<td>Multiuser</td>
<td>200</td>
<td>3000</td>
<td>400</td>
</tr>
<tr>
<td>Express</td>
<td>400</td>
<td>3200</td>
<td>400</td>
</tr>
</tbody>
</table>

For the above parameter set, an interactive job needs only 0.6 sec to swap-out a batch job and a multiuser job will swap-out a batch immediately. Therefore some installation like Washington University and INR use the different parameter sets, which reduce amount of swapping caused by interactive jobs.

SCOPE, like others modern operating system, need very carefull tuning of large number of system parameters in order to achieve high performance. The tuning process may be performed only by means of the system performance monitors, which will be described in one of the following chapters.
KRONOS is another operating system developed by CDC for the same range of computers as SCOPE. KRONOS was designed to provide time-sharing and transaction capabilities for large number of interactive terminals. At certain stage of development, the name of the operating system has been changed into NETWORK OPERATING SYSTEM /NOS/, which basically has the same organization. The new features of NOS are related with the communication processor CD 2550, which can serve as the node of a complicated computer network.

NOS uses many concepts of SCOPE like: peripheral and central monitor, control points, PP communication area, RA+1 requests and recall. The recall program status is provided in both systems to enable efficient use of the central processor in the multiprogramming environment. Often, a CP program must wait for an I/O operation to be completed before more computation can be performed. To eliminate the CPU time wasted if the CP program were placed in a loop to await I/O completion, a CP program can ask the monitor to put the control point into recall status until a later time. Then the CPU may be assigned to execute a program at some other control point. Recall may be automatic or periodic. Auto-recall should be used when a program requests I/O or other system action and cannot proceed until the request is completed. The monitor will not return control until the specific request has been satisfied. Periodic recall can be used, when the program is waiting for any one of several requests to be completed. The program will be activated periodically, so that it can determine which request has been satisfied and whether or not it can proceed.

The main differences, between NOS and SCOPE, are connected with the modular structure, organization of I/O and simplicity of NOS system tables.

Figure 21 shows the residency of the NOS operating system. The subsystems run on the control point like user jobs. Up to 23 control points can exists in NOS. Each subsystem is in charge of the user interface, the submission of jobs to the queues, and initiation of system commands to the monitor. An exception is MAGNET subsystem, which handles the automatic tape assignment.

TELEX subsystem handles time-sharing and differed batch /coming from interactive terminals/. TELEX passes the transaction messages to TRANEX. The transaction subsystem TRANEX is one in which messages received from terminals trigger the execution of one or more tasks to interact with data file or data base. EXPORT/IMPORT subsystem processes the remote batch from 200 USER terminals. The equivalent of JANUS, which handles the local batch, is called BATCHIO subsystem.

The main advantage of modular subsystem is the release all CM space taken by the subsystem tables in the case of the operator drop. In other words CMR for NOS has less permanent system tables then for SCOPE.

The second difference between NOS and SCOPE is connected with the decentralized disk I/O processing by NOS. At first glance, the SCOPE centralized disk I/O processing looks very attractive. It gives the possibility to choose from the stack the request with the shortest execution time and therefore to minimize the disk head movements. However measurements show, that the stack processor preparations for the request processing take a lot of time. Therefore it is better to have the parallel preparation of many requests by independent processors. This is the case with NOS, where each pool peripheral processor has a disk driver included in the PP resident. It gives the possibility of the channel reservation only if the request is ready. Otherwise, another PP can reserve disk channel and it performs the disk I/O.

Figure 22 shows an example of the interactive job flow using NOS decentralized disk I/O feature.

Job scheduling in the NOS operating system is performed similarly as in SCOPE 3.4. An example of the Job Control Parameters is presented on the Figure 23.

NOS provides very powerful control
language, which allows the programmer to transfer control and to perform arithmetic and test function within the control statement record. Control language of statements similar to FORTRAN statements /GOTO, SET, CALL, IF, DISPLAY/. An important feature of control language is capability to create procedure files. A procedure file is a group of control language statements which can be called much like a subroutine for insertion anywhere within the control statement record.

Another feature, important for interactive users, is the existence of indirect permanent files. Indirect permanent files are accessed by using a working copy of the permanent file as a local file attached to the user’s job. If the user wishes the working copy to remain permanent after the file has been altered, the SAVE or REPLACE functions must be issued.

In NOS there is not possible an overflow on disk units. Also it is impossible to create dependency between jobs and to access Extended Core Storage by the user. In spite this deficiencies, NOS is considered as extremely effective operating system for the installations with very large number of interactive terminals.

3. Performance Definitions

The computer system performance may be seen from different points of view. Since the process of computer evaluation involves computer system designers, managers, system programmers and application programmers, it is obvious, that "performance" may be a highly subjective term. Therefore it is defined sometimes as "performance is the degree to which computing system meets the expectations of the person involved with it".

In more popular term performance is understood as measures of system speed and resource use. From the external point of view, it is important “effectiveness” with which the system handles the specific application. From the internal point of view, it is important "efficiency" of the resource utilization in order to process to workload.

The workload is the most difficult factor in the computer performance evaluation. In the production environment, the workload depends very much on day time, day of the week, user customs and work schedule. Using powerful measurement tools it is possible to compare the effects caused by the changes in an operating system during the production time. The measurement tools help us to exclude the workload which is not "typical" for certain time of the day and to introduce only typical workload periods for statistical analysis. However, the production measurements can be only applied to establish rather significant change in the computer performance.

Therefore installations and manufacturers use benchmarks and stimulators to achieve repeatable results. It is of course a problem how for benchmark and stimulator can imitate the "real" workload. The problem is too large extent solved by appropriate dayfile analysers. The dayfile analysers collect such workload characteristics like:
1. Job CPU time
2. Job channel time
3. Number of I/O request
4. Average or maximum memory size requested by a job
5. Job PPU time
6. Priorities assigned to a job.

Some other workload characteristic are more difficult to get: system request interarrival time, blocked time, working set size and locality of reference.

Figure 24 shows the result of the measurements performed using the program \$KLOAD by D.Makosa at INR. \$KLOAD is devoted mostly the measurements of interactive job characteristics and it requires certain changes in SCOPE to obtain additional dayfile messages. The characteristics, obtained by means of \$WORKLOAD, may be used in the stimulation technique.

The stimulators were developed for
SCOPE and NOS operating systems by N. Williams form CDC. The SCOPE STIMULATOR is a set of CP and PP programs that simulate the 6766 multiplexer and TTY terminals. Under normal condition, when the hardware is present, the INTERCOM driver resides in a PPU and communicates with the multiplexer over a data channel. The multiplexer in turn transmits the data to the appropriate terminals. The most basic function of the STIMULATOR is to communicate with the INTERCOM driver over a common data channel. When the STIMULATOR is present, the STIMULATOR routines take the place of the multiplexer hardware. The STIMULATOR processes the driver's function codes as well as receiving and transmitting data destined for each terminal. STIMULATOR run consists of two phases. The first is the stimulation phase where all data received from INTERCOM and calculated response times to each command are written to tape. The second phase consists of data reduction and report generation.

The resources required for stimulation are one control point, one or two dedicated PP's, depending on whether or not the option to save the output to tape is requested, one dedicated data channel, one or two tapes and central memory whose size depends on the number of terminals being simulated and the length of the input tests. The stimulation phase is performed by two PP routines, 1VG and VSM and one CP routine, SIP \( \text{see Figure 25} /\).

The selection of a benchmark is not considered usually as a difficult problem. However it is necessary to include in the benchmark not only the most frequent jobs but also long-run jobs, which take the most of system time. It is also popular to use synthetic benchmarks, which simulate usage system resources without performing the function of normal jobs. By means of control parameters a synthetic job can change its CPU, central memory and I/O requirements and therefore the whole benchmark can be easily constructed.

With the well defined workload we can proceed to further steps in the performance evaluation. The steps will involve the definition of performance measures, the determination of the quantitative values of performance measures and the discussion of the results. Only the first step will be covered in this chapter and the other will be discussed in the following chapters.

Now I would like to remind you some definitions of performance measures which describe the system effectiveness:

1. **THROUGHPUT** is inverse of elapsed time required to process given workload or the number of user jobs processed per unit of time.\( /\)
2. **TURNAROUND TIME** is the time period between submitting a job and receiving the output.
3. **RESPONSE TIME** is the tournaround time for interactive command.

Another set of performance measures describe the system efficiency:

4. **CPU UTILIZATION** is a percent of time a CPU is working for system and users.
5. **UNIT OVERLAP** is a percent of time during which to computer units operate simultaneously \( /\)for example CPU and a channel.\( /\)
6. **EXTERNAL DELAY FACTOR** is the ratio between multi- and monoprogramming turnaround \( /\).
7. **SYSTEM UTILITY** is the weighted sum of all system units \( /\)usually weights are proportional to prices of units.\( /\)
8. **REQUEST WAIT TIME** is a time required to process a request \( /\)CPU or I/O.\( /\)
9. **PAGE FAULT FREQUENCY** is the number of page faults per second.

As you see from the above definitions some performance measures are of the general nature and some other like PAGE FAULT FREQUENCY are applicable only two special operating systems.
4. Measurements Tools

The software and hardware resource utilization is measured by means of software, hardware and hybrid performance monitors. As a software monitor it is understood a special program incorporated into the operating system. Depending on computer and monitor design we can distinguish between Central Software Monitor /CSM/ and Peripheral Software Monitor /PSM/. CSM runs in the same processor as user programs and therefore it is necessary to interrupt the processor activity in order to take measurements. The interrupts may occur regularly at specified time intervals /sampling technique/ or may appear as the result of an event /event driven technique/. PSM is permanently or temporarily residing in the peripheral processor and its activity does not cause additional interrupts of the central processor. PSM may however slightly influence the performance of the operating system due to I/O requests and conflicts on memory banks.

The hardware monitors consist usually of probes, logic box, counters and recording devices /see Figure 26/. The probes are sensing the electronic signals in the circuitry of the measured systems, indicating the resource states being monitored. The logic box allows to perform logic functions on the signals /like AND, OR, NOT/. The results are accumulated in the counters, which content can be displayed or recorded on magnetic tapes.

The hybrid monitor combined the features of hardware monitor with the abilities to analyze the software resources, like files, tables, etc.

In this chapter I am going to describe the example of the CSM, PSM and hardware monitor.

4.1. System Activity Measurement Facility for VSE

The System Activity Measurement Facility /SMF/ is a CSM using the sampling technique for VSE operating system. SMF permits the gathering of information on the following classes of system activity:

1. CPU activity
2. Channel activity and channel - CPU overlap
3. I/O device activity and contention for:
   - Unit record devices
   - Graphics devices
   - Direct access storage devices
   - Communication equipment
   - Magnetic tape devices
   - Character reader devices
4. Paging activity
5. Workload activity.

SMF is limited to reporting on system activity as that activity is communicated to the system /for example, by setting of flags/. As a result of this indirect reporting, statistically sampled values can approach in accuracy only the internal system indications, not necessarily the external activity itself. For example, if a CPU is disabled so that the freeing of a device /device end interruption/ cannot be communicated to the system, the device will appear busy for a longer period of time than it would if it were measured by hardware monitor.

The sampling activity is used to collect only a part of measurements. For example, the percentage of channel busy time is derived by dividing the number of sampling observations during the reporting interval in which the channel was busy, by the total number of observations. However the channel activity count is taken from an operating system counter and it is independent of sampling cycle, which can change from 50 to 999 milliseconds. The channel activity count gives the number of successful Start I/O, which were issued to the channel during the reporting period.

The system operator initiates SMF monitoring with the START command. SMF can also be started as a batch job. The report formatting takes place at the time specified by the parameter INTERVAL.

Figure 27 shows the Paging Activity Report of SMF. This report provides detailed information about demands on
the system paging facilities and the utilization of real and auxiliary storage during the reporting interval.

Explanations of some fields appearing in the report are as follows:
- Pageable System Area include the pageable link pack area and its directory, and the common storage area.
- PAGE Reclaim Rate is the per-second rate of pages frames that are disconnected /stolen/ from an address space or the system pageable area, but are retrieved for reuse before being re-allocated.
- Swap Page-in Rate is the per-second rate of pages read into real storage, as a result of address space swap-ins.
- VIO Page-in is the transfer of a VIO file page from auxiliary to real storage, resulting from a page fault or a PGLOAD on a VIO window.
- Swaps are the number of address space swap sequences, where a swap sequence consists of an address space swap-out and swap-in.

The System Activity Measurement Facilities can produce also SMF records, which contains the results of measurements. From SMF records it is relatively easy to produce another types of reports, e.g. the utilization of the resource as a function of time.

4.2 SCOPE Monitors CIA and GSS

In this section I am going to describe two PSM, which were designed for SCOPE 3.4 performance measurements. The first version of CIA has been developed by P.Jalics at Stuttgart University in 1973. The Stuttgart version of CIA was applicable only to a particular CD 6600 configuration. In 1974 the author of this lecture introduced a CERN version of CIA which is able to measure the performance all CD 6000 and CYBER computers except of CD 7600.

Figure 28 shows the organization of CIA modules. The main part of CIA monitor is a peripheral program CIA which collects all necessary information from the SCOPE system tables or CYBER channels. The information is placed into tables contained in the body of CM resident peripheral program GOG. At the end of measurements, the statistical tables of GOG are rewritten by a peripheral program GOI into the field length of Fortran program CIAIN. With the help of text file CIATXT, CIAIN produces a final report in the form of distribution tables.

CIA can be initiated by the operator peripheral call, which specifies also a sampling delay in seconds. Delay zero forces CIA to take permanent measurements, what roughly corresponds to 20 samplings per second. Otherwise CIA bounce into a PP every specified number of seconds. Internally peripheral program CIA is divided into the following experiments:
A1. Measures time it takes to read hundred 60 bit words from CM to a PP /indicates PP-read saturation level/
A2. Measures time it takes to write hundred 60 bit words from a PP to CM /indicates PP-write saturation level/
AA−AB. Measure status of CPUA and CPUB /RESCHEDULE, CPMT, IDLE, STORAGE, MOVE, SCHEDULER, USER/
BB. Measures number of free PP in the system
CC. Measures control point status /CPU−WAIT, X−RECALL, A−COMPUTE,
B−COMPUTE, Y−RECALL, M−STORMOV, NEXT/
DD. Measures the amount of free CM
EE. Measures the amount of free ECS
FM. Measures number of seven track tape drives free
FN. Measures number of nine track drives free
HH. Measures a distribution of individual user CM field lengths
II. Measures a distribution of individual user ECS field lengths
J1. Measures the number of entries in the input queue
J2. Measures the number of entries in the output queue
J3. Measures the number of local files in the system
K1−K9. Measures the number of stack request for a DST entry
NX. Measures the utilization of two character or one character classes of PP routines
PX. Measures use of PP routines according to three character name
QA-QX. Measure the CPU idle and channel active overlap
RA-RX. Measure the channel activity.

Figure 29 shows a typical results for Experiment K2, which measures the number of stack requests attached to first physical Device Status Table. For each possible number of stack requests, the experiment gives the corresponding rate of occurrence. At the bottom of the table, the mean request stack length is given.

Groningen System Statistics /GSS/ is a bouncing peripheral processor program with the sampling period of 20 seconds. GSS samples about 250 data pertinent to the status of the system. However data are not recorded in a CM table, like in CIA, but on a permanent file /see Figure 30/.

The permanent file is processed afterward by a central processor program giving daily or cumulative system statistics. The analysis of the data collected on a file is more flexible, than this one applied to CIA histograms. It is possible in GSS to calculate not only the mean values of the measured variables but also the dispersions. The measurements are presented often as a function to day time and some of them are specified for the job classes. GSS measures not only the outstanding stack requests but also the total number of processed stack requests, which was for earlier versions of SCOPE 3.4 a difficult task.

Figure 31 shows the result of GSS experiment for control point occupation.

Apart from complex monitors like CIA and GSS a lot of smaller programs is used for the performance measurements of SCOPE like KIVIAT, STATS, PPSTAT and REPORT. KIVIAT /University of Stuttgart/ consists of the measurement code that is included into PP Monitor, the table in CM and KIVIAT program, that processes the table to produce the results. It runs automatically every hour gathering data on disk permanent file for later statistical analysis. The information about CPU utilization, and channel activity is produced in the form of the clear diagrams.

PPSTAT /INR, H. Wojciechowicz/ consists of measurement code included into CPMTR which counts the number of each PP program-call. This information is analyzed later by PPSTAT and the frequency of call and PP program residency are printed. It enables to define the proper residency of the PP programs.

REPORT /INR, J. Dzieciakosz/ scans the dayfile and selects the desired information and immediately prints out the short report of the computer workload status. This report is divided into three parts:

a/ Important events in system activity /time and type of deadstart and interactive subsystem work/

b/ Status of users and system jobs /time of execution, central processor and channels, time of waiting in input queue, central memory used/

c/ Diagram of central processor idle time /sent to dayfile every five minutes/.

As one can see from the above examples, using appropriate measurement tool it is possible to get very detailed information about the performance of SCOPE 3.4 operating system. In fact this not the case with KRONOS /NOs/ for which practically there are no software monitors. Therefore in next section, I am going to describe a hardware monitor for KRONOS measurements.

4.3. Hardware Monitor of KRONOS System

A hardware monitor was used for the study of CDC-6600 computer running at Federal Computer Performance Evaluation and Simulation Center by dr. David S. Lindsay.

The computer is equipped with 500 K of Extended Core Storage /ECS/, twenty Peripheral Processor Units and twenty-four I/O channels /see Figure 32/. The transfer between 131K Central Memory and ECS is possible at rate ten word per microsecond. The four I/O channels are used to drive twenty 544 disk drives. The other two channels are used for eight 659 tape drives and ten 657 tape drives.

The measurement tool was the COMRESS DYNAPROBE 7900 hardware monitor, which
sensors were attached to points on the back panels of the central memory, central processor and channels. With the help of differential amplifier, the sensors are able to detect the fluctuations which correspond to changes in the status of computer components as busy/not busy. DYNAPROBE can collect data with sixteen counters capable of counting 10 pulses per microsecond, and with twelve bits used to sample, at selected rate, twelve signals. The sample interval was chosen to be 10 seconds for this measurement. Also at intervals of 10 seconds, the sixteen counters and twelve bits are written to a 9-track self-contained magnetic tape. The data tape is then evaluated by software reduction program called DYNAPAR, which reads the data recorded on the DYNAPROBE tape and produces analysis reports. The analyser provides 16 pseudo counters for the arithmetic operations on the binary counters. The analyzer also provides the ability to define combinations of the 12 bits as additional counters.

From the description of the KRONOS scheduler it is obvious that many interactive and batch users can be rollout very often from CM. The rollout can be made to ECS or in the case of overflow to disk.

Since DYNAPROBE is not a hybrid monitor, it was necessary to make changes in KRONOS to measure ECS and disk rollout. Code was added to PPU program 180, the rollout processor, to activate an otherwise unused I/O channel during ECS rollout, so that the hardware monitor could detect such rollouts. The same technique was used on another normally unused channel to report disk rollouts.

Using DYNAPROBE the following measurements were performed:
1. CPU utilization
2. Activity on the four disk and two tape channels
3. Number of ECS and Disk Rollouts
4. Number of Exchange-Jumps
5. Utilization of Central Memory hopper, through which all accesses to CM are performed.

Another equipment namely Rand Monitor /Stimulus Generator /RMS/ was used to measure CD 6600 response times to time-sharing terminal interactions. At predetermined times, the RMS issued a teletype message to a user program executing as a time-sharing job on CD 6600. The program was designed and written to consume specified amounts of CPU time and perform specified numbers of disk I/O operations. Thus the RMS was able to time large numbers of interactions requiring varying amounts of system resources.

4.4. Hybrid Monitor for CD 7600

In 1972 with the purchase of the CD 7600, CERN has requested a nonstandard First Level Instrumentation Peripheral Processor /FLIPP/. FLIPP was designed in close collaboration between CERN and CDC in order to provide the performance monitoring in the production environment.

FLIPP uses a normal CD 7600 PPU with 4K byte core memory. One I/O Multiplexer channel is connected to the CPU for ordinary I/O operations, one specialized channel is also connected to the CPU and furthermore there is one channel to each of the other PPU's /see Figure 33/. As an absolute minimum a system monitor must of course have access to all system tables. Therefore, due to the restrictions to the SCM accesses for a normal 7600 PPU, a special channel was connected to FLIPP giving it direct read/write access to all of SCM.

The access to operating system tables in LCM is solved in two steps. Using the SCM write feature FLIPP puts a request for SCM resident tables into a fixed area in SCM. It then sends an interrupt to the CPU. The interrupt will be taken care of by special FLIPP interrupt handler, which in this case will store the requested LCM tables into another fixed area of LCM and then notify FLIPP that the interrupt has been honoured. The FLIPP is able to obtain the information using its SCM read capability. Continuously interrupting the CPU in order to scan a system table in LCM
may affect the system to an appreciable degree. However, in most cases there is no real need to scan the tables with such high rate. Also the interrupts sent from FLIPP have the lowest priority of all CPU interrupts.

The System Status Interface Consists of special hardware which is loaded when some triggering condition arises. The information is held in one of the three 60-bit diassembly registers /ranks A-C/. There 60 bits are broken down into 12-bit bytes as shown on Figure 34.

The information is gated into rank C register of the monitor hardware by some triggering condition. From here it passes one clock period later, to rank B and then to the rank A as these become free. From rank A it can be read by FLIPP. If a trigger occurs when rank C contains data then the new data will be lost. These ranks are used to buffer the data which can come at a peak rate which is faster than a PPU channel can handle.

The triggering conditions which can cause FLIPP to have access to the system status interface data are:

1. FLIPP generates a trigger itself in order to sample system status.
2. A manual switch on a the console it set to cause a stream of triggers at some fixed clock rate.
3. An external signals occurs at some part of the machine which has been selected as a source of triggers. The signal is detected via hardware probe.
4. A keypoint code is sent from selected PPU.
5. A SCM-LCM block transfer is starting or stopping.
6. A key point code is received from CPU.
7. A CPU exchange jump takes place.

Each PPU is connected to the system status interface via ordinary channel. Using a normal output instruction the PPU may then send a key point code bit pattern /6 bits/ indicating what it is doing.

For the CPU, a modified form of the no-operation instruction is used. When executed the three bit operand field will be sent to the system interface. This makes possible only 8 different key point codes, which of course is far from sufficient for all the CPU code of the operating system. However with the additional job identification code, it is possible to distinguish the task, which executed a no-operation instruction.

