Design of a demonstrating system for use in PRX equipment courses

Akhbeis, T.

Award date:
1979
Design of a demonstrating system
for use in PRX Equipment Courses

afstudeerproject van: T. Akhbeis
verricht in de vakgroep EB
in de periode aug. '78 - aug. '79

in opdracht van prof. Ir. A. Heetman
en Ir. J.A. Samwel (Director Philips'
International Telecommunication
Training Centre - Hilversum)

afstudeerhoogleraar: Prof. Ir. Heetman
coaching: Ir. H. Kemper
## CONTENTS

<table>
<thead>
<tr>
<th>Subject</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>ii</td>
</tr>
<tr>
<td>I.1. Processor controlled telephony</td>
<td>I.1.</td>
</tr>
<tr>
<td>I.2. PRX</td>
<td>I.2.</td>
</tr>
<tr>
<td>II.1. Hardware Configuration</td>
<td>II.1.</td>
</tr>
<tr>
<td>II.2. Input/output</td>
<td>II.2.</td>
</tr>
<tr>
<td>II.3. Programming the PIO</td>
<td>II.3.</td>
</tr>
<tr>
<td>II.4. The Z-80 CTC</td>
<td>II.8.</td>
</tr>
<tr>
<td>II.5. Structure of the Channel Logic</td>
<td>II.9.</td>
</tr>
<tr>
<td>II.6. CTC Operating Modes</td>
<td>II.10.</td>
</tr>
<tr>
<td>II.7. CTC Programming</td>
<td>II.11.</td>
</tr>
<tr>
<td>III. Demoprog</td>
<td>III.1.</td>
</tr>
<tr>
<td>III.1 Definition of The DEMOPROG User's Language</td>
<td>III.2.</td>
</tr>
<tr>
<td>IV.2. Scan Source Module</td>
<td>IV.7.</td>
</tr>
</tbody>
</table>

### Appendix A.

1. Conclusion
   3.
   4. References
Subject: Design of a demonstration system for use in PRX Equipment Courses.

The PRX is a computer controlled telephone switching system. There is a need for simulation of several equipment functions for teaching purposes. I chose to design a system to demonstrate the salient features of the central processor and input/output device. Furthermore, the system design is flexible so that it can easily be used for other types of equipment.

The demonstration equipment comprises a Z80 microprocessor which controls the demonstration process and an interface to select and send data to a course oriented panel. A VDU device enables the lecturer to manipulate the process and select depth levels.

To give the system optimum flexibility, I have defined and implemented an application oriented language called DEMOPROG, which is compiled using a compiler written in high level program language PLZ. The user prepares lessons in the form of Demoprog statements. Each statement defines a certain action at the panel. Sequences of actions can, for instance, demonstrate PRX addressing techniques, arithmetic operations, and suchlike.
Introduction

Understanding.

One way of beginning an inquiry into the nature of understanding is to look at the ways in which the words "understand" and "understanding" are used in everyday speech, the object being to see if it is possible to make general comments about the notion which will serve to provide hints and clues for educators in their role as "teachers of understanding".

Suppose someone says, "Piet doesn't understand how to get the computer to work", or "He is the only one who understands how to solve quadratic equation's. In these examples "understand" has the sense of "Know what to do" - "Piet doesn't know what to do to get the computer to work" - and, further, seems to be significantly different from another sense of the word in which the notions of explanation and theoretical rationale are well to the fore. For example, "He doesn't understand the general theory of relativity". Failure to understand, in the sense of not knowing what to do, can be remedied by giving a simple instruction or set of instructions- "You just stick this wire in there", or, in the case of a child who doesn't understand how to "divide fractions", "You just change the divide sign to multiply etc." But the giving of the instruction or instructions, while it may now enable the child to do the division- he now understands what to do - in no way enables him to understand why he does what he does do, just as sticking the wire "in there" in no way enables me to understand why these results in the computer are working now.

We hit the main reason of designing the demonstrating system which shall be used to explain and refine functioning principles of the PRX "Processor controlled Reed eXchange". Experience proofs that one who spends a certain period working on systems based on analog signals, will have difficulties to digest and understand systems founded on digital principles. Lecturers are facing this problem often in the Philips' training Centre "P.I.T.T.C.". One needs a didactic system to pass this barrier towards accumulating the necessary technical knowledge.

Accumulation of Technical Knowledge.

Scientific progress affects the entire social, economic, and technology framework, to which the problem of educational orientation is related. It furthermore directly affects those problems which face individuals during the periods when they are acquiring knowledge or bringing their knowledge up-to-date, and also directly influences educational institutions in which individuals are taught.
Scientific progress therefore has specific effects on orientation, in particular those relating to continuous learning or training, (permanent or recurrent education).