Peripheral equipment connected to FLIPP consists of console, display and disk drive. To be able to control and utilize this equipment as well as the measurement hardware, an operating system called FLOSS/ was developed. The resident part of FLOSS contains keyboard, display and disk drivers. All other code, and in particular measurement routines, are loaded upon demand from the disk. The console is used for initiating and controlling the measurements runs. Their progress and the results obtained may be presented on the display. The disk is used both for storing measurement data and for keeping the non-resident FLIPP software. The recorded data may then be transmitted over the file transfer link to an analysis program in the CPU. For long term storage the data may be routed to the 6000 front-ends and then written onto magnetic tape.

5. Operating System Tuning

In general, tuning a system is the process of modifying its hardware or software characteristics to bring performance closer to installation defined objectives. Usually the operating systems contain a lot of parameters, which can influence system performance. The manufacturers supply the default values for the parameters, however it is impossible to provide an optimal parameter set for wide range of computer models and various possible workloads. A part of the parameter choice, the installations usually introduce many modifications to the operating systems, which can influence to large extent the performance.

In this chapter I am going to discuss the impact of various measurement technique on the operating system tuning and also
its future development.

5.1. Tuning of VS2

The measurement facility MF/1 can be used to determine, if the VS2 Installation Parameter Set /IPS/ reflects the installations turnaround/response requirements. Suppose that the installation's jobs are divided into three performance groups:

<table>
<thead>
<tr>
<th>Job Class</th>
<th>Performance Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal Jobs</td>
<td>2</td>
</tr>
<tr>
<td>High Priority Batch</td>
<td>8</td>
</tr>
<tr>
<td>Low Priority Batch</td>
<td>5</td>
</tr>
</tbody>
</table>

The above performance objectives are presented on the Figure 35.

The installation performs certain number of MF/1 half-hour measurements in order to find the reason of poor response time at their terminals. Two figure of interest in this case are the average system workload level, and the average response time for terminal transactions. Suppose that these reports indicate that, during the time of MF/1 monitoring, the system was operating approximately at workload level 4 and the average response time for time sharing users was 30 seconds /considered by the installation to be poor response time/.

An analysis of the performance objective specification shows that, at workload level 4, performance objective 2 specified that terminal transactions receive 40 service units per unit time. The installation might reasonably wish to increase service rate for its time sharing users in order to improve the response time. This increase would raise the performance objective curve for performance objective 2, as shown on Figure 35.

Figure 36 shows that raising the service rate for performance objective 2 raises the system workload level, all other factor being equal. Transactions associated with performance objective 5 and 8 would therefore be affected. Transactions associated with performance objective 5 would be most severely affected because their demand curve slopes most severely to the right of workload level 4.

Before the installation modifies performance objective 2, it is able to perform further analysis to determine the quantitative effects of such a change. Suppose the MF/1 reports showed an average of 20 jobs associated with performance objective 2, 8 jobs with performance objective 8, and 4 jobs with performance objective 5. Then, some measure of the average system rate of service supplied to all jobs can be obtained by multiplying the number of jobs at each performance objective by the service rate for that performance objective at workload level 4 /i.e., 20 x 40 + 8 x 20 + 4 x 10 = 800 + 160 + 40 = 1000 total service units per unit time/.

In this case, increasing the performance objective 2 by 10 service units, results in a total increase in demand of 200 service units. Thus, the workload level of the system shifts the right to compensate for this increase in demand, while maintaining the system supply of 1000 total service units approximately constant /see Figure 36/. If the installations considers the service rate provided for all performance objectives at the projected new workload to be acceptable, it may modify the IPS to contain the new performance objective 2.

The installation may then use MF/1 monitoring to verify the accuracy of the projections; repeating the entire procedure, if desired, until satisfactory results are achieved.

5.2. Tuning of SCOPE 3.4

During last five years some important modifications has been developed at INR in order to improve SCOPE 3.4 performance.

The first modification was related to the system disk bottleneck, which was the main limiting factor for the early version of the NRI configuration. At this time CD 841 disk with single access was used as the system disk. The author of this lecture
made an observation, that it would be possible to reduce access time on system disk, if one could place the Frequently Used Routines /FUR/ close together in order to limit disk head movements. The measurements indicated 33 FUR in the peripheral program library. The routines with total size 10 K words /octal/ were loaded 5.3 times per second, what corresponds to 90 percent of all peripheral program loads. Since the size of 841 disk cylinder is 17920 words /decimal/, it was possible to place FUR on one cylinder /see Figure 37/.

On the SCOPE 3.4 deadstart tape the bodies of FUR were placed after other PP-programs. Since FUR should start from the beginning of a new cylinder it is usually necessary to cover the rest of the proceeding cylinder with empty PP-routines. During preloading the PP-program bodies are placed on the system disk. Normal loading program IRCP is unable to load the bodies in such order. Modified IRCP reads program name table /PPNT/ in alphabetical order and rearranges it placing all FUR entries at the end of PPNT. Now the PP-program bodies match PPNT entries and disk addresses are properly written to each entry. After loading of PP-program bodies, the alphabetical order of PPNT entries is restored. With the version SCOPE 3.4 - FUR, the throughput was increased about ten percent.

The second modification was the new scheduling for in input and output queues. According to needs of the majority of users a criterion of the top priority in the input queue for a highest paid job was changed to the top priority for a job requiring the least computer resources. In such a way an essential decrease of a wait time for small jobs has been achieved.

The new algorithm takes into consideration job's magnitude J defined as a function of required system resources. The following table shows the possible values of J:

<table>
<thead>
<tr>
<th>MCM</th>
<th>54000</th>
<th>72000</th>
<th>72000</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>300</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>600</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>600</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

MCM denotes the maximum CM size /octal/
TCP denotes the CPU time in second /octal/
The new expression for input queue priority may be written as follows

\[ P_i = J \cdot 1000 B + P \cdot 100B + A_i \cdot t /1/ \]

where

- \( P \) - financial priority ranging from 0 to 6
- \( A_i \) - aging rate in input queue
- \( t \) - job time in the input queue

Additionally, to obtain significant improvement in throughput of small jobs, the latter ones are treated like express jobs and may enter execution independently from large jobs. It enables us to avoid undesirable situation that small jobs, even with high priorities are completely blocked in input queue, because all JDT entries for batch class are occupied by large ones.

Another algorithm has been worked out for the output queue priority:

\[ P_0 = D \left( \frac{1 - S_0/B}{1 + S_0/C} \right) + A_0 \cdot t /2/ \]

where

- \( S_0 \) - output file size /PRU's/
- \( A_0 \) - output queue age rate
- \( D, B, C \) - constants

As you see from the above formula, the hyperbolic function was used in order to obtain very sharp dependence on file size. The constant B describes the maximum size of output file, which can processed automatically. The files of larger size are processed by operator /\( S_0 \geq 10000B \)/.

Normally the other constants are set \( C = 400B \) and \( D = 7000B \). The above algorithms are very convenient for user, they also lead to the substantial reduction
of the input and output queue lengths.

The third modification was the automatic change of the INTERCOM peripheral program residency. The performance measurement made at INR led us to the conclusion that the residency of some PP programs should be dynamically changed, according to the current system load, jobs mixture and other external circumstances. Finally, a special program NRI was written /D.Mokossa/, which is automatically called by DSD when the operator starts INTERCOM work. NRI creates a CPU job, which moves the proper PP programs into the central memory. After INTERCOM drop NRI moves all these programs back to disk memory. Next, some PP programs useful in batch processing are placed in such released area in the central memory. The rest of this area /about 10 K words/ is left for users programs /see Figure 38/.

The fourth modification was related to the I/O scheduling policy of SCOPS 3.4 level 373. As one can see on Figure 4, the INR configuration consists of single access 844 disk subsystem and double access 841 disk subsystem. The CIA measurements have shown, that the usage of 844 disks was low to compare with the 841 disk usage. For example the experiment K2 for 844 gave the following result:

Number of requests in stack - 1

<table>
<thead>
<tr>
<th>Number of requests</th>
<th>Probability - P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.603</td>
</tr>
<tr>
<td>1</td>
<td>0.263</td>
</tr>
<tr>
<td>2</td>
<td>0.098</td>
</tr>
<tr>
<td>3</td>
<td>0.029</td>
</tr>
<tr>
<td>4</td>
<td>0.006</td>
</tr>
<tr>
<td>5</td>
<td>0.001</td>
</tr>
</tbody>
</table>

From the above data we can calculate the utilization factor:

\[ \phi = 1 - P_0 = 0.397 \] /3/

Similarly the average queue length is:

\[ L = \sum_{i=2}^{\infty} P_1^i (1 - i) = 0.179 \] /4/

For the dual access 841, the Experiment K_3 gives the following probability distribution:

<table>
<thead>
<tr>
<th>Number of requests in stack - 1</th>
<th>Probability - P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.293</td>
</tr>
<tr>
<td>1</td>
<td>0.218</td>
</tr>
<tr>
<td>2</td>
<td>0.164</td>
</tr>
<tr>
<td>3</td>
<td>0.126</td>
</tr>
<tr>
<td>4</td>
<td>0.094</td>
</tr>
<tr>
<td>5</td>
<td>0.061</td>
</tr>
<tr>
<td>6</td>
<td>0.031</td>
</tr>
<tr>
<td>7</td>
<td>0.010</td>
</tr>
<tr>
<td>8</td>
<td>0.002</td>
</tr>
</tbody>
</table>

We can consider dual access 841 disk as multiserver consisting of two servers /M=2/. In this case the utilization factor is given by the formula:

\[ \phi = M_0 (1 - P_0 - P_1) + P_1 = 1.196 \] /5/

The queue length is given by the formula

\[ L = \sum_{i=3}^{\infty} P_1^i (1 - i) = 0.681 \] /6/

As we see from the formula /3/ and /5/ the utilization factor per one server is 50 per cent higher for 841 disks. Similarly from the formula /4/ and /6/, it follows that the queue is 4 times longer for 841 disk then for 844 disk.

The measurements of the channel activity gave the following results:

<table>
<thead>
<tr>
<th>Channel</th>
<th>Equipment</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>First Access 841 disk</td>
<td>0.269</td>
</tr>
<tr>
<td>2</td>
<td>Second Access 841 Disk</td>
<td>0.123</td>
</tr>
<tr>
<td>3</td>
<td>844 Disk</td>
<td>0.286</td>
</tr>
</tbody>
</table>

At first glance it looks like that the 844 disk controller is more heavily used then the 841 controllers. However this fact has no importance, since the dual access on 841 disk is used almost exclusively for one disk drive containing the local files /PUBLIC/. Therefore we can say, that in fact we should add the activities on both accesses of 841 disks and to compare with the activity of 844 disks. The sum is 37 per cent higher than the activity of 844 disks.
From the above measurements it was obvious that more work should be put on faster 844 disks. The investigations, performed by the author and D. Makosa, were concentrated on the mechanism of file opening. In order to find a record block for new or overflowing local file, peripheral program 3DO searches the disk equipment with the lowest ACTIVITY. From the equipment 3DO takes the unit with the highest number of free record blocks. The ACTIVITY is stored for each disk equipment in a byte of Device Activity Table and it is calculated every second by 1RN from the formula:

\[
\text{ACTIVITY} = \frac{1}{2} \left( \text{OLDACTIVITY} + \text{NEWACTIVITY} \right) / 7
\]

\[
\text{NEWACTIVITY} = \left[ S + (S + \text{SPEED}) \cdot \text{COUNT} \right] / D
\]

where

for 841 disk
\[ S = 0 \] /No System/, \[ D = 2 \] /dual access/
and for 844 disk
\[ S = 1 \] /system/, \[ D = 1 \] /single access/
For standard SCOPE 3.4.1 system 1RN contains:
for 841 disk
\[ \text{SPEED} = 8 \]
for 844 disk
\[ \text{SPEED} = 4 \]

Since 844 disk contains the operating system, the value of COUNT is usually high. Therefore local files were opened almost entirely on 841 PUBLIC unit. In order to increase ACTIVITY of 841 disk, 1RN was changed such that for 841 disk
\[ \text{SPEED} = 20 \]
and for 844
\[ \text{SPEED} = 2 \].

As a result of 1RN changes, the utilization factor and the queue length of 844 disk increased from 0.397 to 0.476 and from 0.178 to 0.345 respectively. The utilization and the queue length of 841 disk decreased dramatically from 1.196 to 0.660 and from 0.685 to 0.085 respectively. Substantial increase of the central processor utilization by users was observed from 0.647 to 0.795, which roughly corresponds to 23 per cent increase of the throughput.

5.3. Tuning of KRONOS

The hardware monitor presented in the Section 4.3, was used to specify requirements for KRONOS tuning. The measurements, performed by D.S. Lindsay, were taken during approximately three weeks on the first shift.

The average CPU utilization was for the Supervisor 13.4 % and for user programs 52.1 %. In comparison with SCOPE, KRONOS uses much more CPU time in the supervisor state, because PPMTR executes only limited number of functions. The average utilization of disk channels was ranging from 30.5 % to 38.2 %, therefore the disk channels are remarkably evenly balanced in utilization.

The high I/O-CPU overlap of 54.3 % can be understood, if we assume, that I/O and CPU processing are statistically independent events. Thus the probability of simultaneous I/O and CPU should be just the product of the separate probabilities of I/O processing /89.6 %/ and CPU processing /61.1 %/. The product is 54.7 % and it is almost equal to the measured overlap.

The measurements show that approximately 40 % of real time is spent rolling jobs in and out of CM. Moreover the disk rollout consumes about half as much time as ECS rollout, despite the fact that system only rolls a job to disk if ECS will not hold the job.

The first conclusion is to implement more ECS in order to eliminate slow disk rollouts. The second conclusion is to apply the direct CM/ECS transfer, instead of using 10 times slower distributive data path.

The measurements of Central Memory Lockout has shown this effect to be only 1.68 %. However the number of Exchange Jump is incredibly high namely about 2500 per second. This may cause a substantial overhead, because each Exchange-Jump consumes some overhead in monitor time, perhaps in range 10 - 100 μsec.
5.4. Tuning of CD 7600

The hybrid performance monitor FLIPP was used for tuning SCOPE 2 operating at CERN. The measurements were performed by O. Martin and A. Tengvald. The normal workload at CD 7600 consists of the following job mix:

<table>
<thead>
<tr>
<th>JOB CLASS</th>
<th>% JOBS</th>
<th>% TOTAL CP TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPRESS</td>
<td>50 %</td>
<td>4 %</td>
</tr>
<tr>
<td>SHORT</td>
<td>30 %</td>
<td>11 %</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>16 %</td>
<td>3 %</td>
</tr>
<tr>
<td>LONG</td>
<td>4 %</td>
<td>52 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JOB CLASS</th>
<th>MEAN CP TIME USED in seconds</th>
<th>MEAN TURNAROUND TIME in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPRESS</td>
<td>2.5</td>
<td>3</td>
</tr>
<tr>
<td>SHORT</td>
<td>12</td>
<td>15-30</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>50</td>
<td>NOT</td>
</tr>
<tr>
<td>LONG</td>
<td>370</td>
<td>APPLICABLE</td>
</tr>
</tbody>
</table>

The CPU utilization was 84 %, of which Supervisor took 11 % and the rest is consumed by user jobs.

FLIPP is mainly used for SPOOK display, System Information File /SIF/ and SPY monitor. SPOOK display shows the job statistics, disk space and queues, CPU utilization by various subsystems and job status. SIF is used as input by most of the performance analysis programs and the following two records extracted.

1. The Job Termination Record provides wait time in input queue, time spent in execution, number of tapes staged, job class and job history.

2. The System Activity Record provides information on channel and disk activity/response time, data rates/ as well as on main contributors to the Input/Output load. SPY helps users to monitor CPU time used by various parts of job, and therefore it makes possible the substantial reduction of CPU bottlenecks.

The significant drops in CPU utilization have been observed during peak hours. With the help of the SPOOK display it was easy to diagnose the problem which was in fact twofold:

a. The I/O load was very badly distributed across disk channels and this was the result of the very bad disk allocation algorithm used by SCOPE 2.0, where the least full disk was always selected, and of the high activity on the device holding the NUCLEUS library and the permanent file directory /PFD/.

b. The 817 disks were nearly saturated because they had to perform too many arm movements.

Assuming it takes 100 μsec to handle every disk request and ignoring the fact that every pair of 817 disks share the same two channels one can see that the rate of 300 disk requests per minute /50 % device utilization/ is about the threshold beyond which device contention will develop.

Therefore it was decided to cut down the I/O load and to achieve better balancing of the I/O load. The improvements in these directions led to 20 % increase in CPU utilization, 40 % decrease in disk activity and 50 % increase on maximum job throughput.

It particular it was necessary to change the default allocation and transfer size so that more words would be transferred in fewer disk accesses. The A1/T1 combination was chosen /i.e. 10*512 words on 817 disk or 14*512 words on 844/, because the average access time per I/O request would be about the same whether the file was on the 844 or 817 disks, the cost of transferring 5 more blocks is only 11 ms.

This change was first made for File Routeer I/O requests under SCOPE 2.0 and the effect was very visible. With a 6000-7600 link running at 300 KC/second the corresponding load on the disks is approximately 700 I/O requests per minute if A0/T0 is used and only 350 I/O requests with A1/T1.

The next logical step was then to use the same concept for the NUCLEUS library. The advantages were not a great as one
expected due to the fact that the loader does not know the length of the NUCLEUS routines and only about the displacement within first allocation unit where the routine resides.

The solution to the above problem was to write a utility program to ensure that the most frequently used nucleus library routines are positioned such that they end on an allocation unit boundary.

The NULCEUS library is now kept on an 817 disk unit in A3 style /i.e. 40 512 words/.

In order to obtain the better I/O balancing, the PFD was placed on 844 disk for speed /twice as fast an 817 disk for single block transfer/.

Moreover with the change from SCOPE 2.0 to SCOPE 2.1.2 the disk allocation algorithm has changed. This new algorithm is basically the following:
1. Select a subset of devices eligible for allocation by using a space filter.
2. Select a device from the eligible device list in a round-robin fashion.

Therefore if the space filter is very small the least full device will always be selected as under SCOPE 2.0, but if the space filter is very big, all devices are eligible for allocation and selected in round robin way.

This scheme does not take into account device characteristics and channel configuration and therefore can easily lead to an overutilization of the 844 channel. This scheme has been enhanced by simply adding two more levels of filtering so that the procedure is as follows:
1. Apply space filter.
2. Select devices with the smallest current queue size.
3. Select least active devices, where device activity is number of I/O requests submitted divided by a weight.
4. Select the final device in a round robin way.

At CERN a big space filter was used, which is roughly equivalent to the capacity of an 844 disk and this new algorithm has helped in balancing the I/O load over seven disk channels.

Finally a study on frequency usage of Job Supervisor overlays allowed to save approximately 40 K of LCM by using disk resident groups of overlays. It was then decided to implement the capability to keep the overlays of the FTN compiler in LCM, in order to avoid the overhead of loading them from the device holding the NULCEUS library for every subroutine to be compiled /over 1000 subroutines compiled per hour on average during the day/. The result of this change have been very satisfactory.

Acknowledgements

I am greatly indebted to Mr. Czesław Nowicki from Control Data Corporation and Mr. Andrzej Piewicki from International Business Machines Corporation for the valuable material concerning operating systems. I wish to express my appreciation to my colleagues Danuta Makosa, Jerzy Dzieciszek and Henryk Wojciechowicz for many important contributions to the lecture. I am also very grateful to Miss B. Trenel and Mr. O. Martin from CERN for the documentation about FLIPP.

Finally, I would like to thank Miss Ewa Piwok, who patiently typed the manuscript.

Literature

4. "Introduction to OS/VS2 Realise 2", IBM /1973
10. Skagestein G., "Comparison of NOS and Master" Proceedings of the ECODU-XXIV,
Fig. 1 Configuration of IBM 370/168 at CERN

Fig. 2 Dynamic Address Translation Procedure

Fig. 3 Page in Process
Fig. 4 Configuration of CYBER-73 at Institute of Nuclear Research

Fig. 5 Configuration of CD 7600 at CERN

Fig. 6 MFT Main Storage Organization
Fig. 7 Job Processing in MFT

Fig. 8 Job class and priority scheduling in MFT
Fig. 9 Parallel Task Processing by MVT

Fig. 10 MVT Main Storage Organization

Fig. 11 Virtual Storage Overviews
Fig. 12 Virtual Storage layout /Release 2/

Fig. 13 VS2 Release 2 Control Program Overview
### Fig. 14  Performance Objectives

<table>
<thead>
<tr>
<th>Performance Objective Number</th>
<th>Service Rate for Workload Level 10</th>
<th>Service Rate for Workload Level 20</th>
<th>Service Rate for Workload Level 30</th>
<th>Service Rate for Workload Level 40</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>40</td>
<td>20</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
<td>45</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>70</td>
<td>50</td>
<td>35</td>
<td>15</td>
</tr>
</tbody>
</table>

### Fig. 15  Associating User With a Performance Objective
Fig. 16 Multiple Virtual Machine

Fig. 17 SCOPE Central Memory Layout

Fig. 18 SCOPE Monitor Request Processing
Fig. 19  I/O Processing in SCOPE

Fig. 20  Job Processing by SCOPE
Fig. 21  NOS Memory Layout

Fig. 22  Interactive Job Flow in NOS
### QUEUE PRIORITY

<table>
<thead>
<tr>
<th>JOB ORIGIN TYPE</th>
<th>QUEUE TYPE</th>
<th>ENTRY PRIORITY</th>
<th>LOWER BOUND PRIORITY</th>
<th>UPNNR BOUND PRIORITY</th>
<th>INCREMENT</th>
<th>TIME SLICE CM</th>
<th>INITIAL CPU PRIORITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM</td>
<td>INPUT</td>
<td>400</td>
<td>700</td>
<td>3000</td>
<td>1</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>ROLLOUT</td>
<td>4000</td>
<td>100</td>
<td>1000</td>
<td>2</td>
<td>400</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>400</td>
<td>7700</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BATCH</td>
<td>INPUT</td>
<td>2400</td>
<td>2000</td>
<td>4000</td>
<td>1</td>
<td>400</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>ROLLOUT</td>
<td>2400</td>
<td>1000</td>
<td>4000</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>200</td>
<td>7000</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPORT/IMPORT</td>
<td>INPUT</td>
<td>3400</td>
<td>2400</td>
<td>4000</td>
<td>1</td>
<td>400</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>ROLLOUT</td>
<td>3400</td>
<td>1400</td>
<td>4000</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>200</td>
<td>7000</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TELEX</td>
<td>INPUT</td>
<td>4000</td>
<td>3770</td>
<td>7000</td>
<td>1</td>
<td>400</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>ROLLOUT</td>
<td>4000</td>
<td>3740</td>
<td>7000</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>200</td>
<td>7000</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTITERMINAL</td>
<td>INPUT</td>
<td>6774</td>
<td>6720</td>
<td>7400</td>
<td>1</td>
<td>400</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>ROLLOUT</td>
<td>6774</td>
<td>4000</td>
<td>7400</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OUTPUT</td>
<td>6000</td>
<td>7700</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### DELAY PARAMETERS

<table>
<thead>
<tr>
<th>JS</th>
<th>CM</th>
<th>AR</th>
<th>J4</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>200</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Fig. 23 NOS Job Control Parameters

Batch jobs completed - 406
INTERCOM sessions completed - 83
EDITOR sessions completed - 39
Number of INTERCOM commands - 982
Interactions performed - 403
Maximal number of jobs active concurrently - 16
Maximal sessions active concurrently - 10
Maximal EDITOR users active concurrently - 5
The average number of interactions per command is equal to 1.2
The average response time /total response time/number of commands/ - 8 sec.
The average think time /total INTERCOM queue time/number of interactions/ - 32.9 sec.
The central processor time per command - 0.6 sec.
Approximately 900 commands were monitored
The multiple interactions type of commands - 7.9 %
The use of command classes is the following:
File manipulation - 15.7 %
Batch dispositions - 4.5 %
Permanent file - 10 %
Compilers/Application - 8 %
Load/Execute - 7.5 %
Information - 50.1 %
Miscellaneous - 4.2 %

Fig. 24 WORKLOAD Program Results

- 182 -
Fig. 25  SCOPE STIMULATOR

Fig. 26  HARDWARE MONITOR
### Paging Activity

#### MAIN STORAGE PAGING RATES

<table>
<thead>
<tr>
<th>Category</th>
<th>Rate</th>
<th>Percent of Total</th>
<th>Rate</th>
<th>Percent of Total</th>
<th>Rate</th>
<th>Percent of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAGEABLE SYSTEM AREAS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIO</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>NON VIO</td>
<td>1.83</td>
<td>63</td>
<td>8.48</td>
<td>62</td>
<td>0.79</td>
<td>11</td>
</tr>
<tr>
<td>SUM</td>
<td>1.83</td>
<td>63</td>
<td>8.48</td>
<td>62</td>
<td>0.79</td>
<td>11</td>
</tr>
<tr>
<td><strong>ADDRESS SPACES</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIO</td>
<td>0.06</td>
<td>2</td>
<td>1.71</td>
<td>12</td>
<td>2.19</td>
<td>31</td>
</tr>
<tr>
<td>NON VIO</td>
<td>1.01</td>
<td>35</td>
<td>3.37</td>
<td>26</td>
<td>4.01</td>
<td>57</td>
</tr>
<tr>
<td>SUM</td>
<td>1.07</td>
<td>37</td>
<td>5.08</td>
<td>38</td>
<td>6.19</td>
<td>89</td>
</tr>
<tr>
<td><strong>TOTAL SYSTEM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIO</td>
<td>0.06</td>
<td>2</td>
<td>1.71</td>
<td>12</td>
<td>2.19</td>
<td>31</td>
</tr>
<tr>
<td>NON VIO</td>
<td>2.84</td>
<td>98</td>
<td>11.98</td>
<td>88</td>
<td>4.80</td>
<td>69</td>
</tr>
<tr>
<td>SUM</td>
<td>2.89</td>
<td>100</td>
<td>13.56</td>
<td>100</td>
<td>6.99</td>
<td>100</td>
</tr>
</tbody>
</table>

#### AUXILIARY STORAGE USER POOL

<table>
<thead>
<tr>
<th>Category</th>
<th>Rate</th>
<th>Percent</th>
<th>Rate</th>
<th>Percent</th>
<th>Rate</th>
<th>Percent</th>
<th>Rate</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAILABLE SLOTS</td>
<td>14,321</td>
<td>93</td>
<td>UNUSED FRAMES</td>
<td>145</td>
<td>50</td>
<td>SWAPS</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>VIO SLOTS</td>
<td>7</td>
<td>0</td>
<td>DATA PAGES</td>
<td>145</td>
<td>50</td>
<td>AVERAGE PAGES PER SWAP OUT</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>NON-VIO SLOTS</td>
<td>1,100</td>
<td>7</td>
<td>TOTAL FRAMES</td>
<td>290</td>
<td>100</td>
<td>AVERAGE PAGES PER SWAP IN</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>UNAVAILABLE SLOTS</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL SLOTS</td>
<td>15,428</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Fig. 27 Paging Activity of VS2

---

Fig. 28 Organization of CIA Measurements

- 184 -
## EXPERIMENT K2
NUMBER OF STACK REQUESTS

<table>
<thead>
<tr>
<th>MEANING</th>
<th>VALUE</th>
<th>COUNT</th>
<th>COUNT/SAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8923</td>
<td>.232</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>7737</td>
<td>.201</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6704</td>
<td>.174</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5721</td>
<td>.149</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4289</td>
<td>.111</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2728</td>
<td>.071</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1470</td>
<td>.038</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>649</td>
<td>.017</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>208</td>
<td>.005</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>61</td>
<td>.002</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8</td>
<td>.000</td>
</tr>
<tr>
<td>GT 10</td>
<td>999</td>
<td>1</td>
<td>.000</td>
</tr>
</tbody>
</table>

COUNT = 38499
SUM = 85759
MEAN = 2.23
SAMP–TIM = 4

Fig. 29 The Stack Request Distribution Measured by CIA Monitor

![Diagram](image)

**Fig. 30** Organization of GSS Measurements
### Table: CPT-S Occupation Analysis

<table>
<thead>
<tr>
<th>CPT</th>
<th>BUSY CPT-S</th>
<th>CPT-S FOR CPU</th>
<th>CPT-S IN RECALL</th>
<th>CPT-S RECALL/CPU-IDLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TIMES FOUND</td>
<td>PCT</td>
<td>TIMES FOUND</td>
<td>PCT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>320</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>0.7</td>
<td>2</td>
<td>466</td>
</tr>
<tr>
<td>3</td>
<td>120</td>
<td>8.6</td>
<td>3</td>
<td>367</td>
</tr>
<tr>
<td>4</td>
<td>323</td>
<td>21.5</td>
<td>4</td>
<td>185</td>
</tr>
<tr>
<td>5</td>
<td>365</td>
<td>24.3</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>396</td>
<td>20.4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>199</td>
<td>12.3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>107</td>
<td>7.1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
<td>2.6</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>1.1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>0.1</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

### Field Length Analysis

<table>
<thead>
<tr>
<th>FIELD LENGTH</th>
<th>JOBS</th>
<th>PCT</th>
<th>FIELD LENGTH</th>
<th>JOBS</th>
<th>PCT</th>
<th>FIELD LENGTH</th>
<th>JOBS</th>
<th>PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>020000B</td>
<td>3170</td>
<td>35.5</td>
<td>020000B</td>
<td>115</td>
<td>3.6</td>
<td>020000B</td>
<td>3052</td>
<td>82.3</td>
</tr>
<tr>
<td>040000B</td>
<td>1060</td>
<td>20.4</td>
<td>040000B</td>
<td>752</td>
<td>23.9</td>
<td>040000B</td>
<td>317</td>
<td>18.7</td>
</tr>
<tr>
<td>060000B</td>
<td>2272</td>
<td>17.0</td>
<td>060000B</td>
<td>572</td>
<td>22.7</td>
<td>060000B</td>
<td>660</td>
<td>13.2</td>
</tr>
<tr>
<td>100000B</td>
<td>337</td>
<td>11.5</td>
<td>100000B</td>
<td>731</td>
<td>22.3</td>
<td>100000B</td>
<td>206</td>
<td>3.3</td>
</tr>
<tr>
<td>120000B</td>
<td>355</td>
<td>2.9</td>
<td>120000B</td>
<td>172</td>
<td>3.3</td>
<td>120000B</td>
<td>63</td>
<td>1.3</td>
</tr>
<tr>
<td>240000B</td>
<td>355</td>
<td>2.9</td>
<td>240000B</td>
<td>172</td>
<td>3.3</td>
<td>240000B</td>
<td>63</td>
<td>1.3</td>
</tr>
<tr>
<td>180000B</td>
<td>510</td>
<td>1.1</td>
<td>180000B</td>
<td>79</td>
<td>2.4</td>
<td>180000B</td>
<td>7</td>
<td>0.1</td>
</tr>
<tr>
<td>200000B</td>
<td>21</td>
<td>0.9</td>
<td>200000B</td>
<td>63</td>
<td>1.6</td>
<td>200000B</td>
<td>10</td>
<td>0.2</td>
</tr>
<tr>
<td>220000B</td>
<td>176</td>
<td>2.2</td>
<td>220000B</td>
<td>173</td>
<td>5.3</td>
<td>220000B</td>
<td>3</td>
<td>0.1</td>
</tr>
<tr>
<td>240000B</td>
<td>116</td>
<td>0.2</td>
<td>240000B</td>
<td>14</td>
<td>0.4</td>
<td>240000B</td>
<td>2</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Averages:**

- **Busy:** 5.4
- **For CPU:** 2.2
- **Recall:** 3.3

---

**Fig. 31** GSS REPORT on Control Point Occupation

---

**Fig. 32** Hardware Monitor for KRONOS System
Fig. 33 FLIPP Configuration

Fig. 34 Disassembly Register Bit Assignments
Fig. 35 Changing a Performance Objective

Fig. 36 Effects of Performance Objective Change

Fig. 37 Concentration of Frequently Used Routines on One Cylinder

Fig. 38 Automatic Reloading of INTERCOM Routines
Abstract

In recent years, as the hardware cost/capability ratio has continued to decrease and as much of the routine data processing has been computerized, the emphasis in software development has shifted from just getting systems operational to the maintenance of existing systems, reduction of duplication by integration, selective addition of new applications, systems that are more usable, maintainable, portable and reliable and to improving the productivity of software developers.

This paper examines a number of trends that are changing the methods by which software is being produced and used. More and more of the research and development is now being directed towards producing systems that have the desirable properties mentioned above. Also, more computer-aided tools are being developed and made available. The most important trend probably is the introduction of software development support facilities which provide an integrated set of tools, based on a central computerized data base. At this stage, these systems perform primarily clerical work but gradually their facilities are being expanded. A number of research and development efforts are concerned with moving closer to the ultimate objective of producing executable software for a particular computing environment directly from a set of functional requirements.

A large part of the software developed today is still custom built to meet a specific set of requirements in a particular computing environment. Considerable effort is being directed towards reducing the amount of software that must be newly produced each time by making use of software that already exists.

Many of the concepts and techniques in which these developments are based are not new. Many tools are available but are not widely used. The availability of technology does not necessarily mean that it will be used. Some of the reasons for this are examined as a basis for the consideration of issues involved in practical application of new system development technology.

1. Software Environment

1.1 Computer Based Information Processing Systems and Related Terminology

Large amounts of resources are being devoted to development and maintenance of computer based information processing systems, i.e., systems that include hardware, software and other components. Much of the concern with the efficiency and effectiveness of these systems today is focused on the software e.g., "software is the problem". This paper is primarily concerned with software development, however it is desirable to first place software in its proper context within systems. Terminology is not yet standard in this field, and it is therefore necessary to start with a number of definitions. In general these are consistent with those used in the recent encyclopedias by Salatson and Heek (1976) and Belzer, Holzman, and Kent (1975).

1.1.1 Classification of Systems. The terms such as systems, information systems, information processing systems are used with widely different meanings. The meanings in which they are used in this paper are defined in this section and illustrated in the example in Figure 1.1.1.

Organization

An organization is a legal entity, or sub-unit of a legal entity, that is uniquely and specifically identifiable. Usually an information system is a part of an organization which has a basic function other than information processing. In Figure 1.1.1 the example of an organization is a Manufacturing Enterprise.

Organizational sub-systems

An organization may have a number of sub-systems which are used to accomplish, or contribute to the accomplishment of, the basic function. One of these subsystems is an Information System. In Figure 1.1.1, the Manufacturing Enterprise has sub-systems such as Production, Logistics, Distribution, Finance, etc.

Information Systems

The Information System is the sub-system of an organization in which information (in the form of data) is received, recorded, processed, stored, retrieved and transmitted. The Information System may consist of an informal system and (formal) Information Processing Systems. The informal system consists of the information processing in which data is not recorded.

Information Processing Systems

The subsystems of the Information System in which data are recorded and processed following a formal procedure are called Information Processing Systems. Two kinds of Information Processing Systems may be distinguished: manual and computer-based. Manual systems are those in which all operations are performed manually. Computer-based information processing systems are those in which some operations (though not necessarily all) are performed by a computer. Since this paper is primarily concerned with computer-based systems, the term Information Processing System will be used to refer to Computer-based Information Processing Systems. Each Information Processing System consists of some non-computerized procedures, a data base, application software, and a computing system as indicated in Figure 1.1.1.

1.1.2 Classification of Software. Software may be subdivided into components at various levels: System, subsystem, module, and statement.

Software System. A software system is a collection of software components that, if used together, accomplish a complete information processing function.

Software Subsystem. A software subsystem is a part of software system that, if used together,
accomplishes a separately defined part of the whole software system's function. In very complex software systems it may be useful to distinguish more than one level of software subsystems.

**Software Components.** Software components are parts of software system that constitute basic individual units such as programs, routines, subroutines, modules, units of data description and executable modules.

**Software Module.** The smallest part of a software system that can be utilized by a software component at the same or higher level to accomplish a defined function.

**Statement.** The smallest self-contained component of a Software System which can be expressed in the source programming language.

In a particular system, these definitions can be made more specific since they depend on the operating system under which the system is to be run and the programming language in which it is written. Since the discussion in this paper is independent of operating systems and programming languages, the terms system, subsystem module, and statement will be used with the general meaning given above.

Software may be classified by its operational status. An excellent taxonomy is given by Brooks (1975) Figure 1.1.1:

"In the upper left of Fig. 1.1.2 is a program. It is complete in itself, ready to be run by the author on the system on which it was developed...

There are two ways a program can be converted into a more useful, but more costly, object. These two ways are represented by the boundaries in the diagram.

Moving down across the horizontal boundary, a program becomes a programming product. This is a program that can be run, tested, repaired, and extended by anybody. It is usable in many operating environments, for many sets of data...

Promotion of a program to a programming product requires its thorough documentation, so that anyone may use it, fix it, and extend it. As a rule of thumb, I estimate that a programming product costs at least three times as much as a debugged program with the same function.

Moving across the vertical boundary, a program becomes a component in a programming system. This is a collection of interacting programs, coordinated in function and disciplined in format, so that the assemblage constitutes an entire facility for large tasks...

A programming system component costs at least three times as much as a stand-alone program of the same function. The cost may be greater if the system has many components.

In the lower right-hand corner of Fig. 1.1.2 stands the programming systems product. This differs from the simple program in all of the above ways. It costs nine times as much. But it is the truly useful object, the intended product of most system programming efforts."

Software is frequently classified as either system software or application software. System Software is used during the execution of other software; it includes operating systems, monitors, etc. It is necessary because the application software cannot run on the raw hardware itself. Application Software includes software components tailored to the user's needs, including all software products that are not part of system software as defined above.

The application area that is of most concern in this paper is that of software development. All programs supporting the development of software, such as assemblers, compilers, translators, program generators, design-aid packages and the like are included in this classification.

It will be convenient to distinguish between generalized software and custom-tailored software. Generalized Software is software produced to meet the needs of a number of users for the same general application in different organizations and different computing environments. Custom-tailored Software is software designed to meet specific requirements in a specific situation and for a specific environment.

### 1.2 Environment in Which Software is Developed and Used

Since software is developed in many different circumstances and used in many different situations, it is desirable to classify the different cases and to attempt to identify some parameters which characterize them. Boehm (1976) p. 1239, considers the application of software engineering to two areas:

**AREA 1:** Detailed design and coding of system software by experts in a relatively economics-independent context.

**AREA 2:** Requirements analysis, design, test and maintenance of applications software by technicians in an economics-driven context.

The parameters that he considers in distinguishing between Areas 1 and 2 are:

- whether the requirements analysis is done before or after the design and coding start
- whether the software is "system" or "application" software
- whether the development is done by software experts or by technician programmers
- whether (economic) efficiency is relevant or not.

However, he does not consider other parameters which are also important. One of these is the

----------

1System software is usually considered as being provided by the hardware manufacturer though Dolotta, et al. (1976) predicts a separation between the user's Installation Control Program (ICP) and the vendor-supplied System Control Program (SCP).
motivation which causes the software to be produced. There appear to be at least the following reasons for organizations to produce software: for their own use, for other organizations under contract for sale (or lease) to other organizations, to sell a service, to sell hardware, or to carry out research.

Another important parameter is the way in which an organization meets its requirements: by buying a service, by having a software system developed by others under contract, by purchasing or leasing available software, or by developing its own software.

The combination of these and other parameters leads to the identification of six major environments in which the software problem is important.

**The Systems Department in an Organization.** The unit in an organization that runs the production operation, whose productive results for various users in the organization is frequently called the Systems Department. Its production facilities include computers, systems software, applications software, data bases, etc. The department maintains the existing application software and also develops new systems. Many new systems involve integration of existing systems. The Systems Department may develop new systems, have systems built under contract or acquire existing systems.

**Organizations which acquire software systems by contract.** This environment is similar to the previous one in that the software must be maintained. However since the organization did not develop the software itself, it must depend on the developer for the documentation that is necessary for use and maintenance.

**Software Enterprises.** These are organizations that produce software for others either under contract or for sale as standard products.

**Companies using software to offer a service.** Such business services are becoming more common and are predicted to increase, e.g., Seidman (1975): "The line between computer services vendors and computer equipment vendors is beginning to blur. By 1980, most of us will be buying solutions to problems instead of buying hardware and/or services. In order to deal with the rapidly increasing competition at both the high and low revenue ends of the spectrum, computer services vendors will evolve into Multi-Service Vendors (MSV's), selling solutions to users' problems. This will include, as required, the provision of hardware, software, and facilities management (FM), in addition to the traditional remote computing and batch services currently offered. To the dp user, this will mean a greater opportunity to get the most efficient and most economical solution to his problem with a minimum of comparison studies and installation and integration problems."

**Hardware Manufacturers.** These produce system software, particularly operating systems, and other software that interfaces directly with hardware.

**Academic, research and development organizations.** These produce software primarily to demonstrate feasibility rather than to accomplish applications in a production environment. State-of-art software is produced by experts primarily for research purposes. In this situation, the software is not intended to be used for operational uses, and cost and performance may not be as important as demonstrating feasibility.

The above environments are ones in which a large amount of software must be produced and maintained over a long period of time. During this time the requirements and computing environments may change. In such an environment any given software system is difficult to maintain. Further, since the software is always being changed, it may exist in many versions, each with many releases.

### 1.3 System Development, Software Development and Software Engineering

#### 1.3.1 Evolution of System Development

**Methods.** Software development is only one aspect of Information Processing Systems. It is, therefore, desirable to examine the evolution of system development because improvements in software development must be compatible with the changes in the total environment.

Initially systems were developed by ad hoc methods. Such an approach is practical only for very small systems where only few people are involved. Since it is usually not feasible to divide systems into separate, distinct small systems, this approach is not relevant to the environments described in section 1.2.

The traditional or classical approach to system development is the System Life Cycle method based on a sequence of phases controlled by a project management system. The approach, covered in many books and publications, (e.g., Hartman (1968), Hinz et al. (1974), Metzger (1973)), consists of:

1. Specifying a number of activities that must be carried out,
2. Identifying a number of phases with the results to be obtained in each,
3. Describing the results, i.e., documentation, which must be produced at the end of each phase.

Figure 1.3.1, taken from the front cover of Metzger, illustrates these aspects that characterize all such methods. Frequently, the procedures are designed for a particular type of system. For example, the U.S. Department of the Navy (1974) has a documentation standard for tactical digital systems.

Even though information systems are a type of system, few applications of systems theory have appeared in the literature. One interesting example is the attempt to apply control theory to information systems by Lehman (1969), Belady and Lehman (1971), and Belady and Lehman (1978). Their models of the life cycle of systems include the concepts of releases and versions. Another example is given by Putnam (1977) who analyzes a Systems Department with many systems in different stages of their life cycle.

#### 1.3.2 Software Development

The System Life Cycle method usually includes the following activities in software development:

- Determination of the requirements that the software is to accomplish
- Design of the software including specification of the major modules
- Design of the individual modules
- Programming the individual modules
- Assembling or compiling the programs
- Debugging and testing individual programs
- Integrating and testing the programs as a system

The exact procedure that has to be followed in a particular computing environment depends on the methods used for design, the conventions and standards that have been adopted, the programming language, and the operating system.

1.3.3 Software Engineering and Software Production. While the System Life Cycle method is in widespread use, the model on which it is based is too simple for many practical situations today. One improvement which has been proposed is frequently referred to as Software Engineering. The term has been defined by Boehm (1976) as follows:

"SOFTWARE ENGINEERING: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them."

The Software Engineering concept implies a change from preoccupation with programming to a broader view encompassing software (i.e., a collection of interrelated programs), recognition of the need for tradeoffs among a number of criteria and consideration of the applicability of engineering practices. However, the term still is too narrow for a comprehensive view of systems since software is only one part of systems; systems also include non-computerized procedures and hardware. The non-software or manual procedures include many aspects of user interfaces that determine the value of the system to the organization which is paying for it.

In this paper, the term "Software Engineering" will be used to describe the initial design and development of prototype software. The term "Software Production" will be used to describe all the activities necessary to produce software that is satisfactory for operational use.