Knowledge can be roughly split into two types. The first type is that of technical knowledge. In this field, knowledge relating to a given sector is generally and necessarily ordered, hierarchical, and it can be accumulated. A particular problem can only be studied after first learning a certain amount of information, laws, and techniques, each of which in their turn demand still more preliminary elementary knowledge.

The second type of knowledge is found particularly in the literary fields, and is not necessarily hierarchical, or at least not to the same extent or in the same way: we can talk about a literary or philosophical work in a way which is more or less apt, more or less original, more of less profound, but none of these expressions implies the necessary preliminary acquisition of an ordered and accumulated body of knowledge. These two types of knowledge differ not only in the way in which they are acquired, they also differ in the way in which they become out of date. No doubt our understanding of a philosophical or literary work evolves over a period of time. But such an evolution can hardly be compared to the kind of knowledge involved in scientific progress. Difficulties which may be experienced in taking up literary or philosophical studies after a certain lapse of time, are not comparable of those experienced by someone doing the same thing in scientific fields.

Finally also the criteria used to assess attainment are not the same for the two types of knowledge. In the scientific field, predictions can be confirmed or invalidated by facts which can be clearly observed, as can the success or failure of applications of scientific knowledge. This is not the case with literary or philosophical knowledge.

Individual acquisition of some kind of knowledge appears to be a necessity. The computer capable of supplying all the information we need without our having to "know" ourselves is, and will remain, a myth. In fact, at the documentation level alone, to feed a computer with questions presupposes the ability to manipulate a set of descriptive characteristics in the field of knowledge in question, and to manipulate such a set of characteristics, always complex in structure requires preliminary training and apprenticeship. Furthermore, and above all, the myth of the perfect computer is based on a confusion between documentation and knowledge. The use of computers in documentation cannot eliminate the fundamental distinction between these two ideas. A Frenchman may have a French-English dictionary and an English grammar and still be unable to read an English text. Undeniably, the rapid expansion of knowledge makes it indispensable to teach "know-how"; education as a process of teaching people how to learn. But is "know-how" possible without basic knowledge? Is it possible to teach how to learn without first accumulating knowledge?
We must be aware that continuous training, already necessary in many fields, will be more and more urgent requirements in the future.

Continuous education can have the aim of enabling a worker to change his occupation during his working life. To what extent can this be achieved?

It would be a mistake to think that the speed of scientific progress creates a greater equality of individual opportunity to acquire new knowledge in a given branch. In fact an individual educated in a given field today, has the best chance of mastering new techniques which may be developed in the future in his particular field. Despite facilities for continuous learning, it will probably remain difficult to become highly qualified in a field outside the with which one is familiar, and the higher the qualification required, the greater the difficulty.

We come now to the essence of our case, another function of continuous training should be enable an individual who has to change the form of his function to a modern one. One aspect of an orientation system would be to provide guidance and assistance in such cases. The speed with which knowledge and techniques become out of date, is indeed one of the main difficulties in designing systems of continuous training and guidance. A qualification obtained in the past is not like a railway ticket which enables you to rejoin the train at the station where you left it. The ticket may only be valid for a short period. Of course, readaptation courses could be provided, enabling people to bring themselves up-to-date in their various fields. The main problem is to decide on the duration of such refresher courses, and to estimate the investments required in terms of individual effort and financing. This difficulty will grow as educational systems themselves, become less and less inert and traditionally minded, with rapid changes in method and context.

Building up a demonstrating system to explain the principles of the PRX, has an embedded aim; for such trainees who have been building up their experience in the conventional central telephony stations, and suddenly have to maintain a completely different telephone station working in digital principles, controlled by a computer. Being accustomed to test analog signal, demands certain apparatuses, while testing a digital signal requires a different technical science and philosophy. It is hard for human beings to change drastically from one way of thinking, imagining, to another. The essence of designing this system, is to be used as a tool for the teacher during his lessons. The system is flexible, it can be adapted to explain the working methods of the computer.
Section I.

I.1. Processor controlled telephony.

The history of telephony recognizes revolutionary principles based on the new developed technical science called the digital communication techniques. For a long time switching systems used to be conceived around a few ingenious electro-mechanical devices, such as relays, selectors and crossbar switches.

The study of switching systems was based on the evaluation of block diagrams and on the operation of the individual units. Nowadays one uses concepts such as switching network, directly and indirectly controlled systems, systems having common network control and storage requirements of systems, without bothering, whether certain functions are realized by hardware and wiring or by software and electrically changeable memory contents. Hardware needs no longer to be designed for carrying out certain specific functions per call, but can be generalized to the point where it can be used, in combination with the coded information stored in the memories, in the decision making process of each call, in other words, most functions can be realized as software modules.