1.3.4 Personnel Classification. An important consideration in improvement in software development is the personnel associated with systems and their respective roles. Four major groups of personnel involved with can be distinguished: the software personnel, the information processing personnel, the personnel using the target system and the organization's management. The software personnel are directly involved with the software and are concerned with its development, operation and maintenance. These are Software System Designers, Programmers, the software/hardware Operators and Program Librarians. The second group - the information processing personnel - are concerned with the development, operation and maintenance of the entire organization's information processing system. These consist of Systems Managers, Project Leaders, System Analysts, System Architects, Data Base Designers, Data Base Administrators, System Operators, Hardware Engineers, Communication Engineers, Project and System Librarians. The third group, the target system's users, consist of organization's personnel and its customers or customers' personnel. This group is involved with preparing input data, observing the software format and using the output data to perform its tasks. The fourth group, the organization's management, consists of persons bearing the overall responsibility for the organization's image and the organization's economic effectiveness. Among this group are the managers responsible for the Systems Department in the organization.

1.4 Software Costs and Other Characteristics

While it is clearly desirable to "improve" software, in the software field there is as yet no satisfactory way to measure progress in quality or productivity. Currently, the most common measure is lines of code. Even the basic definition of "code" is subject to interpretation, e.g., does it include comment lines? Furthermore, the use of this measure as a productivity measure leads to undesirable results. If a programmer knows that he will be judged by the number of lines of code he will be motivated to avoid the use of macros and libraries, to use low level programming rather than higher level statements, and avoid generalizing code into reusable units because this reduces the number of lines of code and also tends to make the resulting code more difficult to produce. The lack of satisfactory quantitative and scientific methods and adequate measures is emphasized in most of the surveys of the state of the art of information systems. See, for example, Dolotta et al. (1976).

There are also no generally accepted characteristics in which improvements can be measured. Terms such as reliability, quality, maintainability, etc. are used without definition or with different definitions by different authors. Recently, several efforts have been made to define software characteristics and develop a method for measuring the values of the characteristics for a particular unit of software. See, for example, Boehm (1975), Gilb (1976) and Reifer (1976). The characteristics tree developed by Boehm is shown in Figure 1.4.1. Here characteristics will be divided into two categories: cost and other. The "other" characteristics will be classified as in Telohro (1974).

1.4.1 Software Cost. Three methods for reducing the cost of software to the final user may be considered:

- Reducing the cost to develop software to meet given requirements. This in turn can be accomplished by improving the productivity of software developers or by having more of the software developed by machines at lower cost.

- Reducing the amount of new software which must be developed to meet a given requirement.

- Producing software which will be used by a number of users. The cost to each should decrease proportionately to the number of users. In practice the decrease will not be proportionate since some of the cost such as maintenance increases as the number of users increases. Furthermore, obtaining a number of users to use the same software inevitably requires a marketing effort.

One of the difficulties in improving software is deciding what should be included in the cost of software. It is now being recognized that the
cost of a system is much more than the cost of development. For example, there are costs involved in training and documentation. It is also being recognized that development costs may be small relative to maintenance costs during the life of the system. Decisions on systems are frequently based on cost of software only, and decisions on software are frequently based on development costs only.

1.4.2 Software Characteristics. Ideally software should have three major types of characteristics: accomplishing a given set of requirements, be efficient in its use of resources, be easily changeable as the situation changes.

Achievement of these goals usually involves a tradeoff since improvement in one may result in degredation in others. Systems are built to serve organizations. Consequently the systems must accomplish the requirements that are stated by the organization. A first objective, therefore, is to build systems that accomplish those requirements. Requirements are seldom absolute; the organization and its environment change and therefore the requirements may change. Also, requirements are not independent of cost. There is usually a point beyond which it is not worthwhile to achieve the requirement. There is also a tradeoff in the other direction, i.e., as the cost of a system decreases the organization becomes more willing to adapt itself to the system rather than insisting on requirements which may cost more.

Many of the techniques considered in this paper have a potential of reducing the need for the tradeoff by making it easier to change software to fit a particular situation.

1.5 Improvement of Software Development

The attempts to improve software development arise out of the recognition of the need for changes and the recognition that improvement is possible.

1.5.1 The Need for Improvement. A number of studies of the state of software development have shown that the quantity and quality of software being produced is already a limitation on the effective utilization of computers and that this problem will become even more serious in the future. Two studies were conducted for the U.S. Air Force (CCIF-85 and SADPR-85); the first has been summarized by Boehm (1973). An extensive study has been conducted by the SILT committee of SHARE, Dolotta et al. (1975). IBM has conducted a survey on application development problems (Perry 1975). A general overview of the effectiveness and efficiency of System Department is given by Canning (1975). While these studies contain some quantitative data, comprehensive data is hard to obtain. (An excellent summary of available data is given by Phister (1976).)

Development problems sometimes have to be reported as hypothetical situations as in the MUDI report (Welsh 1975). One situation where quantitative data sometimes appears in situations which have to be resolved by legal methods. (Wiseham 1976).

There seems to be no question that improvement in qualitative characteristics is essential. Furthermore the amount of software needed in the future cannot be produced by present manual methods (Prywee 1975 and Dolotta 1976).

Industries which start with a new product or service go through a life cycle. In the initial growth phase, the industry is rapidly expanding its markets, frequently by expanding the capability of its produce or service. The industry reaches maturity when its product can no longer find new markets; it is then limited to the replacement market. If that market shrinks enough, possibly due to newer products, the industry becomes extinct.

The computer industry has been evolving and clearly is still in a growth phase. The rate of evolution of computers has been very rapid. Evolution of computer hardware is divided into "generations", each of which centers around a major jump in capability and/or decrease in cost. The generation concept has been applied to hardware, (Withington 1974), operating systems, (Benning 1977), RAE management, (Gibson and Nolan 1974), systems development (Benjamin 1972), and systems analysis techniques, (Couger 1973, 1974). The evolution is still continuing, particularly in hardware. The cost of hardware to perform a given computation has been decreasing and the relative cost of hardware to software, in a given system, has also been decreasing.

Since hardware costs have been decreased it is worthwhile to examine the reasons and to see whether the same approach could be applied to software to reduce the cost of software and at the same time improve its quality.

Hardware costs have decreased for three major reasons:

- Research in the underlying phenomena. This resulted in discovery of circuits for performing basic operations at much higher speed with much greater reliability.
- Differentiation of functions involved in the hardware system life cycle, and development and use of appropriate mechanized tools for each of the functions. The separate functions that exist in most hardware manufactures are: engineering, prototype development, manufacturing, marketing, and maintenance.
- Mass production of components and sub-assemblies. This has permitted investment in production facilties to reduce the per-unit cost.

The analogy with software production is not perfect and some of the relative costs are different. For example, the cost to produce another copy of a program is insignificant whereas the cost to produce, for example, another disc drive is not. Nevertheless if one includes all relevant costs associated with information processing system the analogy becomes much closer.

This paper is concerned with the application of the concepts in the last two points above (differentiation of function, and mass production) to software development. In particular Section 2, 3 and 4 describe the tools that might be used in each of the functions in the software system life cycle. Section 5 covers the software analogue of mass production; namely the use of existing software components. Section 6 discusses the technical, operational and economic feasibility of changing the development of software. Any approach to improving productivity should take
into account the evolution occurring in the computer field. On one hand, the type of software that is to be developed and the manner by which software productivity is measured will be affected by evolution in hardware. On the other hand, the investment in software at one stage in the cycle may limit the opportunity to make use of new hardware capabilities at another stage.

One factor affecting productivity is the type of systems being built. Systems are becoming large and more complex. Consequently, software development emphasis is on the sharing of facilities. However, the increase in the availability of mini-computers has led to distributed processing which may both make systems smaller and simpler and resource sharing less important. One example of the possible changes in software development is indicated by Higo (1977):

"We have all heard about Citibank's massive conversion to minicomputers. We are now beginning to hear the rest of the story and it involves a lot more than hardware. In somewhat oversimplified form the current situation was recently described like this:

- Citibank, one of the world's largest corporations, is junking all its big IBM mainframes.
- It is replacing its big machines with minis - dozens of them, of every make and model.
- It has drastically reduced its permanent systems and programming staff. For new development, it rents consultants for as long or short a time as needed.
- It used to take two to five years to design and implement a new system. Now the bank is getting the job done in as little as six weeks."

1.6 Summary and Conclusions

In the development of software that is to be used operationally in computer based information processing systems (see definition in section 1.1), the emphasis must be on "programming system product" as specified by Brooks (1975), Figure 1.1.2.

The primary use of programming system products is in System Departments as defined in Section 1.2. The environment in which these techniques are most relevant is one in which a large amount of software must be produced and maintained over a long period of time during which the requirements and computing environment may change. Any given software system in such an environment is large with hundreds of modules and tens of thousands of lines of source statement. The software must be developed and maintained by staff whose membership is also changing. The software may exist in many versions each with many releases. Techniques that are useful in this environment are not appropriate for situations in which state-of-art software is produced by software experts primarily for demonstration purposes. In these situations software is not intended to be used for operational uses and does not have to be maintained and cost and performance may not be as important as demonstrating feasibility.

Software may be developed by the using organization, by software companies under contract, by software companies for sale or lease or by research institutes. The techniques to be discussed are appropriate to all except the latter.

Currently much of the software intended for operational use is produced under the system life cycle approach as described in Section 1.3. This approach has been characterized by the following:

- most software is developed manually with little use of tools other than compilers, editors, linkers and loaders, and operating systems.
- most software has been designed to satisfy a particular requirement in a particular computer environment.

To improve software development it is necessary to be able to identify the characteristics in which improvement is desired and to be able to measure improvement. These characteristics are examined in section 1.4. The most important are:

- to reduce the cost of software to a particular user
- to improve the degree to which the functional capabilities meet the users needs
- to increase the likelihood that the software functions correctly and efficiently.

A basic premise is that software development and maintenance will change in the following major ways:

- software development in the future will be aided by coordinated set of tools (manual as well as computer aided).
- more of the software development will be done by computers.
- the amount of new software, that would otherwise have to be produced, will be reduced by synthesizing systems out of existing software modules.

Interest in improvement originates with the desire to utilize more fully the potential capability of computer and is pursued because there are indications that improvement is possible, as indicated in section 1.5.

The basic motivation for the approach to software development outlined in this paper is that a substantial reduction can be obtained in the cost of a software system to an individual user while at the same time making it more reliable and more adaptable to changes in the computing environment. However, these new approaches will not be adopted automatically. Some of the reasons why progress has not been faster are examined in section 6.

These considerations lead to the following conclusions:

- The productivity of software developers should be improved by making use of the computer for data processing activities.
- The productivity of software developer can be further improved by automating as much of the software development and

- 194 -
The productivity can also be further improved by reducing the amount of new software to be developed.

A survey of the state of the art of techniques to accomplish these approaches to improvement of productivity is given to provide the specification for a Software Support System. The current status of development along these lines is summarized in the following sections. The tools available for information processing system development are surveyed and several state of the art approaches to integrating the tools are examined in section 2. The specification for a software system capable of supporting information processing systems during their entire life cycle (development, operation and maintenance) is outlined in section 3. Section 4 describes approaches to using the computer to develop software. Section 5 is concerned with reducing the amount of new software that must be developed.

2. Tools for Software Development, Operation and Maintenance

The tools for system development, operation and maintenance may be divided into manual tools and computer-aided tools. The manual tools are discussed in section 2.1. Computer-aided tools are classified and some of the factors that limit their use are discussed in section 2.2. The next four subsections describe briefly some attempts to overcome these limitations: the National Software Library in section 2.3, the System Development Laboratories in section 2.4, the Application Management System in section 2.5 and the Information Automat in section 2.6. Some conclusions on the use of tools are given in Section 2.7.

2.1 Manual Tools

The term "manual tools" is intended to cover all the skills, knowledge, disciplines and procedures that are used in system development, operation, and maintenance. To a large extent these tools have been developed out of necessity in order to use the computing hardware and software as they have evolved. The tools therefore, are ad hoc and in many cases have to be learned by experience since they have not been adequately documented.

In the last few years considerable attention has been directed to the Improved Programming Techniques developed by IBM. These include Chief Programmer Team organization, Structured Walk Through, Program Librarian, Top Down Development and Structured Programming. These techniques achieved considerable attention as a result of the spectacular results reported by Baker (1972) when they were used in a project for the New York Times. Consequently many organizations have adopted these methods and consider them worthwhile.

However, it should be noted that these methods by themselves do not necessarily lead to software that has all the desirable qualities. This is indicated by the experience of the New York Times with the system produced by IBM, as reported by Plauger (1975):

- Number of errors. "The developers claimed, in fact, a record of one bug for every 10,000 lines of delivered code:

- "That's all they found because they didn't do that much testing. We figure that if there's only one bug per 10,000 lines of code, then IBM owes us about a million additional lines."

- Hard to maintain. "Why was it hard to maintain? Among other things it was poorly parameterized. When the staff attempted to introduce a new type of display terminal, they were horrified to discover that all sorts of modules 'knew' how many lines there were to a page, for instance, and how many characters per line. It was a major effort to make the code more general."

- "The major drawback is that three or four experts are still required to maintain the code. Ms. Sznuch estimates that it takes a newcomer about six months to get on board and safely make changes."

- Efficiency. "It was slow, for one thing, too slow to be marketable. And when the maintenance programmers looked inside to see how to tune it up, they weren't terribly happy about what they found. The upshot was that the system limped along for about nine months with only one 'friendly user' (i.e., non-paying), while The New York Times programmers puzzled out the code and hammered it into decent shape. The Data Bank is alive and working today, but it has yet to fulfill its original promise."

- Quality of code. "As for the quality of the code, the Times programmers found it to be highly variable. They soon learned to spot the individual writing styles of different member of the development team. 'You could hardly call it "egoless" code'."

- Design decisions. "But she is critical of some of the design decisions made which, she says, 'meet the letter of the stated requirements but not the spirit.' She suggests that computer professionals have a duty to inform users of specifications that will be regretted later."

This particular example, of course, does not mean that these Improved Programming Productivity techniques should not be used. In fact, practical experience with these techniques in many organizations suggests that they should be adopted, but the desired benefits will not be obtained unless the means to achieve them are explicitly included.

One of the reasons for the lack of adequate manual tools is that there has as yet been relatively little progress in the investigation of the underlying phenomena of information systems; there is, in fact, little agreement as to what classes of phenomena are really basic. Wegner (1976), for example, considers the ramifications of four influential definitions of computer science:


2. Computer science is the study of
algorithms, Knuth, (1968).


4. Computer science is the study and management of complexity, Dijkstra, (1965)."

He concludes that

"these definitions reflect four different approaches to the study of computer science and has given rise to four very different paradigms for computer science research. Computer science is perhaps unique in the diversity of its admissible research paradigms, and this diversity is perhaps responsible for deciding how computer scientists should be trained and for differences of opinion concerning the nature of computer science research."

2.2 Computer-aided Tools

Many computer-aided tools for system development, operation and maintenance have been produced. In particular, there are the compilers, linkers, loaders, operating systems, utilities such as sort, which exist in every computing installation of any size. There are also many other tools intended to aid the development of application software.

Each computing facility contains usually only one operating system and only one or very few compilers. The use of these tools is essential for use of the facility and therefore controlled and enforced. Since the entire operation of the computer facility and portfolio of application software depends on these tools, there is a great reluctance to change the basic system software or even to allow any way to do it. Unfortunately, either the use of application software development tools frequently is incompatible with the existing system software, or the benefits come from overcoming the deficiencies of the system software, and therefore, a basic conflict arises. The problem becomes one of obtaining the benefits from the application software while at the same time maintaining stability in the system software. (The virtual machine concept is one approach to overcoming this dilemma.) Directories of tools to aid in the design, development, operation, modification and maintenance of information processing systems, have been published, e.g., by ICP and Datapro. More detailed descriptions have also appeared. For example, Naftaly, et al. (1972) have described a number of tools available to aid development of systems in COBOL; Reifer (1975b) has provided a list of tools available in the U.S. Air Force. Several authors have attempted to classify available tools: Reifer (1975a) and Curry (1975). Reifer has developed definitions for the 80 types of tools; the types are listed in Appendix 1.

All evidence clearly indicates that despite the existence of many computer-aided tools, very few are being used and most of these are not used very extensively. This conclusion was the opinion of many at Session L351 of the recent SHARE XLVII meeting which was devoted to a "Survey of Facilities for Application Development." The session report contains the following evaluation:

"There are hundreds of tools available to aid the application development process. Some installations have over a hundred separate products installed but find that most are used by only a handful of people and many are used only once or twice. It appears that there are some problems associated with the use of the tools presently available: 1) poor human factors, 2) lack of education and publicity, 3) incompatibilities with installation standards and other tools, 4) not a part of an integrated system, 5) specialized nature of some tools (i.e., only good at serving one particular non-recurring problem) and 6) unsatisfactory performance (i.e., does not improve productivity of the application development process). In addition, most users are not aware of the range of tools available, their benefits, and other users experience with them."

The major reasons for the limited usage of tools are:

- **Knowledge of existence.** Despite the attempt to provide directories of existing tools, it is frequently difficult to determine where a tool with the required characteristics for a given task exists.

- **Access and availability.** A tool may not be available to a person, even if he knows of its existence, understands it and wishes to use it, because he does not have access to the computer system on which it is installed.

- **New languages.** In order to use a tool, it is usually necessary for a person to learn two languages. First is the language used by the tool itself and second is the operating system or command language. The first is almost always a new language while the second may or may not be.

- **Interfaces.** Most tools have their own input and output formats and consequently, each user must reformat his data to be acceptable to the tool and then reformat the output if he needs it for another tool.

- **Status.** Many tools, in the terminology used by Brooks (1975), are "Programs" rather than "Programming Systems Products". They are inadequate user documentation and are inadequately tested. After a certain amount of trying to use a tool without success, the user may well give up and decide the effort to get the tool working is not worth it.

The potential benefit of using tools in systems development has lead to a number of attempts to overcome these difficulties. These approaches are characterized here by the degree to which they make use of what already exists. The first approach starts with the premise that it must be possible to incorporate existing tools essentially unchanged. An example is the National Software Works (Section 2.3). The second approach also makes as much use of existing tools as possible but attempts to integrate them by providing common interfaces. Examples are the Software Development Laboratories described in Section 2.4. A somewhat similar approach is incorporated in the Application Management System described in Section 2.5. The third approach is based on the premise that the basic compatibility problems can only be solved by a completely new start. A proposal in this direction is the Information Automat (Section
2.6).

2.3 National Software Works

The National Software Works (Figure 2.3.1) is an extension of the ARFANET Project, (Bradner 1976). The initial objective of the ARFANET was to provide communication among a number of computer centers so that any person having access to one center could use any of the other computing environments. This helped to simplify the access problem but it did not reduce the other difficulties mentioned above. The user must still know the existence of a tool and the operating system command language of the environment on which it is located before he can start to use it. The National Software Works starts with a premise that the existing tools will remain unchanged and that a "super structure", i.e., a network operating system known as the Works Manager will be placed on top of the present network. This Works Manager will allocate resources, control access to tools, and maintain a central index of available tools. In addition, each computing environment will contain a front end which is a Command Language Interface to the Works Manager. Each host computing environment which contains tools will have an additional software component known as the Foreman that will act as the interface in using the local tools. In addition, each (tool building) host will contain a file package to provide file compatibility.

This system is currently in development and when completed will make it easier for users of the ARFANET to use tools at other hosts and will somewhat simplify the interface problems among tools by the file. However, it does not help to alleviate the problem of each tool having its own language. In effect, it adds another language and command system which a user must learn though this one will hopefully serve him at many hosts so that he can use local tools without having to know the local operating system language.

The ARFANET is a research vehicle, most of the hosts are research computing centers and most of the users are researchers who are interested in tool development. They are, therefore, less likely to produce Programming System Product Packages and also are more likely to tolerate the limitations of other people's programs when they try to use them. To what extent this experience will be translatable into a production software environment is not clear.

2.4 Software Development Laboratories

A number of "laboratories" have been proposed to aid software development by providing an "integrated" set of tools. These exist in two different situations. The first one is a development group which needs tools and in which management decides some investment in providing an integrated set of tools is desirable. Examples of this are the DAIS system of the U.S. Air Force, (Ruth 1973) (Figure 2.4.1), the SDL Laboratory (Naval Electronics Laboratory Center (1976), and the Support Software System described by Crayford, et al. (1976). An early example is Clear-Caster (Browne 1969). A system which has been in operational use for several years is CADS, produced by ICL for support of production and maintenance of an operating system (Fratten and Snowdon 1976). Davis and Vick (1976) describe the SDS system which is intended for support of development of real time systems. In these situations the relevant tools are identified and some investment is made in making the tools easier to use by providing standard interfaces and providing standard data base systems.

Another class of such systems are ones which are basically designed as programming development laboratories. These include de Jong (1973), TOPD (Henderson and Snowdon (1974), ECL (Wegbreit 1971, Cheatham and Wegbreit 1972), and SL-1 (Gardner, et al. 1976).

A proposal for such a system is given by Irvine and Brackett (1976) (Figure 2.4.2). They view their SEF (Software Engineering Facility) as consisting of a System Analyzer, an Interface, a Test and Validation Module, and a Control Monitor. All of these interfaces with a Software Engineering Data Base. Other proposals have been made by Hamilton and Zeldin (1976), Walters (1977), and Fails, et.al. (1976).

Another example of such a system oriented not so much towards software development, but rather towards providing application users direct access to a number of tools is the GMIS system developed by Donovan (1976). GMIS provides the access through the use of the virtual machine concept and through a generalized relational data base management system.

These software development laboratories form the basis for the Software Support System proposed in Section 3.

2.5 Application Management System (AMS)

The Application Management System (AMS) was developed by TREX, Inc; user experience is described by Wright (1975) and in the ED Analyser, Canning (1975). An overview of the system is given in Figure 2.5.1.

The system consists of four major subsystems. Each produces or updates a data base. The first subsystem, the Data Dictionary Generator, accepts data definition and produces the Data Dictionary. The second subsystem, the AMS Compiler, uses this Data Dictionary and produces a data base of data processing modules from statements in a Data Processing Language. This data base together with the Data Dictionary is used by the third subsystem, the AMS Configurator, to produce an executable system called the Run Control File. The fourth subsystem, the AMS Monitor, executes the Run Control File under the direction of the operating system. Each of the subsystems is controlled by a Control Language and each one also produces documentation in addition to updating the AMS data bases.

Wright (1975) describes the use of AMS in the development of two systems LMS and FMS; some information from his paper is summarized in Appendix 2.

---

1 For an interesting discussion of the role of tool development in large projects, see Brooks 1975, p. 127.
2.6 The Information Automat

The Information Automat has been proposed by Wilson (1976). (See also Mills and Wilson (1976)). He begins with the premise that all the support software, including the operating systems, the system utilities, data base management systems, compilers, etc., have been developed independently and each has its own language and its own interfaces. He proposes to develop a kernel system which will be the central software surrounding the hardware to provide these services in an integrated fashion. Hofmagle and Wilson (1976). (Figure 2.6.1). Surrounding this kernel system there will be other layers of support which will provide necessary services for application development and operation. The basic concept is that one language can serve as the communication media for all the various services. His kernel system therefore will provide the basic support that are now provided by a number of independent systems: TSO, JCL, IMS, GIS, FORTRAN, COBOL, PL/I, OS/VS, etc. The central concept is a language known as a "semantic language" which can be used at each level for each service. While the vocabulary and the actions will vary, the structure will be the same and hence it will be simpler to learn by all those who have to use it.

The fundamental advantage of the approach will be to eliminate duplication. Now each tool in the support systems in today's system must have its own language interpreter and analyzer; in the Information Automat kernel system only one will be required. Similarly, all data base management will be done by one program and all the support systems will use the same data base system. This will reduce the interfaces of the various support systems. The Information Automat proposal has been described in some detail in a number of documents, but it has not yet been implemented. It is undoubtedly possible to do so and to produce a system which is much smaller and more compact and easier to use than existing systems. However, the elapsed time and costs required are likely to be large. In addition it is likely to be incompatible with existing systems and would therefore require anyone planning to use it to start afresh with this system. Existing software tools would have to be rewritten to be included.

2.7 Summary and Conclusions

A number of software tools are being developed. However, they are seldom compatible. A user has to learn each of the tools, prepare the input in the form needed, and probably have to use the tools on different computing environments.

Some argue that the software development facilities can now be built, for example, Stillman and Leong-Hong (1975):

"The technology already exists to construct a software production facility. Its components, e.g., advanced programming languages, structured programming precompilers, dynamic analyzers, intelligent text editors and sophisticated file management systems, are within the state of the art. What has not been done is to identify and develop an effective set of tools, integrate them into a single system, and make that system available to a broad spectrum of users."

This, at the present time, is probably a little premature. It is certainly true that the individual components can be constructed. However, the technology to produce an integrated set of tools still remains to be demonstrated. A number of approaches are being tried and more have been proposed as the brief summary given in this paper shows.

The approach followed by the National Software Works will undoubtedly make it easier for computer scientists to use tools developed and installed at another computer environment. However, it will probably not be directly useful to systems departments or for software development in which operational software must be produced.

The approach implied by the Information Automat is no doubt ideal and, if implemented, could result both in a substantial decrease of the size of software required for a complete set of tools and a substantial decrease in the cost of using the tools. However, even if it were implemented, there would be problems with incorporating the already existing software produced under previous methods.

One way to solve the interface problem is to have the tools work from a common data base. The approaches to building programming laboratories are usually based on the concept of one level of user using one basic language. To be more useful these systems will have to serve a number of different types of users with a number of different language levels.

What is likely to be practical and applicable in the next few years is the concept of a coordinated facility for software development that makes use of existing operating systems and some other standard software such as data base management systems. Furthermore, this facility should provide not only for the development of new software but also for the maintenance of existing applications software. The specification for such a facility are discussed in the next section.

3. Software Support Systems (SSS)

This section outlines the requirements for a Software Support System (SSS) which would provide a coordinated set of tools for software development, operation and maintenance. The facilities discussed are those necessary to develop a new software system to meet a given set of requirements in the environments described in section 1. The specifications outlined in this section are based on the analysis of software development systems (some of which are mentioned in Section 2) and on experience with prototypes of parts of a system, such as the Problem Statement Analyzer (PSA), Teichroew and Hershey (1977).

Then, assuming that a basic Software Support System exists, the next two sections will describe additional methods of decreasing the cost and increasing the quality of the completed products. Section 4 is devoted to methods for reducing the amount of software which must be produced manually by having more of the software produced by the computer. Section 5 is concerned with synthesizing a system out of software which already exists, i.e., out of "reusable modules".

3.1 Software Support System as a Decision Support System

- 198 -
It has become fashionable to characterize certain types of information processing systems as Decision Support Systems. For example, Sprague and Watson (1977) define them as follows:

"Just recently, information systems with rather unique characteristics have begun to emerge. These systems, usually referred to as Decision Support Systems, feature decision models, a data base and the decision maker as subsystems and are specifically oriented to supporting decision making."

and state that a

"Decision Support Systems (DSS) should have the following characteristics.

1. The DSS is designed specifically to support decision making. Attention to information flows, report structure, and data base design is specifically related to this primary objective.

2. The DSS is interactive to allow the manager or his representative fast access to models and data. The interactive capability is not necessarily to provide immediate access to minutes-old data, but, rather, to give access to data and models at a speed which matches the thought processes of the manager.

3. The DSS is flexible enough to satisfy the decision making requirements of many types of managers - those in different functional areas, at various managerial levels, and with different management styles.

4. The DSS is an integrated set of data and models which allows the models to work together, and thus avoid suboptimization whenever possible.

5. The DSS is dynamic enough to keep itself up to date without major or frequent ad hoc revisions.

6. The DSS is sophisticated, utilizing modern information processing and management science techniques whenever appropriate."

The Software Support System proposed in this paper can be considered a Decision Support System in the sense that it supports the decision making by System Management (usually a separate department in an organization), Project Leaders, Analysts, System Architects, Software Designers, Programmers, System Operators, Data Base Designers, Data Base Administrators, Project and Program Librarians, and Target System Users, that is, all personnel associated with information processing system and software system development, use and maintenance.

3.2 Users

One of the most effective ways to ensure that a system will be operationally feasible and economically cost effective is to have one system provide for all the needs of a set of users whose needs are related.

All users should be able to enter the results of their own decisions into the Software Support System data base and then should be able to obtain the information implied by the above characteristics of Decision Support System. An overview of the interaction of the various users with the Software Support System is shown in Figure 3.2.1.

Information Processing Systems management should obtain summary reports describing the status of various projects and various systems. The systems department staff should be able to obtain summary information necessary for the coordination of various projects, evaluation of new projects, and determination of the impact of proposed changes. Project Leaders need to obtain relevant data about projects they manage including both management and technical data.

The Analysts should be able to obtain status and checking reports about the data entered into the data base, including its consistency and completeness. After the analysis is completed, they should also obtain the necessary final reports.

The System Architect should be able to obtain the description of the overall system with which he is concerned and all the information provided by the Analysts. The Software Designer should be able to obtain the requirements produced by the Analysts as well as the specification produced by the System Architect. The Programmer should be able to obtain the up to date specification of the programs he is assigned to develop and the source code of programs of similar functions as well as reports indicating whether all program branches were tested against all requirements and whether the programming documentation is complete. The System Operators should be able to obtain an up to date report on the status and progress of the operated application system (input data preparation, error messages issued, handling of exceptions, action report handling by the target users). During operations, the Operation Staff should be able to record operations data and relate it to the system description. Similarly, the Data Base Designer should be able to obtain the requirements for the data base derived by the Analysts and the System Architect. The Data Base Administrator should be able to enforce data base standards and determine the requirements for integrated data bases. The Project Librarian and Program Librarian together with the maintenance staff, should be able to keep track of all the information regarding the project documentation. It is their responsibility to maintain a complete description of the system and a record of changes that have been made. Target System Users should be able to obtain the description of target systems that pertain to them.

3.3 Data Base

In order to ensure that all the relevant data will be available, it will be necessary to store it in one integrated data base in which the various types of information are separately identified. Thus, the appropriate subparts of the information can be selected for use by the various individuals. As with any data base system, appropriate controls must be exercised to ensure that the integrity of the data base is preserved. Individuals must have access to only that part of the information that they are entitled to and are able to modify, change or augment only the part of the data base which has been authorized for them.
The data base structure will be very complex; for example, Figure 3.3.1 shows the data structure for a prototype of one sub-sub-system of an Software Support System which is designed for the storage, manipulation and retrieval of FORTRAN modules.

3.4 Major Subsystems of a Software Support System

The Software Support System would consist of a number of major subsystems as shown in Figure 3.4.1.

The Requirements Determination System allows Analysts and Target System Users to enter data describing existing and proposed systems. The software performs various checks and analyses and produces hard copy documentation on request. It maintains a data base containing the description of the target system from the "user's point of view".

The Development System allows Architects, Software Designers, Data Base Designers and Programmers to develop and construct a system to accomplish a set of requirements recorded in the Requirements Data Base. This system is divided into two subsystems. First, the Software Engineering Facility is designed to permit the rapid and efficient development of a prototype to demonstrate feasibility. This is consistent with the definition of Software Engineering given in Section 1.3.2. Second, the Software Production System is designed to begin with a prototype system and produce a production version.

The Support System allows software systems to be operated and maintained. It is subdivided into two systems. The Operations Facility controls the operation of the target system in an actual environment. It collects statistics that may be used to improve performance. The Installation and Maintenance system provides the capability to install the target system in a particular hardware environment and to maintain it as changes occur. The Management System allows management to maintain plans and obtain data from each of the various project and system data bases. It also provides for all the coordination necessary among projects and systems.

The Data Base System contains all the data that describes the target systems and the projects. It is subdivided into a number of data bases, but the relevant interconnections among all the subdata bases are maintained.

4. Reducing the Manual Effort to Produce Software

4.1 Classification of Methods

In the discussion in the previous section, it was implicitly assumed that the basic tool for software development was an assembly or higher level programming language (FORTRAN, COBOL, PL/I, ALGOL, PASCAL, etc.). The major purpose of the Software Support System was to have the clerical operations performed by the computer. In addition, the Software Support System performed a number of checks and provided management information. In essence the software was still produced manually, though the process was aided by the computer. This section is concerned with methods by which the software can be produced more automatically and, therefore, the manual effort can be reduced. All of these methods basically attempt to reduce the number and effort of manual operations between a statement of requirements and the executable system code. The "decision models" and decision making aids of the Software Support System will be increased and it will become more of a Decision Support System.

The methods for producing executable systems from higher level descriptions may be classified in three ways. One approach to higher level definition is to "specify" larger units of the system. This concept is discussed in section 4.2. Another approach is to present the specifications in a form natural to human beings but which can also be implemented by computer software. These forms are described in section 4.3. The methods by which the executable system is actually produced are outlined in section 4.4. Conclusions on the present state of these methods and trends in their use are given in section 4.5.

4.2 Level of Specifications

Software systems generally may be classified as described in section 1.1.2. When the current general purpose programming languages are used, the structure is completely specified by the programmer and the compiler has very little freedom in changing the structure. Therefore if the compiler is to do more of the work it must have more freedom to receive specifications at a higher level. A number of levels are possible; here the levels used are system, subsystem, and module.

Specifications given at the system level are functional specifications that state what the system is to do but not how it is to do it, at least not in a way that constrains the software structure. There need, therefore, be no correspondence between the structure in the specifications and the structure in the software.

When specifications are given at the subsystem level, each subset of specification will be transformed into a software subsystem. One case where this occurs is where specifications are given in terms of a transaction. The specification for a given type of transaction are implemented as one subsystem; within a single type of transaction the compiler may choose its own software structure.

When specifications are given at the module level, each module is implemented as a software module and within a module the compiler decides on the software structure.

4.3 Form of Specification

The form in which requirements are acceptable as input to the computer may be classified as follows:

Natural language. Ideally, any text in a natural language will be acceptable. The software system starts without any knowledge of the problem domain and builds its entire knowledge on what it obtains from the input. Several systems based on natural languages are discussed by Heidorn (1976).

Specialized languages. The structure of the input is still natural language but certain words or phrases have preassigned meanings to the system. These languages are easier to use and the specifications are easier to implement but each language is restricted to a special application field.
Relational languages. These languages have specific structures and a certain set of reserved words. Input is free format. Examples are the Problem Statement Language (Teichroew and Hershey, 1976) and Requirements Statement Language (Bell, et al. 1977 and Alford 1977).

Forms. Forms have long been advocated as a means of expressing requirements since they are easier to use than languages. Examples of the use of forms are the ADS (Accurately Defined Systems) developed by National Cash Register and TAg (Time Automated Grid) developed by IBM. (A comparison is given by Teichroew 1972). A recent example is the programming language used in The System for Business Automation (SBA) prepared by Zloof and de Jong (1977). An example of a program to produce an invoice is given in Figure 4.3.1. A diagram of the language structure is given in Figure 4.3.2. The language attempts to show the format of the documents graphically. The domain over which the program is to be executed is shown by underlining an item in a table (which is considered to be a relation).

Decision Tables. Decision tables have been advocated as useful tools for specifying requirements. However, there appears to be general agreement that they are not widely used. See, for example, Chvalovsky (1976) and Central Computer Agency (1973). One of the reasons given in the latter report is that decision tables by themselves are not enough, they should be incorporated into a more comprehensive system. Attempts to do so have been made in HSL/1 developed by Hoskins (Rhodes, 1972) and marketed by Martin Marietta and in AMS developed by TRES, Inc. (Wright 1975). The HSL/1 approach to programming is contrasted with the traditional method in Figure 4.3.3. The AMS system is briefly described in section 3.3.

Questionnaires. The questionnaire approach is simply a list of multiple choice questions. A specification consists of the selection of one of each of the available choices. The first formal description of the technique appears to the one given by Ginsberg, et al. 1965. The combination of this technique with decision tables was advocated by Low (1973):

"Programming by questionnaire combines aspects of decision table programming and general purpose programming by using decision tables to construct an application program through the selection of certain source statements from a predefined file. It is proposed that programming by questionnaire is a useful compromise between general and special purpose programming for a significant class of large scale problems."

The most extensive use of the technique appears to be by IBM in the Application Customizer which is used to produce executable code for System/3.

Graphical Languages. Graphical languages appear attractive as a means of specifying requirements since human beings can comprehend a large amount of information when it is presented graphically. Graphical languages have been proposed as supplementary forms to other languages, (see, for example, Young and Kent, 1961). When recorded manually, however, graphical languages must be transformed into input suitable for computer processing. Few organizations yet have extensive capability to input graphical information directly and, therefore, graphical language input is not widely used. This conclusion is also reached by Rose (1976) in relation to specification of hardware:

"Computer hardware description languages (CHDL's) will continue to develop and to be used as "front ends" of conventional hardware design automation systems, but the research emphasis will have been on system descriptive languages such as that used by LOGOS. It is probable that these languages will be string-oriented rather than graphic because of the cost of sophisticated terminals."

Module Specification Languages. These languages focus on specifications at the functional level and usually include some facility to describe relationships among modules. Examples are the languages proposed by Rin (1976), Beck(1976), and DeFremer and Kron(1976).

Augmented (Procedural) Programming Languages. A number of extension to procedural language have been proposed. Some of these incorporate structured programming constructs, for example RAPFOR, (Kennighan and Plauger, 1976)). Others are designed, not so much to reduce the amount of manual programming, as to provide facilities to state requirements which will make the object code more reliable. Another augmentation is in the direction of making the language extensible for data types (Liskov 1975).

4.4 Production of Software Systems

The methods discussed in this section are designed to implement higher level specifications (at one of the levels outlined in 4.2), in one of the forms described in 4.3, and produce an "executable" system automatically with minimum human intervention. For the purpose of this section, it does not matter whether this "executable" system is object code for a given computing environment or whether it is source code in a programming language. In practice, of course, the performance of the system may very well be affected by what form of executable system is produced.

4.4.1 Artificial Intelligence. The input to this method is either text in a natural language or in a specialized natural language. A survey of the state-of-the art in automatic programming natural language dialogue is given by Heldorn (1976). He begins with this introduction:

"Since the early days of computing, effort has been put into automating more and more of the programming process. The ultimate objective in automatic programming is a system that can carry on a natural language dialogue with a user (especially a nonprogrammer) about his requirements and then produce an appropriate program for him. Although the basic idea of 'programming in English' has often been expressed in the literature, only in recent years have any serious attempts been made toward producing such a system."

He then describes four experimental systems and discusses a number of research issues and
concludes with the statement:

"Higher level considerations such as this will have to be dealt with in addition to the more technical issues discussed above before natural language automatic programming can become a practical reality."

One of the systems surveyed is the SAFE system being developed by Balzer (ISI Research Staff 1976). He describes his approach as follows:

"Only modest gains in programming productivity have been produced in 25 years of software research, but the groundwork has been laid for major advances through rationalization and automated aids. This groundwork rests on two critical ideas: that specification must be separated from implementation, and that the separation between these two processes should be a formal operational abstract (i.e., very high level) program rather than a nonoperational requirements specification. Structured programming represents the first results of combining these ideas. It is a special case of a more general two-phase process, called Abstraction Programming, in which an informal and imprecise specification is transformed into a formal abstract operational program, which is then transformed into a concrete (i.e., detailed low-level) program by optimization. Abstract programming thus consists of a specification phase and an implementation (optimization) phase which share a formal abstract operational program as their common interface."

He summarizes the current status as follows:

"Though the results using this example are very promising, and although we have attempted to build a general system capable of handling a wide variety of specifications from many different domains, it is extremely difficult to extrapolate from a single data point. We therefore are planning to present several different and more complex examples to the system during the next year."

There seems to be general agreement that while this approach is promising and research should be continued, it will be some time before this method is capable of producing results that compare with what now is being produced manually.

4.4.2 Software System Generators. The concept of generating complete systems from a set of requirements has long been a dream of application software developers. Recently, the generator concept has been described by Dolotta, et al. (1976). Their description is given in Appendix 3.

The system generator approach differs from the artificial intelligence approach primarily in that some knowledge of either the application area or the software system structure is assumed. The domain is therefore much more limited and specialized.

Three types of software system generators may be considered. In the first, the generator is capable of producing modules and assembling them to accomplish the stated requirements. In its full generality, this is equivalent to the artificial intelligence approach. In practice, however, the domain must usually be constrained in some way.

A second approach assumes the existence of a number of modules. The generator needs only to decide which modules are necessary and provide for the required interconnections. The system based on "Programming by questionnaire" (Ginsburg (1965), Low (1972)) is an example of this level of generators. Another example is automatic program synthesis (Feldman (1972), and Lee, Charnig and Waldinger (1974)).

A third level occurs when a skeleton of the system exists and the generator merely parameterizes it to fit a particular environment. The system generation of an operating system for a particular environment is an example of this level.

4.4.3 Module Preprocessors. The module preprocessors work at the module level, i.e., each unit of specification is transformed into a software module. The methods differ by the degree to which the specification is rearranged or transformed by the preprocessor. In the system proposed by Back (1976) for the specification given in a module specification language are transformed to "Standard Processes." In general, these methods are based on producing a directed graph from the specifications and then partitioning the graph in some way.

4.4.4 Compilers for Augmented Programming Languages. These software systems accept statements in augmented languages (including structural programming constructs and abstract data types) and produce either source statements in the programming language or object code. These compilers expand the augmented statements but generally do not rearrange statements to the extent done by the Preprocessors.

4.5 Conclusions

The preceding discussion indicates that a number of approaches for producing executable systems from higher level specifications have been proposed and/or are being developed. These use different levels of specification, different forms of specification and different approaches to the generation of executable systems. However, the use of these techniques, to reduce the amount of software that must be produced, is so far limited.

System Generators are the method with the best chance to yield a dramatic reduction in the cost of the software components of systems and at the same time improving the effectiveness of the system for the user. However, before System Generators achieve widespread use, a number of problems will have to be overcome. The most pressing problem is introducing methods for persuading the organizational unit that needs a new system, to accept one that is generated rather than custom built. One of the necessary conditions for acceptance of the generated target system is that it, in fact, accomplish the requirements. Thus, considerable skill is required to produce generators that meet a wide enough range of users to justify the investment.

Under the right circumstances, these approaches may contribute substantially to reducing the cost of high quality software. Thus, it is worthwhile
to consider the factors which are currently limiting their use, so that these techniques can be incorporated into the approach outlined in section 6.

5. Reducing the Amount of Software that must be Produced

5.1 Introduction

There are a number of methods, techniques, and approaches that will reduce the amount of "new" software necessary to meet a specific requirement. A person or organization who has a requirement can obtain software to meet his needs by one of the ways identified in section 1.2: develop the software himself, have it developed for him, select from a set of available packages, or select it from a set of available "turn-key" system.

The greater the proportion of the software that is already available, and therefore, does not have to be developed, the greater the potential savings. The software should be less expensive since the development cost is spread over more users, it should be available sooner since it does not have to be developed, and it should be more reliable since it has presumably already been tested.

The developer of software should also gain from using predeveloped software; he should be able to improve his profit through increased sales since each user can obtain the software at a low cost, he would be able to increase the amount of development he does since he becomes more competitive as compared to the custom-tailored developers, and he can afford to invest in improving his development methods along the lines described in sections 3 and 4.

The potential advantages of reusable modules is illustrated by the price charged by one software company, Martin-Marietta, as shown in Appendix 4. According to its advertisements, the company charges $6 per statement for custom-tailored system but only $1 per statement if the system is built entirely out of existing modules.

Despite the apparent advantages of making maximum use of existing software and of designing and building software to have a maximum number of reusable modules, a large amount of the development today is concerned with designing and building new custom-tailored systems to meet each requirement. This section examines the present status of "generalized" software and considers some of the problems that must be solved if the concept is to be incorporated into the Software Support System.

Section 5.2 discusses attempts to use existing software where the software itself can be obtained at nominal cost. Section 5.3 outlines very briefly the current status of the software industry which, by definition, is motivated to produce software with as broad a market as possible. Section 5.4 describes one attempt to develop a method to synthesize systems from modules based on application functions. Section 5.5 and 5.6 outline attempts, at national levels, to develop systems based on reusable modules in the USSR and Japan respectively. Section 5.7 and 5.8 describes some of the issues that must be addressed and outlines some possible approaches in reusing modules and generalized software respectively.

Section 5.9 summarizes how software development in a Software Support System, as defined in Section 3 and augmented by the software generator techniques discussed in Section 4, can be further improved by incorporating the concepts of reusable modules and generalized software.

5.2 Software Distribution

A great deal of software is available for use at a nominal or even zero cost. Nevertheless this software is not widely used. The reasons for this include all the reasons given in section 2 for the specific case of software for development of application software. Large organizations which develop software for their own use also attempt to sell it to others as generalized software. While many packages are available very few of them achieve any substantial number of installations. The most successful is Mark IV which is marketed by Information, Inc. According to Datamation (December 1976, p. 142) there are over 1000 installations of Mark IV which have resulted in more than 40,000,000 dollars in revenue. Analysis of other packages that have been purchased by many users, shows that the most success has been achieved in application-independent packages. Data base management is an example of an area where few organizations can now justify producing a software system for themselves. However, software enterprises have had limited success in marketing generalized application packages.