Increasing the traffic volume of the telephone network, has led to new traffic measurement and network management techniques, which are to enable better use and control of the network as a whole and, in this way, guarantee an overall high grade of service. The exchange has to be approached as integral with the network. Remote control and integration of all operational and administrative network functions are dominating factors in the architecture of the new generation of telephone exchanges, such as the PRX family.
1.2. PRX.

The Philips semi-electronic stored-program controlled telephone system — the characters stand for Processor-controlled Reed eXchange — is the result of long experience gained in telephone switching and in the application of electronics and computers in communication technology.

The PRX is designed for use in public telephone networks, its functions ranging from satellite to transit operation. It may also be used in multi-exchange networks or as a combined local/trunk exchange.

The principle of the "stored-program control" is the most important milestone in telephone switching development. According to the principle, all logical control functions and data are concentrated in the central processor, allowing a flexibility and variety in switching facilities which, if ever, could be achieved in electromechanical systems only at great expenditure. The software nature of the logics permits easy accessibility and changeability of functions at data via internal and also external data channels. Functions such as blocking and unblocking subscribers, changing routing patterns, reading out metering data and other more complicated functions can be initiated in a rather simple way, by commands given via data links.

1.3. Parts of The PRX.

As shown in the general block diagram of Fig.II.1, the system comprises three principal sections:

- Switching Network
- Central Control Unit
- Interface Equipment.
I.3.1. Switching Network.

The heart of the switching network is a trunklink network. It contains all unit and switches which are required to set up local and trunk connections. The switches are in the form of reed relays.

I.3.2. Central Control Unit.

All functions to be performed in the switching network for the establishment, supervision and release of connections, are fully controlled by the Central Control Unit (CCU, see Fig. I.1), in other words, it gives commands, receives information and stores data in the memory, makes calculations with its arithmetic unit. Also it updates all connections, initiates the operation of the various units, shortly, it controls the operation of the PRX.

The CCU consists roughly of the following parts,

a. Processor Core memory,

b. Central Processing Unit, (CPU)

c. Input/Output unit, (IOU)

d. Alarm and Switching Unit, (ASU)

and of course many other parts which I have neglected for the sake of the subject I am handling.

I.3.2a. The memory contains different kinds of information:

. The program section containing instructions for the processor,

. Data store for subscribers' numbers, subscribers' meters, trunkline numbers, etc.

. Working area for temporary storage of data.

I.3.2b. The Central Processor is the TCP 18 unit.

It is a binary single-address machine with a word and instruction length of 16 bits: it has six program-accessible registers. Address modification and indirect-addressing is specified per instruction, and full addressability of the memory is obtained by relocating the 8-bit address field of the instruction, using the contents of an 18-bit relocation register, which is loaded per program module.
Interrupts, autonomous transports and scan results received via the I/O unit have hardware priority over instructions. A program hesitation technique is used, in which the processor control decides after each completed instruction which operation will be executed next.

The data channels (see Fig. 1.1) have memory access priority over the central processor, and operates using the cycle-stealing principle.

A crystal-controlled processor clock provides 16 timeslots with an interval of 110 nanoseconds. Moreover, each central processor unit is equipped with a real-time clock which is used to schedule the autonomous scan procedure in accordance with the real-time requirements of the telephone equipment, to measure the recognition time of signalling systems, to guard the operation time of slow subsystems such as markers and drivers, etc. The basic clock period is 12.5 milliseconds, which can be halved or doubled by means of strapping.

I.3.2c. The input/output unit permits asynchronous co-operation between the central processor unit and the interface equipment, and is to some extent independent of its associated central processor unit. It contains six registers, three of which are used to store data required for the autonomous scan procedure which determines the current status of the exchange. The test points in subscriber line circuits and junctors are arranged in groups of 16 which are scanned in a parallel mode. The scanning is not performed by program to prevent the processor loading from becoming very high to yield only a small amount of status changes. The procedure is a program-initiated hardware subroutine which autonomously increments the current memory address and seizes the appropriate registers of the central processor to perform, compare, load and store functions at the moment the new status of the test points in the switching network is received.

At the end of each autonomous scan cycle, the collected logical differences are processed by program.
Approximately the same procedure is used in the case of autonomous data transports via the control channel.

A test-access connection between the dual processors is provided, through which a faulty processor can be investigated by means of diagnostic programs in the on-line machine.