5.4 Modularization Based on Decomposition of Application Functions

The synthesis approach to software system development attempts to realize a system to accomplish a given set of requirements from a given library of available components. There have been a number of attempts to use this approach though few have been reported in the literature in any degree of detail. One attempt was the Information Systems Engineering effort described by Welsh (1968). It is an instructive example because it involved a relatively large effort sustained over a number of years, because it was concerned with application modules, and because it did not succeed in replacing the conventional custom-tailoring approach.

The effort was motivated by the attempt of a large industrial firm to avoid building unique systems to accomplish essentially similar requirements (payroll, production scheduling, inventory control, etc.) in each semi-autonomous subsidiary. (The first approach was to try to build a system to accomplish a given requirement at one subsidiary and then transport it to the others - this was abandoned as impractical after a few attempts). The approach, summarized in Figure 5.4.1, uses three tools:

- predefined units of information
processing requirements. For example, one Function might be "determine manpower requirements".

**Satisfiers**

- preprogrammed modules or routines that satisfy a given Function. The library of Satisfiers may contain more than one Satisfier for a given Function.

**Configurator**

- a form in which the needs or requirements of a particular set of users are stated.

The synthesis consists of three major steps:

a. The requirements for a system are determined and expressed using the Configurator.

b. A solution to the needs is synthesized using the Functions and the available Satisfiers. If no acceptable Satisfier is available for a given Function, a new one is produced.

c. The system is "assembled" and is then ready for installation and test.

A large part of this development effort consisted of identifying and specifying the Functions. The criteria used to identify the Functions were:

- the Functions had to be mutually exclusive,
- the Functions had to be exhaustive,
- the Functions had to be of approximately the right size.

Approximately fifty functions were identified and described. In some cases a description required several hundred pages. The functions are listed in Appendix 6.

Despite the extensive development effort and intensive attempts to apply the results, the approach is not being used. The reasons are partly technological. Functions that covered data processing, e.g., Message Discrimination and Validation, File Update and Surveillance and Report Data Compiler, were apparently implemented in such a way that the synthesized system could not compete in usage of computer resources with custom tailored systems.

However, the main reasons for the lack of adoption were institutional. The managers of the subsidiaries could not be convinced that there would be sufficient gains to warrant the loss of autonomy that they had with the custom tailored systems. Corporate management on the other hand was not convinced that corporate benefits would be sufficient to warrant forcing the subsidiaries to use the synthesis approach against their will.

5.5 **System Development in the USSR**

The USSR is engaged in a major effort to develop a relatively large number of computerized management information systems (Myasnikov 1974, pp. 88,95). In order to facilitate the development procedure, detailed guidelines have been established and approved by the Union Committee for Science and Technology of the USSR Council of Ministers (1972). The development procedure is based on the concept of synthesizing systems from a number of existing modules rather than custom building a system for each application. The need for standardization of the modules is widely emphasized (Machrov et al 1974, p. 281 and Myasnikov 1974 p. 92). Besides the long-existing Central Institute of Scientific Research in Management and Control Technology in Minak (employing several thousand researchers), in 1974 the computer industry created in Kalinin a Software Development Center charged with developing, from standardized modules, application packages serving all the potential users in the USSR, (Myasnikov 1974 p. 92). The users are strongly urged to build "open" application systems, which could be enhanced in the future when the basic system has already been in operation for some time. The wide use of the first 130 standardized application packages has been recommended by the Union Committee for Science and Technology of the USSR Council of Ministers (Myasnikov 1974, p. 92). The cooperation in this field between the USA and the USSR has become feasible since the same Committee recommended COBOL, ALGOL, FORTRAN and PL/I as basic programming languages and the IBM standards have been adopted for the development of the RIAD Computer Series (Myasnikov 1974 p. 91).

5.6 **System Development in Japan**

In 1973, the Japanese Ministry of International Trade and Industry (MITI) funded a project to make it easier, faster, and cheaper to produce application programs from small chunks of previously written and stored application modules, and assemble these into working, or nearly complete, new programs. The results as reported by Datamation, (September 1976 p. 97), were as follows:

"The five groups, however, each did their own thing with little, if any, management or coordination from a central body. There was not even agreement on the definition of a module. At least one group wrote its modules in FORTRAN. Another used DPL. And one group really didn't know what the others were doing. The result, say some observers in Japan, was a flop."

MITI in 1976 created a new corporation, Joint System Development Corporation (JSD), which has initiated a five year program to produce a Program Productivity Development System (PPDS). An overview of the initial system is given in Figure 5.6.1 (JSD, 1976).

5.7 **Issues in Reusable Software**

In dealing with a complex system, either in analyzing an existing system or in building a new one, it is often beneficial to treat it as a hierarchy of modules comprising the entire system (Simon 1962). The use of subassemblies in manufacturing industry is a well-known engineering application of this principle. With regard to information processing systems, modularization has long been recognized as desirable, and many papers have been written on modular programs, modular software, modular file organization, and modular systems architecture. However, many systems when completed are not, in fact, modular. There are, therefore, many factors which tend to work against effective modularization even if the designers have that as their goal. Some argue that modularization will be achieved only if it is forced and there is no other choice.
5.7.1 How to Modularize. Any given system can be decomposed into smaller systems or components in many different ways. A number of methods for decomposition have been proposed. Some of the problems are:

1. Whether to modularize on application-independent functions or on application-dependent functions. Frequently an examination of application software will reveal some application independent functions that are repeated several times. The identification of these functions and the provision of modules to accomplish them when they are required usually leads to a reduction in the amount of application software that is needed. See for example IBM (1977).

2. How to separate data definition from processing definition. In the early days the lack of distinction between data and instructions was regarded by many as one of the great achievements of the electronic computer. It has become increasingly clear, however, that the distinction should be made and the trend is clearly towards further and further separation.

   - COBOL separated the Data Division from the Procedure Division, thus allowing each to be defined, and hopefully modified, separately though they have to be compiled together.

   - Data Base Management Systems carry this concept one step further by having separate Data Definition Languages and Data Manipulation Languages.

   - Data Dictionaries provide for definition of data elements that contain the user oriented descriptions as well as computer oriented descriptions.

   - Another trend is to distinguish data that describes the structure of the organization, and is used to control processing, from other data. In its past the latter has frequently been added in the programs; the objective now is to remove them from the program and put them in tables so that the tables can be changed without changing the program.

3. How to separate processing from control-flow. By separating modules which perform basic functions from the logic which causes one function to be performed rather than another, the likelihood that the basic modules will be reusable is increased.

5.7.2 What is the Best Size for a Reusable Module. One of the basic questions in reusable module development is the size of the component of software which should be standardized or made reusable. The terminology, referring to size of software components, used here is defined in section 1.1.2. All of the levels system, including subsystem, program, routine and module have been used as the unit of reusable components.

Many programming languages have provisions for incorporating predefined sets of statements, usually called macros. One of the very first concepts in software development was that of the closed subroutine. Almost every installation has, over time, developed a library of subroutines which perform common useful functions, and most programming languages have adopted standard set of subroutines such as SIN, COS, etc. The extension of this principle to the program level results in the many utility functions which are generally available often at the system command level for copying, sorting, editing, etc. Taking advantage of these well-defined and tested capabilities not only simplifies the programming task of the system developer, but makes the developed system easier to understand and maintain.

Turn-key systems are acquired in cases where the buyer does not have the capability to produce the system himself or where he is prevented from doing so by legal or policy constraints. The former occurs primarily in small organizations which up to the present, have absorbed only a small part of the computer industry's products. This will probably be relatively unimportant for the software industry since the turn-key systems are likely to make maximum use of firmware. The latter occurs in government and in some industries under government regulatory control. These situations are also likely to remain relatively unimportant. The second situation arises when the cost of a turn-key system is so clearly less than that of a custom built system that the buyer has no choice. This requires that the buyer know the relative costs of both turn-key and custom made systems including maintenance and other related costs.

While software components at all these levels can be made reusable, the level with the most potential for mass usage in the "module" level. The term "module", as defined in 1.1.2, is a component of software that can be compiled separately but may contain components which can also be compiled separately.

5.7.3 Efficiency. It is generally accepted that modular systems are inefficient because of the overhead involved in linking and calling modules. Camp and Jensen (1976) have investigated the system cost or overhead associated with software modules. They conclude that the added cost is relatively small compared to the savings gained in software development and maintainability. The overhead that currently exists can probably be reduced in future systems by more effective linking and loading.

5.7.4 Parameterization. All generalized approaches, even turn-key systems and those based on modules, require adaptation to an individual installation for three major reasons: hardware configurations are different and change over time, the individual installation requirements differ, and not all want all options, and the software itself is changed to correct errors and improve performance. It is therefore desirable to embed as few parameters in a module as possible.

The early information processing systems were stand-alone applications with their own files. This resulted in considerable duplication of data, duplication of processing operations, and great difficulty in maintaining the systems. For example, one major food processing firm had to rewrite its entire order entry, production control and distribution software when a new product was introduced because the knowledge of the existing
product codes was embedded in the individual modules. Flauger (1976) describes the difficulties caused by lack of parameterization in the New York Times project.

5.8 Issues in Generalized Software

The individual programmer should organize his programs so that he can reuse portions of his code without having to rewrite it. However, the components are usually not used by others since they are not documented well enough. If components are going to be reused they must be developed with this in mind. Provisions must be made for component definition and development, descriptor software designation, definition, maintenance and modification, and a system construction method that compiles and links the object code.

Since all this requires substantial investment, the success of a developer of a system based on reusable modules depends on whether or not there is a sufficiently large market for the modules.

There are a number of different ways in which the subdivision into modules can be made. One of the most pervasive is on whether the function to be performed is dependent on the application or whether it is a data processing function which is independent of a particular application. An example of application independent functions is given in section 5.4.

5.9 Use of the Software Support System

The key problem in basing a system on the utilization of modules is specification of what each module should do. An approach which allows each module to designate or develop its own modules and add them to the library will not be successful. Three requirements are necessary. First, some set of criteria will have to be established for the specification of a module. The criteria will probably include several types of information; both structural and functional. Second, the modules will have to be adequately described, catalogued, and documented so that a prospective user will be able to locate those that might satisfy his needs. Third, the modules will have to be constructed according to some standards for both internal structure and external interfaces.

Generalized systems have had only limited success. If these methods are to be pursued, the reasons for the lack of success must be identified to determine whether they can be overcome or whether the evolution in other areas will change conditions so as to favor generalized systems.

The concepts of reusable modules and generalized software can be incorporated in the Software Support System. This may not immediately and/or completely overcome all the difficulties identified in section 5.7 and 5.8, but considerable improvement can be made. The use of a data base to record all system descriptions automatically enforces a standard definition of a module. Furthermore, the data base contains descriptive information about each module. This will simplify determining whether a module that accommodates a desired function already exists.

Each module will be stored in a data base together with its external description. When a new module is entered into a data base it will be examined and its external description verified as being consistent with its internal operation. It would be possible also at this time to search for possible overlap with existing modules.

6. Summary and Consideration for Progress

6.1 Summary

The concern of this paper has been with surveying the state-of-the-art in software, examining the attempts that have been and are being made to improve software, analyzing the reasons why these attempts have not always been successful, and finally, synthesizing an approach which will overcome many of the reasons for the lack of success in the past. The objective is to propose methods to obtain the same degree of improvement in software as that which has been obtained in hardware. The hardware improvements in capability, reliability and reduction in cost have been achieved through research, separation of engineering and manufacturing functions and mass production. These same concepts can be applied to software, though not as readily.

The environment in which these techniques are most relevant is one in which a large amount of software must be produced and maintained over a long period of time during which the requirements and computing environment may change. A software system in such an environment is usually large with hundreds of modules and tens of thousands of lines of source statement. The software must be developed and maintained by staff whose membership is also changing. The software may exist in many versions each with many releases. The techniques are not necessarily appropriate for situations in which state-of-the-art software is produced by experts primarily for research purposes. In these situations software is not intended to be used for operational uses and does not have to be maintained. Cost and performance may not be as important as demonstrating feasibility.

6.1.1 Major Components. Software can be provided to the user at a substantially lower cost, be more reliable and more adaptable as his environment changes, by an approach that consists of three components.

Coordinated software development tools to provide assistance to the professional software developers. The assistance will be clerical, for enforcing conventions, and managerial. The clerical function will provide storage of the software components and preparation of hard copy documentation. The system will ensure that conventions are being followed, will provide for performing a check of various kinds on the software components. It will also provide managerial functions for keeping track of progress and for evaluating the impact of changes. The system itself however at this point does not produce software; the software is still produced manually but through the use of the computer as an automated information storage and retrieval system. More detailed specifications are given in section 3.

Increased use of the computer to reduce the amount of software which must be developed "manually". These techniques permit the computer to be used to reduce the manual work of actually generating the software and also support decision-making by all those involved with software. The techniques are described in section 4.
Increased use of existing software to reduce the amount of new software that must be developed in a given system. One of the most effective ways to reduce costs and time is to make use of software that already exists and has been fully debugged and tested. The techniques for accomplishing this are discussed in section 5.

In the past, individual approaches in most of these areas have been tried, but progress has been slow. The impediments to progress are not limited to technical problems, though these do exist. More serious are the management, economic, and political problems. Analysis of the various techniques indicates that many of these limitations can be overcome by an integrated approach rather than considering them to be mutually exclusive.

6.1.2 Implementation Objectives. The creation of comprehensive Software Support System should be based on the following concepts:

- An integrated, coordinated data base that contains all the relevant information about the software. The more complete the relevant information, the more the data base will also be able to serve all the needs of various users of the system. A coordinated data base also eliminates the difficulties caused by incompatible interfaces.

- The system should serve as much of the information needs of all the individuals concerned with systems. The more these needs are met, the more likely it is that the individuals will use the system and provide the necessary input data.

- The system should provide for continual evolution. It should be able to accommodate the software which already exists and which must be maintained. It should also be able to provide for future transition to other environments. This includes both hardware and hard software such as operating systems and database management systems.

- The system should have a consistent approach to languages. This will make the entire system easier to understand and reduce the time required to learn to use the system. It will also reduce the cost of implementing the system and extending the functions it provides.

- The system should provide consistent interfaces among the various functions that it provides. The reports should have standards formats so that the contents of one report may be used as the input to another.

6.1.3 Potential Benefits.

A Software Support System with the capabilities outlined above has the potential to provide the same improvements for software as has been obtained in hardware. These will become particularly evident over time as a data base of reusable modules is built up and as the techniques to generate software from a higher level specifications are gradually refined.

In time, the quality of the software produced should be superior than production software produced by these methods will replace software developed by traditional methods. This is exactly the way that commercially produced data base management systems have completely replaced individually developed data base management systems.

6.2 Considerations for Progress

It is obvious that a development of a Software Support System with the capabilities described above is no small undertaking. The technical, operational and economic feasibility of such a system must be examined and the specific actions that are needed for such a system to become viable determined.

6.2.1 Technical Feasibility. The first requirement is the ability to maintain in a data base, all information about the software system to be developed. Data base management systems in existence and have been used in cases such as PSA (Teichrow and Hershey, 1977) and REVS (Davis and Vick, 1976). The amount of information that has to be stored is less than that of many applications in which these data base management systems are now being used. The performance requirements of such systems need to be determined, though again this is not expected to be any major difficulty. It must be recognized, however, that a Data Base Administrator function will be necessary in this application as it is in other applications using data base management systems.

One of the major requirements is to develop an integrated approach to the use of languages for specifying the input to the system, and for commanding it to perform the necessary operations. A consistent approach is necessary as there will be many different users and each requires a language that includes his own vocabulary. A large number of languages, however, become difficult to learn and difficult to maintain. A number of different languages are in existence and have been discussed in section 3. It appears that natural languages will not be suitable since natural language processing is still too difficult. The more promising approach is the use of a relational language such as that used in PSL and RSL, supplemented by one or more levels of procedural programming languages. This approach has the advantage that the terms in the language can be selected so as to be natural to the user. The approach of the specifying objects and their relations is easy to follow and applies to most of the areas which the system is to be used. It will of course be necessary for the data base administrator to ensure consistency in the naming of objects and relationships. Thus, the interrelationships among objects specified by different users can be maintained. The basic relational language can be supplemented by forms, tables, graphical language, etc. as desirable.

It will be desirable to evaluate these languages further to improve their ease of understanding, power of communication, and human engineering.

One of the major advantages of the Software Support System will be the ability to generate executable systems for different computing environments. The techniques available for this have been discussed in section 4. Many of these techniques require further evaluation; however, with the basic approach of the data base and the languages mentioned above, these can be incorporated as they are being developed and proved. The desirability of developing systems that operate in different computing environments can be improved by the techniques which were
identified in section 5.

A number of current limitations in progress in reusing software are identified in section 5. The Software Support System requires, and implies, that all software components can be named. This is the first step in being able to identify and reuse components. The system encourages descriptions that will be helpful in reusing the software. Nevertheless, additional discipline will have to be provided by management to ensure that the adequate description exists for the identification of the potentially reusable software components.

6.2.2 Operational Feasibility. Even if a Software Support System is built there remains the question of whether it would be used. Clearly, such a system is first and foremost a management tool. Its basic objective is to reduce the cost and improve the quality of software from the point of view of the organization that is producing it. Therefore, it is absolutely essential that the approach has full management understanding and support.

However, the major users of the system will be the professionals: Systems Analysts, System Architects, Software Designers, Data Base Designers, Programmers, Quality Assurance, etc. These individuals will use it if it provides functions that they recognize are important, which management recognizes as important, and which the system performs more effectively than if they were done manually. This implies that in the design of the system considerable attention must be given to each one of the functions if the system is to be cost effective.

The use of the system will also depend to a good extent on its ease of use and hence, considerable attention must be paid to making the system as simple to use and as reliable as possible.

Further research and development is required into what functions should be accomplished and how they can best be performed in the system. On the basis of experience gained with the prototypes, such as P2A (which at present provides only a part of the functions of a full Software Support System) it is evident that it is possible to design a system that such professionals will use.

6.2.3 Economic Feasibility. The System clearly requires substantial investment and the initial problem is to obtain the necessary resources. One approach is to build the system software in phases. Thus, benefits are obtained from the beginning and since the system is open ended, compatible components can be added as they become available.

Even if it is available, can such a system compete with present manual methods? There are clearly costs associated with the use of such a system which includes training, software, maintenance, documentation, etc. These costs are all highly visible. The benefits, on the other hand, in a particular application in a particular installation, are likely not to be as visible. They will be hard to measure, and take effect only over a period of time. However, it can be argued that, in general, the out-of-pocket direct costs of producing systems with such a facility should be no more than the present direct costs if they were completely measured.

6.3 Conclusion

For an organization that is continuously developing and maintaining software, the use of a Software Support System should result in substantially lower costs and better overall quality. As this use becomes completely dominant, the benefit of reusing systems and being able to generate systems with less and less manual intervention will become more evident. If these design objectives are successful, the system should eventually become indispensable.

References


Blosser, Patrick Alan, An Automatic System for Application Software Generation and Portability, Ph.D. Dissertation, August 1976, Purdue University.


Canning, Richard C., "Do We Have the Right Resources?" EDP Analyzer, Vol. 13 No. 7, July 1975.


Computer Weekly, "Hewlett-Packard enlists the marketing aid of Hokusy," July 15, 1975


Datapro, *Directory of Software*, Datapro Research Corporation, New Jersey


Dijkstra, E.W., "A constructive approach to the problem of program correctness, BIT, August, 1968.


Ginsberg, A.S., Markowitz, H.M. and Oldfather, P.M., "Programming by Questionnaire," RM-4460-RR, RAND CORP., Santa Monica, Calif. 1965


Hofnagle, G.F., M.L. Wilson, "The Kernel System of the Information Automat," IBM, 1800 Frederick Pike, Gaithersburg, MD 20760, April, 1976


ICP, ICP Quarterly, International Computer Programs, Inc. Indianapolis, Indiana


Leavitt, Don, "Center to Aid Sharing of Federal Software," Computer World, 1976


Lucena, Carlos J., Cowan, Donald D., "Toward a System's Environment for Computer Assisted Programming," Department of Applied Analysis & Computer Science, University of Waterloo, Waterloo, Ontario, Canada, CS-76-06


Myasnikov, V.A. "Computer-aided management today - Experiences, problems and prospects." Article written by the Head of the Board of Computer Technology and Management and Control Systems at the Union Committee for Science and Technology of the USSR Council of Ministers. EKO = Economics and


Naval Electronics Laboratory Center, “Systems Design Laboratory Preliminary Design Report,” Naval Electronics Laboratory Center, San Diego, California, NELC N725 TR7145


Ruth, John C., “DAIS: The First Step,” NAVCON ’73 Record


Takasaki, I., EDP-IT’s Production and Application in Japan, Proceedings Third Annual Computer Applications Symposium, Chukalongkorn University, Bangkok, Thailand, July 1972.


Union Committee for Science and Technology of the USSR Council of Ministers — General methodological guidelines for development of computer-aided management systems in enterprises. ( Gosudarstvennym Comitet Soveta Ministrov USSR po Naukii i Technike — Obshchiorientalievie rukovodstvenoie materialy po soderzhanii avtomatizirovannykh sistem upravleniia predpripriatiiam, Minsk 1972.)


Wilson, Max, "The Information Automat," Proceedings, SHARE XVI, Session L103, Montreal, August 15-20, 1976


Wright, A.W., "User Experience with the Application Management System (AMS)." Info '75, New York September 9, 1975.


Appendix 1
Types of Tools Defined by Reifer (1975a)

1. Accuracy Study Processor
2. Analytical Modeling
3. Analyzer
4. Automated Test Generator
5. Automated Verification Systems
6. Bootstrap Loader
7. Comparator
8. Compiler
9. Compiler Building System
10. Compiler Validation System
11. Compressor
12. Consistency Checker
13. Constants Auto Checker
14. Correctness Proofs
15. Cross-Assembler
16. Cross-Reference Program
17. Data Base Analyzer
18. Data Definition Language
19. Data-Exception Handler
20. Decompiler
21. Design Language Processor
22. Diagnostics/Debug Aids
23. Driver
24. Dynamic Simulator
25. Editor
26. Engineering/Scientific Simulations
27. Environment Simulator
28. Emulation
29. EQUATE Program
30. Extensible Language Processor
31. Flowchart
32. Generator
33. Hardware Monitors
34. Instruction Simulator
35. Instruction Trace
36. Interface Checker
37. Interpreter
38. Interrupt Analyzer
39. Language Processors
40. Library
41. Linkage Editor
42. Linking Loading
43. Loader
44. Logic/Equation Generator
45. Macroprocessor
46. MAP Program
47. Modular Programming
48. Overlay Program
49. Peripheral Simulator
50. Postprocessor
51. Preprocessor
52. Process Construction
53. Production Libraries
54. Program Flow Analyzer
55. Program Sequencer
56. Record Generator
57. Report Generator
58. Requirements Language Processor
59. Requirements Tracer
60. Restructuring Program
61. Scoring Program
62. Simulator
63. SNAP Generator
64. Software Monitor
65. Standards
Appendix 2

"APPLICATION PROGRAM DIMENSIONS"

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>527</td>
<td>474</td>
<td>1,001</td>
</tr>
</tbody>
</table>

SOURCE STATEMENTS EXECUTABLE

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>166,488</td>
<td>86,851</td>
<td>253,139</td>
</tr>
</tbody>
</table>

COMMENT/DOCUMENTATION

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>20,538</td>
<td>28,221</td>
<td>48,759</td>
</tr>
</tbody>
</table>

TOTAL

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>187,026</td>
<td>114,872</td>
<td>301,898</td>
</tr>
</tbody>
</table>

PRODUCTIVITY ANALYSIS

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 Mos</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MANDAYS EXPENDED

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,235</td>
<td>2,375</td>
<td>3,610</td>
</tr>
</tbody>
</table>

TOTAL SOURCE STATEMENTS

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>166,488</td>
<td>86,851</td>
<td>253,139</td>
</tr>
</tbody>
</table>

AVERAGE SOURCE STATEMENT PER MANDAY

<table>
<thead>
<tr>
<th>LMG</th>
<th>FMS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>151.4</td>
<td>48.4</td>
<td>83.6</td>
</tr>
</tbody>
</table>

NOTE: MANDAYS ARE ADJUSTED FOR 40-HOUR WEEK AND 21 DAYS PER MANNORTH

DEVELOPMENT BENEFITS

- simplicity
- reduced amount of code AMS language facilities eliminates application programming
- standard system architecture eliminates part of design phase
- documentation machine generated as development progresses
- problem resolutions faster - debugging and testing aids

OPERATIONAL BENEFITS

- reliability
- maintainability
- extendability
- non-obsolete

Appendix 3
Application Generator Proposed by Dolotta (1975)

"The primary purpose of this study is to point out those requirements of the end users that imply significant research or development efforts, rather than to attempt to present complete solutions to meet such requirements. It seems appropriate, however, to outline an example of the type of solution that could meet the rather severe set of requirements given above.

One can envision an applications generator. This tool can be implemented in several different forms, each meant for a specific type of application (e.g., payroll), or for a class of applications. The developer and the applications generator would interact via a "menu" of alternative selections (primarily multiple choice, or true-false) or a television-like console. The generator would view each application as a hierarchically organized set of building blocks that are to be tied together in a manner dictated by the choices offered by the generator and selected by the developer.

At the beginning of the interaction, the choices offered to the developer would consist of high-level function or general intent alternatives. Further on in the development "session," the offered alternatives would be at more detailed functional levels, until input formats, output formats, data base requirements, etc., would be specified at the most detailed level. At every level, however, alternatives offered to the developer by the generator would be limited to those consistent with the higher-level selections already made earlier in the "session."

Such a generator cannot, of course, contain all conceivable functions that will (or might) be desired. This introduces the requirement for always being able to select an alternative that is "none of the above." In this case, the developer would write a new module (or a hierarchy of new modules) to fulfill the new requirements. This would be done in a highly standardized and structured manner, so that, once implemented and tested, the new functions would be automatically added to the applications generator and could then be selected by subsequent developers."
At the completion of the specification process, the generator would bind together the modules of code (which could consist of a combination of object modules, generated source modules, and interpretive modules) necessary to perform the specified functions. This could (and almost always would) be followed by an interactive checkout phase, where either automatically generated or user-specified test data would be run through the resulting applications system for verification by the developer and by the end user.

It seems quite feasible to develop such an applications generator for a specific type of application, such as inventory control. Although this could have a significant impact on an industry-wide basis, the effect on productivity in any one given installation would not be revolutionary. It is also conceivable that generators could be developed for broad sets of applications, such as for on-line transaction systems seems conceivable today, and the development of one such generator has been reported in the literature [Maynard 1974]. Given the expected requirements for this type of applications, such a generator’s impact on development productivity could be substantial.

Assuming that such an approach is practical, it is easy to see how the objectives stated above could be realized. For example, if we look at the difficulties the end user encounters in trying to convey his requirements to the system developer, we can see that this approach would help by allowing the end user and the developer to work as a team in interacting with the generator; conceivably, it could also allow the end user to interact directly with the generator; the applications developer would then be needed only in situations where a new function is required.

One of the basic strengths of this approach is the restructing of the application that is gained by the existence of the generator and by the participation of the generator in its dialog with the developer. One of the severe weaknesses the industry currently suffers from is that the responsibility for initiative in application structuring is vested in the developer and is, therefore, unique for essentially every occurrence of each application. This has defeated most previous efforts to use standardized modules as building blocks in new applications.

It is apparent that this approach is an extreme departure from current development techniques. If it is, in fact, necessary to make such a substantial departure from current methods (and we believe that it is) in order to solve the programmer productivity problem, then we cannot overemphasize the need for extensive research and development efforts in this area."

Appendix 8

Prices for Software Development, Martin Marietta (1976)

"SYSTEMS AT $6.42+. $1 PER PROGRAM STATEMENT"

In-house programming costs $6.42 to $10 per line including testing. We build customer systems at $6.42 per line; price, performance and delivery guaranteed. You don’t save much money but you may save time and avoid schedule bottlenecks. You do get flexibility, access to specialist skills and assured results.

Working at your site and in our offices, we deliver programs at $4 per line. We design robust systems: easy to use, test and modify. We pay attention to detail in coding, documenting and testing. We provide a maintenance warranty.

Using off-the-shelf Modular Application Systems (MAS) building blocks, the cost reduces to $2 per program statement, for a system to your exact specification. In some respects, the quality exceeds that of custom built systems: economy of scale permitting unusual refinement. MAS, being built to flex, adapts to changing company needs. So, over the years, the MAS approach can offer a better fit that a custom built system, while avoiding code redundancy.

If only standard MAS modules are used, the resultant system will cost $1 per line. You get all the benefits of MAS: performance proven in over 500 implementations, provision for total integration and honed down operating costs."

Appendix 5


"The Federal Software Exchange Center (FSEC), intended to foster sharing of programs between federal agencies, is now becoming operational under the General Services Administration (GSA) Automated Data and Telecommunications Service. Outlined in a Federal Property Management Regulation (FPMR 10-12-16) issued last February, FSEC's stated goal is to reduce 'overall costs, time and use of personnel resources for software acquisition and/or development.' The FPMR calls for the pooling of information of 'common use software' by all federal agencies 'having [DP] facilities, resources or requirements.'

Once gathered, this information is to be maintained in a catalog, published and updated quarterly by the National Technical Information Service. Agencies will be required to search through the listings of what is available from FSEC before they are allowed to acquire any software from outside sources, according to Chris Bythewood, who has organized the operation at GSA.

Software covered by the exchange is limited to programs written by agency staffs or by outside contractors working for agencies. Explicitly excluded are programs that are classified, proprietary or 'developed with revolving funds' or software 'to which the government does not possess the full rights of ownership', in the language of the FPMR.

Though developed with federal funding, programs in the FSEC will be considered 'property' and therefore not in the public domain. Only federal agencies will have access to them, Bythewood said, noting however that the status of the software is currently under legal review.

Government 'property' cannot be given away (to a nongovernment user, for example) without specific authorization, he explained. On the other hand, an agency ‘giving away’ a copy of a software routine still has the routine for its own use and hasn't
lost any real property at all - which makes a very awkward situation, logically and legally," he said.

While one part of the PFM defines what agencies must do to support the creation and maintenance of the FECC library, another paragraph outlines the expected benefits from use of the exchange and another states what can happen if agencies try to bypass using the center altogether."

Appendix 6
Functions Defined by Welsh (1968)

ENGINEERING FUNCTIONS

Determine System Requirements
Determine Logical Design
Determine Physical Design
Determine Organizational Assignments
Determine Procedures
Establish Time Values
Determine Labor Classifications
Coordinate Design Introduction

ADMINISTRATION FUNCTIONS

Performance Determination
Performance Evaluation
Performance Projection
Decision Aids and Simulation
Personnel

Develop Labor Hours
Analyze Attendance
Compute Payroll
Develop Purchased Material Costs
Develop Service and Loading Rates
Develop Labor Rates
Develop Standard Costs
Develop Price
Compute Billing
Compute Recoveries
Compute Disbursements
Compute Actual Costs

SYSTEM CONTROL FUNCTIONS

Message Discrimination and Validation
File Update and Surveillance
Report Data Compiler
Data Services

RESOURCE CONTROL FUNCTIONS

Forecast Requirements
Determine Safety Requirements
Determine Net Requirements
Determine Economic Quantities
Aggregate Economic Planning
Detailed Economic Planning
Determine Dispatch Priority
Select Vendor
Determine Changes From Previous Plans
Material Planning

A MANUFACTURING ENTERPRISE

![Diagram of a manufacturing enterprise structure](image)

FIGURE 1.1.1 EXAMPLE OF SYSTEM STRUCTURE

- 215 -
Figure 1.1.2 Evolution of the Programming Systems Product

Figure 1.3.1 The Program Development Cycle

Figure 1.4.1 Software Characteristics Tree
Figure 2.5.1 National Software Works

Figure 2.4.1 DASIS Software Design & Verification System (SDVS)
Ruth, J. C., "DASIS: The First Step," DACCON '75, Record.

Figure 2.4.2 SEE: Processor View
FIGURE 2.5.1 Application Management System (AMS)


IA KERNEL SYSTEM

I. Basic User Functions
   Basic user services
   Concept definition & modelling
   User instruction & interrogation

II. Data Structure Operations
    Basic data structure operations
    Direct data entry
    Document/Graphics processing
    Data migration

III. Procedure Operations
     Interactive problem solving
     Basic procedure operations
     Language processing services
     Procedure testing
     Procedure migration

IV. System Management Functions
    System administration
    Control program services

TODAY'S SYSTEMS

TSO, JCL, SPF,
IMS/DC, CRJE, IQF,
REL, GPSS, COURSEWRITER, etc.

DBDA, IMS, GIS,
VIDEO/570,
RPG, AMS, STAIRS,
etc.

BASIC, APL,
ALC, FORTAN, COBOL, PL/I,
PLAN, VS, XPL,
TESTMAR, CLEAR,
etc.

SMF, RS, KL,
OS/VS, RTG, CICS, etc.

FIGURE 2.6.1 Comparable Facilities in the Kernel System and Today's Systems


FIGURE 3.2.1 Software Support System
FIGURE 3.3.1 Data Structure of Selected Subsystem of Software Support System

FIGURE 3.4.1 Major Subsystem of Software Support System
FIGURE 4.3.1 Example of SBA Programs to Produce an Invoice

FIGURE 4.3.2 Summary of SBA Programming Language
FIGURE 4.3.3 HOYNSM System Language (HSL)

FIGURE 5.4.1

FIGURE 5.6.1 PROGRAMMING PRODUCTIVITY DEVELOPMENT SYSTEM (PPDS)
JOINT SYSTEM DEVELOPMENT CORPORATION, "AN OUTLINE OF THE PROGRAMMING PRODUCTIVITY DEVELOPMENT SYSTEM (PPDS)," TOKYO, DECEMBER 1976

- 221 -
ABSTRACT

The topics covered is "Electro-optical means of implementation of a special-purpose processor dedicated to high-energy physics experiments". Basic principles of optical computing are given and some means to overcome the limitations of the techniques are treated : analysis of real time electro-optical transducers in respect of their matching to high energy detectors, special complex filtering dedicated to pattern analysis.

1. Introduction

The subject is limited to a study of special-purpose processors \(^1\), because, today, general purpose optical computers aren't effective. The reasons why special purpose processors are used are mainly :
- they are more cost effective than general purpose processors and,
- they permit to overcome real-time constraints (go faster ...).

If one analyses why general purpose processors are limited, one finds that part of the limitation is due to their basic principle : sequential processing of elementary computing steps. This limitation is too valid for classical electronic special purpose processors.

It follows that a good technique would be to switch to a processor able to do complex calculations (such as direct computation of integrals) in a parallel manner. Associative parallel processors are of such a type. Electronic \(^2\) \(^3\) and optical implementations exist and a hybrid optical implementation will be described here.

But let us be very careful : digital techniques possess many advantages about which the tremendous growth rate of LSI, VLSI is such, that it is very well possible that the all sequential digital way will achieve the promises of the parallel analog processor before the latter, and cheaper !

This leads to a first remark : analog representation of variables is limited in precision, thus a good rule to follow is to code the analog representation to become digital (hence a hybrid processor ...).

Why then, think about the use of an optical processor ? Because of the tremendous potential possibilities of such processors, as they are demonstrated in image processing techniques :

The true processing time (no input/output time included) to recognize and extract a given pattern in an image 512x512 points (8 bits) is of the order of 2 nsec ! This means 262 1/4 correlation integrals computed, and an information throughput of \(\approx 10^{15}\) bits/sec.

But loading of the image (input time) and reading of the result (output time) decreases the figure by a large amount, depending on the transducers used as will be seen and discussed here.

2. Principles of Optical Computing

Optical Computing is based upon the Fourier Transform property of a spherical lens. Hence the calculations use all the Fourier Transform Mathematics and associated correlation / convolution integrals. See ref. \(^4\) for a refresh of these mathematics.

2.1. General principle

A spherical lens which images a coherently illuminated object, produces a Fourier Transform at a plane intermediate between object and image plane \(^5\), \(^6\). Although the lack of a rigorous proof of the fundamental diffraction integral creates some questions about the complete validity of the relationship, it is true that the relationship is valid for almost every practical system.
Remark that coherently illuminated systems are linear in amplitude, versus incoherently illuminated systems which are linear in intensity.

A physical reasoning \(7\) able to make comprehensive the Fourier Transform relationship of a spherical lens, follows (fig. 1).

![Fourier Transform relationship diagram](image)

**Fig. 1 Fourier Transform relationship**

On Fig. 1, the lens performs like a collimator, which means that for a point source, mathematically expressed as a Dirac impulse, the diverging bundle of rays is converted into a plane wave where aperture \(A\) limits its extend, and thus the bandwidth of the system.

As the point source is displaced by \(x_o\) from the optical axis, the plane wave is tilted at an angle \(\gamma = \arctan \frac{x_o}{F}\) (\(F\) : focal length of the lens). This means that the light distribution in \(P_2\) is again uniform, but that the phase varies progressively in \(P_2\). This is expressed by the relationship

\[
\delta(x-x_o) \rightarrow \exp(-jpx_o) \quad \text{with} \quad \frac{2\pi x_o}{\lambda F}
\]

which is the Fourier Transform of a displaced Dirac impulse.

One may now admit, that as any image is a sum of displaced point sources, which each produces a tilted plane wave which all add in \(P_2\), that the light distribution in \(P_2\) (frequency plane) is the Fourier Transform of the input image.

2.2. **Spectral analysis : Fourier Processor**

2.2.1. **Spatial Frequencies Wiener Spectrum.** In plane \(P_2\) we have an amplitude distribution. If one records this spectrum with an intensity sensitive medium (such as a photographic emulsion), then a Wiener Spectrum is recorded (intensity spectrum). An obvious remark is that we speak about spatial frequencies which are the inverse of the distance between periodic image points.

The simple optical scheme presented, allows then a record of the Wiener Spectrum of an input image placed in \(P_1\). References \(8\) \(9\) \(10\) \(11\) \(12\) give a complete review about this optical process and on the general topics of optical computing. A lot of examples are shown, from the simplest, the airy disk, which is the Fourier Transform of a circular aperture; to the more complex situation of aerial image analysis.

2.2.2. **Frequency Plane Filtering.** It is now understandable that some filtering action may be done. The principle is to use a spatial filter to modify the signal to extract the desired information. In other words, we place in \(P_2\) a transparency which possess an amplitude (attenuation) response and a phase (optical path difference) response. The light distribution in \(P_2\) is then modified by this spatial filter (complex transmittance). For instance if one places in \(P_2\) a black spot on the optical axis (spatial frequency 0), then the DC component of the existant spectrum does no more exist: this is a removal of the DC component of an image, or a contrast improvement, which may be observed on the image restored by a second Fourier Transform lens. More complex filtering may be done by more complex filters (ex : a black ring centered on the optical axis acts as a band rejection filter) which implement also some phase filtering action. But we must remark that these phase filters are difficult to implement, because it is necessary to achieve an optical path difference, i.e. a refraction index variation or a thickness variation of a material.
2.2.3. Matched filtering. This kind of filtering realizes both amplitude and phase filtering, and this, matched to a reference object. It is the optical (and bidimensional) equivalent of matched filtering in information theory, which states that if we consider an input signal \( s(t) \) with a constant phase and the signal-to-noise ratio \( \frac{S}{N} \) will be maximum if the frequency response of the filter is optimal:

\[
H_{\text{opt}}(f) = \frac{K}{N_{bb}(f)} P^*(f) \exp(-2\pi j f t_0)
\]

with \( P^*(f) \) the complex conjugate of the Fourier Transform of \( p(t) \) and \( N_{bb}(f) \) the spectral density of noise \( n(t) \).

This optimal filter reduces to a matched filter as we suppose that \( N_{bb} = c = A \).

Thus:

\[
H_{\text{opt}} \rightarrow H_{\text{matched}} = \frac{K}{A} P^*(f) \exp(-2\pi j f t_0)
\]

\[
= C_0 P^*(f)
\]

If one places such a filter in the Fourier Frequency plane of a correlator (Fig. 2) it yields to following equations:

In the input plane \( P_1 \) there is a transparency which modulates the light amplitude by \( s(x,y) \).

In the frequency plane \( P_2 \), this leads to an amplitude distribution \( S(u,v) \) with \( S(u,v) \) the Fourier Transform of \( s(x,y) \), which is modified by the filter.

Thus the light distribution becomes \( C_0 S \odot S^* \)

which after a second Fourier Transform yields to an amplitude distribution \( C_0 S \odot S^* \odot \) correlation in the \( P_3 \) output plane, i.e. the autocorrelation of the signal.

With a filter matched to a reference \( r(x,y) \) we would have a cross correlation instead of the autocorrelation one.

The optical implementation of such a matched filter requires the availability of a complex transmittance proportional to \( P^* \).

This is a difficult problem since it is necessary to record on some medium a phase variation. The problem was first solved by Van der Lught in 1964 by using a reference wave to produce a carrier frequency which supports the phase variations as an amplitude modulation. This produces the so-called Fourier holograms, or Van der Lught filters, which are a photographic plate record of a Wiener spectrum plus the amplitude modulation of a carrier frequency, modulation proportional to the phase spectrum.

Since that day, many applications of this technique have been made, especially in image processing (character recognition, finger print recognition, etc...) and generally in pattern recognition.

3. First application to high energy experiments

Since flipped memories in MWPC in a number of chamber planes form an "image" which may be considered as a pattern, and that optical correlators are usable for pattern recognition, it is obvious that optical correlators may be used in high energy physics.

The principle is to search for a reference pattern (a 'good' event) in the signal pattern (a trigger).

Three kinds of optical processor may be conceivable:

- Fourier Processor which uses the spatial frequency information content of the patterns, rather than the direct spatial information. Optically seen, it is a simple Fourier Transform processor
- Matched Processor which uses correlation techniques to recognize the reference pattern, thus including the phase components.

- Synthetic Filtering Processor which uses optical correlation techniques, but with synthetic filters which allow coding of images and non optically feasible filters. This kind of processor is also hybrid by essence (both optical and electronic, digital and analog).

3.1. Fourier Processor

It is based on the idea that the frequency spectrum may be typical, and unique for a reference pattern, such as aligned points. In fact this is true as is shown in 15), but not very easy to use since the differences between the various patterns lead only to small variations on amplitude and frequency of a modulating factor of the central lobe of the spectrum.

One may understand this if one remarks that alignment of points regularly spaced along the x coordinate in a plane, means simply a constant variation of the y coordinate of the points. This yields a unique y spatial frequency which may well be detected.

3.2. Matched Processor 15)

We now use a full correlator with a Van der Lugt Filter, thus acting on both amplitude and phase. Figure 3 shows a summary of the results of such a processor (computer simulation). It is a plot of light intensity versus displacement of "sparks" (3 horizontal sparks are the reference which is checked for). It is clearly seen that a discrimination is possible, thus yielding an alignment criterion. However the difference between autocorrelation and cross correlation is not great and must be improved.

And many problem remain open:
- availability of real time electro-optical transducers
- matching of these transducers to high energy detectors
- complex filtering techniques which allow a better discrimination of the auto-correlation peak versus the cross correlation peaks.

4. Electrooptical transducers

The general purpose of an optical transducer is to implement a complex light modulation: amplitude and phase modulation of an incoming coherent and plane wave front. They are used as input transducers in the correlator of Fig. 2.

Output transducers are without problems: TV camera, photo sensitive arrays, CCD arrays, etc ...

4.1. Real-time transducers 10) 16) 17)

Real time transducers which are most interesting for our purpose may be divided in Electrically Addressable Light Modulators (EALM) and Optically Addressable Light Modulators (OALM). OALMs act as image converters, incoherent to coherent transducers, and EALMs are electrically written image transducers which are of highest importance.

Representative factors in the comparison of Spatial Light Modulators are:
- diffraction efficiency
- sensitivity
- erase mechanism
- storage capability
- lifetime
- type of modulation
- optical quality
- ease of fabrication
- complexity
- cycle time
- response time
- resolution and contrast
- type of addressing.

The four last characteristics are very important for the use in high energy physics experiments. For instance, consider the addressing scheme, which can be parallel (most interesting) or point by point. This means that an electronic scanned EALM which is sequentially loaded (by an electron beam for instance) belongs to the second class and needs a lot of time to load the image; thus degrades the performances by adding much input time.

Parallel addressing EALMs are more interesting because their loading time is reduced, for instance for matrix type EALMs.
4.2. Pockels effect transducers 18) 19) 20)

Pockels effect may be summarized as a variation of optical index $\Delta N$ produced in some crystals by an electric field. For uniform polarized light a phase variation is produced, according to:

$$\psi = \frac{2\pi L \Delta N}{\lambda}$$

$L$ : Length of path
$\lambda$ : Wave length of light

This means an elliptic polarisation of emerging light which can be analysed as an amplitude modulation with an output polarizer. Light transmission $T$ will therefore be

$$T = \sin \frac{\psi}{2} = \sin \frac{\pi L \Delta N}{\lambda}$$

$\Delta N$ proportional to the electric field.

Various EALMs make use of this effect: an electric charge pattern is deposited on one side of a Pockels effect crystal creating local refraction index variations. Thus, this crystal placed between crossed polarizers produces a light amplitude modulation. Reference 18 gives a full description of the process and a review of the various existing types.

4.3. Liquid Crystal Matrix (L.C.M.) 22) 23) 24)

A nematic liquid crystal with an homeotropic structure has the molecules oriented perpendicular to the plates (Fig. 4). Negative dielectric anisotropy molecules are also perpendicular to the plates, as is the optic axis. With an electric field applied across the plates, the molecules and the optic axis tend to align parallel to the plates, thus the transparency of the liquid crystal placed between crossed polarizers is changed.

Fig. 4 Liquid Crystal mode of operation
If one deposits a matrix type raster of conducting stripes on the plates, a LCM is created which produces the birefringence effect locally at the crossing of a vertical and an horizontal stripe. The addressing scheme is now semi-parallel: one full line of n elements may be addressed at one time.

The characteristics of our matrix are:
- 128 x 128 elements
- P. Methoxybenzilidene P.A.M. Butylaniline
- Indium oxide etched stripes: 280μm wide, 20μm spaced
- Cell thickness: 8μm
- Addressing voltages: word: 120 VRMS
  bit: 13 VRMS
- Optical efficiency in "on" state: 86 %
- Contrast ratio: 50 to 100/1
- Natural storage time: 2.5 msec
- Max data transfer rate
  without erasing: 10^6 bits/sec
  (with erasing techniques this rate may be increased by a factor 100).

It follows that for the best EALM, the Membrane Light Modulator, which is an electrically deformable membrane mirror, the data transfer rate lowers by a factor 10^{5/6} the possibilities of the optical computer (10^9 to 10^{15} bits/sec). Therefore the main problem of optical computers usability is the availability of high performance EALMs!

For the other transducers, one must remark that the best available is the LCM, with its low resolution, but semi parallel addressing scheme which permits data transfer rates of the order of 10^6/7 bits/sec without many problems.

To conclude: real time EALMs are nowadays available to research laboratories, with performances which degrade much the optimal performance but make the optical real time computer feasible.

4.4. Comparison of the input transducers

Table 1 gives a brief summary of the characteristics of various transducers.

<table>
<thead>
<tr>
<th>Type</th>
<th>Surface cm²</th>
<th>Resolution line: pairs/mm</th>
<th>Addressing scheme: semi-parallel</th>
<th>Memory: NO</th>
<th>Gray scale: research</th>
<th>Availabilty: stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Membrane</td>
<td>20</td>
<td>40</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Light Modulator</td>
<td>12</td>
<td>70</td>
<td>sequence 1 hour</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TITUS</td>
<td>2.25</td>
<td>80</td>
<td>sequence 2 hours</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROM</td>
<td>2</td>
<td>5 to 20</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.M.C.</td>
<td>5 to 20</td>
<td></td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Table 1 |

---

- 227 -
5. Filters

To understand the importance of filtering, let us see what may be done if any conceivable filter is feasible.

Suppose we want to make:
- code conversion
- coordinates transformation
- image transformation
- etc...

This is always the same problem, which may be expressed as follows:
"an output \( b(x, y) \) is wanted as an input \( a(x, y) \) is given".

By Fourier Transformation this leads to:

\[
A(u, v) \text{ given } ; \quad B(u, v) \text{ wanted.}
\]

Thus if the filter
\[
F(u, v) = \frac{B(u, v)}{A(u, v)}
\]
does exist

an optical correlator where \( F(u, v) \) is placed in the Frequency plane will produce at its output, the wanted \( b(x, y) \) when \( a(x, y) \) is given at the input.

Because \( a(x, y) \to A(u, v) \)
and \( A(u, v) \cdot F(u, v) = B(u, v) \)
and \( B(u, v) \to b(x, y) \)

Thus it is most important to create filters which are mathematically expressed.

But this is not always possible with optical methods. Therefore the use of synthetic filters.

5.1. Synthetic filters

A synthetic filter is a computer calculated complex transparency. The complex amplitude is represented by various methods able to record it on a medium. Then a photographic or electronic reduction is made which leads to the final filter. Of course the complex amplitude is generally obtained by Fourier Transforming an input "image" and modifying by computer the result (weighting of certain frequencies, derivation, etc...).

The main problem which arises is the recording of the complex amplitude on a transparent medium, because both pure amplitude and phase must be recorded.

Other problems are due to the sampling of the filter which is done, and to the digitization of amplitude and phase.

But the basic problem remains phase recording.

Three methods are known:
- Bleaching of a photographic film
  (not reliable, difficult)
- amplitude modulation by a phase
  (Vander Lugt Method)
- Lohmann Binary Holograms

5.1.1. Vander Lugt Method

The idea is to record a complex amplitude by amplitude modulation of a carrier frequency. Therefore the necessity to introduce a reference term which produces the carrier frequency.

In an unidimensional case, we have:
- reference \( R = \exp j \Phi(x) \)
- complex amplitude to be recorded
\[
F(x) = \rho(x) \exp j \Phi(x)
\]
- total intensity
\[
|aF(x) + R(x)|^2 = a^2|F|^2 + |R|^2
\]
\[
= a^2|F|^2 + 2a \rho(x) \cos[\Phi(x) - \Phi(x)]
\]

which makes apparent the amplitude modulation by \( \Phi \).

On a digital computer one computes
only \( C(x) + \rho(x) \cos[\Phi(x)] \)
where \( C(x) \) is a chosen constant.

This is mainly a digital version of the analog optical Vander Lugt Method.

5.1.2. Lohmann Binary Holograms

This more recent method is considered to be better and leads to many similar methods.

A main reason is that there is no more need of a linear recording medium, since only binary patterns are recorded (either transparent or dark positions).

Consider a dark plane, with regular spaced apertures (distance \( x_0 \)) in it, in both directions \( (x, y) \). A small displacement \( \epsilon \) of an aperture produces a phase difference
\[
\Phi = \frac{2\pi \epsilon}{x_0}
\]

Thus displacing an aperture is equivalent to a record of phase; and the size of the aperture records the amplitude by allowing more or less light to cross it.

This is a very good and simple method of recording of a complex amplitude.

However some errors exist:
- due to sampling of the filter

- 228 -
- due to amplitude quantification (digitization)
- due to phase quantification
- due to the limited size of the hologram.

Both references 25) 28) make an extensive study of the errors, to conclude that phase recording is more important than amplitude recording, but varies slower with the number of quantification steps 31). Special treatment of the computed filter, such as diffusion of the input signal leads also to better results.

To conclude, it is well possible to compute and record a mathematically expressed filter. The filter quality obtained is good enough for most purposes, and the methods involved in filter implementation are under control.

5.2. Special filters 28) 29) 30

Recently some progress has been made in the concept of special, optimal, filters for pattern recognition. But all these filters are only implementable with synthetic filters methods.

5.2.1. Principal components 29). To identify one out of 32 possible items, one can proceed with 32 analog measurements (correlations) or with 5 binary measurements (only 5 yes/no questions are needed to identify one out of 32 items). This remark lead Lohmann to the principal components concept.

If one considers \( N \) items \( U(n,m) \) (binary patterns with \( N \) cells), the principal components are solution of the \( kN \) equations with \( kN \) unknowns:

\[
\sum U(n,m) \cdot P(k,m) = b(k,n) \quad n = 1, N \quad m = 1, M
\]

which means that each \( n \) th item \( U(n,m) \) is compared with each \( k \) th principal component \( P(k,m) \) to form a coded response \( b(k,m) \) which is a binary word.

Figure 5 gives a trivial example.

![Table and Figure 5](image)

This brings some advantages:
- a binary measurement is made (easy, sure)
- a coded response is obtained, leading to code optimization
- a few number of measurements has to be made \( (k \) measurements for \( 2^k = N \) items) meaning some degree of filter multiplexing.
- and even a further filter multiplexing may be made (spatial filter multiplex) to reduce the number of measurements to one.

5.2.2. F.O.U.T. 28) 30). This further filter multiplexing and a modification of the definition equations into true correlation leads Fleuret to his definition of the Filtre Optimisé Unique de Transcodage. To the advantages of the Lohmann filter, one becomes a better resolution and a unique filter. Table 2 shows 28) the improvement over the matched filter produced. The result is impressive and means that FOUT may solve the discrimination problem encountered in § 3.2.
Table 2. Relative intensities of auto/cross correlation peaks vs different filters.

6. Application to high energy physics

6.1. Characteristics of the application

In high energy physics experiments, special purpose processors are needed to find points (solve multiple hits ambiguity) and find tracks (associate points to tracks).

The main characteristics of these processors are:
- for the point finding processor:
  the problem is a coordinate change problem, to solve multiple point association ambiguity (see filtering with synthetic filters...)
- for the track finding processor,
  the track:
    . is sampled
    . is approximative (missing points, spurious points, displaced points)
    . shape is varied (linear, curved)
    . is multiple (5/10...)
    . classification is difficult (there exists no reference or too many)
    . finding needs a high resolution and speed.

On the other hand, optical methods, are based upon correlation / filtering techniques, which all need the knowledge of a reference or of noise (antireference...). Therefore problems due to a lack of reference or to a too broad definition of reference arise.

A solution of these problems may be searched in the special filters like FOUT which are heavily multiplexed, and therefore accept many references. They also discriminate easily between auto and cross correlation of items which are very similar, thus giving hope that a high resolution is possible.

And also, optical transducers have speed and resolution limits which may be analysed.

However one must consider that optical computing is a basic parallel process, which means that multiple tracks are accommodated as a single track. Thus processing time is independent of number of tracks / points !

5.2. Detectors / Transducers matching

To overcome speed and resolution problems and to achieve overall efficiency the match between high energy detectors and EALM must be analysed. The guidelines are:
- stay parallel as long as possible
- thus convert to sequential only one time
- suppress the numerous coding / decoding and input / output time, in the chain of processors actually used (chamber read out, cluster coding, preprocessing, code conversion, acquisition...).

The characteristics of the detectors seen from an optical computer's point of view are very different for MWPC or drift chambers.

In MWPC the data is digital, location coded (mapping representation), parallel, memorized, available within a few microseconds in large quantities (> 10^3).
In drift chambers the data exists in a complex manner: digital, location coded and instantaneous for the wire number; digital, binary coded, delayed for the drift time. Smaller quantities exists, available in a few 10 microseconds. Data is obtained by a sequential conversion (drift time).

Therefore it is clear that the MWFC data structure is matched to a parallel addressing display such as a LCM, and to a mapping kind of display.

On the other hand, the drift chamber data structure is not well matched to optical transducers with parallel addressing scheme. And due to the digital coded data structure, a numerical kind of display would be better suited.

This means that MWFC are best suited to match an EALM without any electronics modification; and that drift chambers need modification of the electronics equipment to adapt to EALMs (until performances are degraded).

This leads us to ask an important question: how should one represent optically an event?

A chamber is divided in N x M cells; thus different possibilities exist:
a) to each cell corresponds a unique position in the input image (mapping kind of display)
b) to each hit corresponds a code in the input image (numerical kind of display)
c) one peculiar code could be the coded cell coordinates...
d) or another code, more complex, could be the hologram of the cell position, or of the cell's coordinates...

a), b), and c) are conventional representations, with the difference of a higher information density for b), c), d) makes me dream about the possibilities opened if suitable transducers would exist!

6.3. Possible application

Our laboratory is in the course of mounting a LCM (128 x 128) on our optical computer, with a FOUT filter in the frequency plane, to make pattern recognition measurements. An application of this processor could be as a special purpose processor for track finding.

The 128 x 128 LCM in a mapping mode of display can accommodate 16 chambers projections with a 1 out of 1024 resolution.

A possible FOUT uses 64 reference tracks, up to 1024 may be accommodated if one uses up to 10 principals components.

Processing will take a time t:

\[ t = t_{\text{load LCM}} + t_{\text{compute}} + t_{\text{output}} \]

With

\[ t_{\text{load LCM}} = 10 \text{ microseconds per 128 wires} \]
\[ t_{\text{computing}} = \text{negligible} \]
\[ t_{\text{output}} = 10 \text{ to 20 microseconds due to the parallel to serial conversion needed by the following electronics.} \]

(at the output, one finds the code of the recognized track).

A price estimation of the whole gives 70 KSF.

These expected performances will be checked experimentally.

Remark that a LCM 512 x 512 is available in a prototype stage, which would mean a loading time divided by four.

A greater resolution / speed improvement is expected with the numerical kind of display which would allow 8 numbers of 16 digits accuracy to be loaded and processed simultaneously within 10 microseconds. But this is at a research stage.

7. Conclusions

These lectures made an attempt to explore a new way of computing with non classical electronics means, with the purpose to overcome the limitation of special electronic processors. The aim was not to give a finished practical implementation immediately usable in high energy physics, but to show how this could be done, at what cost, and which performances could be expected. But remember, the topic is still in a full research stage and only for a few steps is a development stage.
8. Acknowledgements

I want to thank my colleagues Gresser, Schultz, Hueber, Perrin, Mure and Wolff for their contributions and collaboration. I must thank the organizing committee for their kind invitation and especially Dr C. Verkerk. Many thanks also to the secretaries and typists for their hard labour.

REFERENCES

1. C. Verkerk ; "Special Purpose Processors"
Lectures presented at the 3rd CERN School of Computing, Bergen, Norway
CERN - DATA HANDLING DIVISION DD/74/27
September 1974

2. J.A. Rudolph, L.C. Fulmer, W.C. Mailander ;
"The coming age of the associative processor"
Electronics 15th February 1971

3. L.C. Hoble ; "Parallel processor systems.
Technology and applications"
Spartan books - N.Y. 1970

4. E.O. Brigham ; "The Fast Fourier Transform"
Prentice Hall Inc. Englewood Cliffs N.J.
1974

5. H.H. Hopkins ; "On the diffraction theory of optical images"

vol. IT-2 , pp 56-65, June 1956

7. A. Vander Lucht ; "Coherent Optical Processing"

8. K. Preston Jr. ; "Coherent optical computers"
Mac Graw Hill N. Y. 1972

9. M. Françon ; "Optique, Formation et Traitement des images".
Masson, Paris 1970

10. Y.E. Nesterikhin, G.W. Stroke,
W.E. Kock ; "Optical Information Processing"
Plenum Press, N.Y. 1976


12. G.W. Stroke ; "An introduction to Coherent Optics and Holography"
Academic Press, N.Y. 1966

13. P. Zanella ; "Developments in data collection and processing for electronic experiments" CERN DD/73/May 14th 1973

14. J. Soleman, T.A. Numamacker ;
"Pattern Recognition Processors and its application to straight line reconstruction of spark chambers data".
Nucl. Instr. Meth. 107, 15, 1973

15. M. Perrin, G. Metzger ;


74CH0862 - 3C


20. P. Dumont, D. Bossier ; "Photitius : un relais optique convertisseur d'images"
Acta Electronica, 18, 2, 1975, 131-141


25. A. Bruel, J.C. Cazaux; "Hologrammes générés par calculateurs et applications informatiques". Thèses d'état No 624 et 625, Université Paul Sabatier TOULOUSE 1974


27. B.R. Brown; "Computer Syntheses of holograms and spatial filters". see ref 16) p 215


LIST OF PARTICIPANTS

LECTURERS

BENDANZ, R. CYFRONET, INR, Swierk, Poland
BELSHE, D. University of Oslo, Norway
BLOCH, T. CERN, Geneva, Switzerland
DRIJARD, D. CERN, Geneva, Switzerland
GABATHULER, E. CERN, Geneva, Switzerland
KUNZ, P. Stanford University, U.S.A.

METZLER, G. Institut des Sciences Exactes et Appliquées, Mulhouse, France
MONRAD-KROHN, L. MYCRON, Oslo, Norway
SENDALL, D.M. CERN, Geneva, Switzerland
Treichnow, D. University of Michigan, Ann Arbor, U.S.A.
VENCZER, C. CERN, Geneva, Switzerland
ZACHAROV, V. University of London, U.K.
ZELAZNY, R. INR, Swierk, Poland

SEMINAR SPEAKERS

GZIK, Z. CYFRONET, INR, Swierk, Poland
GZINSKI, W. CYFRONET, INR, Swierk, Poland
MACHURA, M. CYFRONET, INR, Swierk, Poland
PIERKAT, J. Institute of Experimental Physics, Wroclaw, Poland

---------

ANDERSSON, G. Department of Nuclear Physics, Lund, Sweden
BARASIK, Z.A. Instytut Badan Jadrowych, Warsaw, Poland
BALEY, R. Rutherford Laboratory, Chilton, U.K.
BEREZNOI, V. ITEP, Serpukhov, USSR
BINGINI, M.E. INFN, Frascati, Italy
BILHAUT, R. LAL, Orsay, France
BLOCH, C. CERN, Geneva, Switzerland
BOGDANOVA, N. Institute of Nuclear Research, Sofia, Bulgaria
BRIGHT, D. Daresbury Laboratory, Warrington, U.K.
BUGGE, L. University of Oslo, Norway and CERN, Geneva, Switzerland
CAMELIERI, T. DESY, Hamburg, Federal Republic of Germany
CERIOPOLINI, L. Istituto Matematica Applicata, Bologna, Italy
CONWAY, J. University College London, U.K.
COUFERES, C. CERN - Saclay, Gif-sur-Yvette, France
CRANFIELD, R. University College London, U.K.
DUCORS, A. LAL, Orsay, France
DURIEU, L. CERN, Geneva, Switzerland
DZIECIAZSZEK, J. CYFRONET, INR, Swierk, Poland
ESKESKY, K. University of Mining & Metallurgy, Cracow, Poland
FEYER, J.-M. Institut de Physique, Louvain-la-Neuve, Belgium
FINLEY, I. University of Cambridge, U.K.
FISCHER, T. Fakultät für Physik, Freiburg, Federal Republic of Germany
FISHER, S. Rutherford Laboratory, Chilton, U.K.
FRUTIMTH, R. Institut für Hochenergiephysik, Vienna, Austria
GAJEMSKI, J. University of Warsaw, Poland
GEE, N. Rutherford Laboratory, Chilton, U.K.
GIERLKA, J. Jagellonian University, Cracow, Poland
GOEKE, W. Fakultät für Physik, Freiburg, Federal Republic of Germany
GOTOWSKI, Z. CYFRONET, INR, Swierk, Poland
HAGLUND, R. University of Lund, Sweden
HARVEY, J. The University, Southampton, U.K.
HASEMANN, H. DESY, Hamburg, Federal Republic of Germany
HEMERY, J.-Y. CERN, Geneva, Switzerland
KIRKSBOM, K. University of Oslo, Norway and CERN, Geneva, Switzerland
KISIELINSKI, B. Institute of Nuclear Physics, Cracow, Poland
KISIELINSKI, M. Institute of Nuclear Research, Swierk, Poland
KLOK, P. Nijmegen University, Netherlands
KOKKINAS, P. NRC Demokritos, Athens, Greece
KRIZ, V. Czechoslovak Academy of Sciences, Prague, CSSR
KUZENKO, V. IHEF, Serpuchoz, USSR
LA HARGE, J.-P. CISTI, Gif-sur-Yvette, France
LAGHAKANGAS, R. University of Helsinki, Finland
LITSCHER, M. CERN, Geneva, Switzerland
LUTY, J. Institute of Nuclear Research, Swierk, Poland
MACALLISTER, J. University of Oxford, U.K.
MADEYSKI, B. Institute of Nuclear Physics, Cracow, Poland
MAROSA, D. CYFRONET, INR, Swierk, Poland
MATVEEV, V. ITEP, Moscow, USSR
MEIJER, B. IKO, Amsterdam, Netherlands
MITUKIEWICZ, E. University of Warsaw, Poland
MUR, M. CEA Saclay, Gif-sur-Yvette, France
NOZ, D. DESY, Hamburg, Federal Republic of Germany
PALKA, H. Institute of Nuclear Physics, Cracow, Poland
PEACH, K. University of Edinburgh, U.K.
PIACENTINO, G. University of Padua, Italy
PIMIA, M. University of Helsinki, Finland
PONATONSKI, M. CYFRONET, INR, Swierk, Poland
POPELSKI, W. Institute of Nuclear Research, Swierk, Poland
RASZKOWSKI, P. University of Warsaw, Poland
REGLER, M. Institut für Hochenergiephysik, Vienna, Austria
RUCHNOSKA-LUKASIAK, E. University of Warsaw, Poland
RUMMER, W. University of Bonn, Federal Republic of Germany
SAUERLAND, J. Max-Planck-Institut, Munich, Federal Rep. of Germany
SCOTT, I. The University, Glasgow, U.K.
SEIS, I. CERN, Geneva, Switzerland
SERIO, M. INFN, Frascati, Italy
SHAVE, A. CERN, Geneva, Switzerland
SKLENAR, I. Czechoslovak Academy of Sciences, Prague, CSSR
SLUBOWSKI, K. University of Warsaw, Poland
THOMAUSKE, B. Fakultät für Physik, Freiburg, Fed.Rep. of Germany
VOGT, H. Inst. für Hochenergiephysik, Zeuthen, German Democratic Rep.
WALCZAK, R. University of Warsaw, Poland
WALTERMANN, G. CERN, Geneva, Switzerland
WATTS, R. CERN, Geneva, Switzerland
WEINAR, R. SCO CYFRONET, Cracow, Poland
WILLE, M. Inst. für Hochenergiephysik, Zeuthen, German Democratic Rep.
WOJCIECHOWICZ, B. CYFRONET, INR, Swierk, Poland
WOLNY, J. SCO CYFRONET, Cracow, Poland
ZALEWSKI, J. Institute of Nuclear Research, Warsaw, Poland
ZOLTOWSKI, T. Institute of Nuclear Research, Warsaw, Poland