I .3.2d. The two processors operate instruction-synchronously in a dual mode, and are continuous compared for proper performance. The configuration is governed automatically by program-controlled Alarm and Switch-over Unit.
Fig 3.1
Section II.

II.1 Hardware Configuration

The design of any digital system can be broken down into a succession of manageable steps. This is an essential characteristic of the design process. And, to the extent that the designer consciously subdivides the design task into these steps, he is able to bring a variety of potent design tools to bear on the problem. Some of the steps are:

1. Assessing how the system inputs and outputs constrain the system design
2. Deciding how much time is available to carry out the required data manipulations. This determines whether specific algorithms will be implemented in parallel (for speed) or serially (for economy). It also determines whether a specific logic line is fast enough to do the job.
3. Studying the operations involved in order to develop an algorithm which is particularly well suited to the problem.
4. Specifying a system structure considering the above mentioned points, we shall describe during this selection the parts of the system. (see fig. II.1.)

1. Z80 microcomputer system the MCZ 1/20
2. Interface between the microcomputer and the output
3. A lamppanel simulating the necessary course, in this case the panel describes the Basic Computer Technique.

Let us explain those parts point by point:

1. The Z80 microcomputer is a software developing system supplied with 32kB byte memory. The system is controlled with the R10 operating system. The user can develop his software with the assembler or the high programming language the PLZ.

2. The interface has the task to save the data instantly which send by the microprocessor, till the right unit has been selected. (see fig. II.2.)
This will be conceived by the latches type 8212 of Intel, attached to the lamp drivers.

3. The panel consists of 256 lamps, to simulate the necessary functions, needed for the BCT course. (see fig. II.3.)

II.2 Input/Output

In this section we need to apply PLZ/SYS parameter passing given in the interrupt routine. To be more accurate the interrupt routine needs to call sequentially the Panel output procedure, called PANIO, to send two actual parameters to the output. By means of this call two actual parameters are passing to the panel output. The parameters are the pattern and the group.

The procedure PANIO had to be written in PLZ/ASM, due to its hardware facility. In the PANIO procedure we are selecting one PIO (programmable input output) of the four which are mounted in the IOB (input output board). Further detail can be achieved in the next section.

Let us discuss now how the passing parameters procedures are applied. Each PLZ procedure has access to both static data (declared Global or Internal), allocated in what is called the Activation Record (AREC) for each invocation of a procedure. Static data is simply accessed at absolute memory address. Each AREC is allocated on the stack where the current Top of Stack word is pointed to by the register SP. Consequently, new AREC's are allocated to memory addresses which are decreasing as the stack grows. In addition, the register IX always points to a fixed position within the AREC and is used as a base to access local variables and parameters which are all known to be fixed displacements from the IX register.
The format of the AREC consists of the following sections:

1. An "In Parameters" passed from the caller of the procedure.

2. A Mark-Stack Record (MREC) described below.

The MREC is simply two fields: the first contains the return address of the calling procedure. The second field is the value of the calling procedure's IX register, which is to be restored upon return to reestablish the correct environment for the calling procedure. The IX point to the low memory address of the field which contains the calling procedure's IX value during the active procedure's lifetime.

The following diagram clarifies the format of an AREC and indicates the in parameters.

The procedure PAN10 can be found at the end of this section.

II.3 Programming the PIO (Programmable Input Output)

The PIO which we are using is one of the four PIO's mounted on the IOB Board. The IOB interfaces directly to the MCS system data bus, with buffer logic.

The PIO is programmed by writing data to the PIO control ports as a series of commands.
The command formats are standardised as follows:

The operating mode of each I/O port can be selected by a control byte with the following format, in our case we have selected mode 0.

\[
\begin{array}{cccccc}
D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 \\
0 & 0 & x & x & 1 & 1 & 1 & 1 \\
\end{array} \equiv \ phi \ H
\]

[Diagram]

To select one of the PIO mounted in the IOB board, we need to send the following sequence of bytes.

\[
\begin{array}{cccccc}
D_7 & D_6 & D_5 & D_4 & D_3 & D_2 & D_1 & D_0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

- 0 0 selects data for Port A
- 0 1 selects data for Port B
- 1 0 selects control for Port A
- 1 1 selects control for Port B.

[Diagram]

This leads to the following code:

- \text{FC H ; selecting port A for the pattern}
- \text{FD H ; selecting port B for the group}
- \text{FE H ; selecting port A address}
- \text{FF H ; selecting port B address.}
**PANIO Procedure**  (written in PLZ/ASM)

**patout module**

**Global**

**panio procedure**

**entry**

```
Push IX ; save caller's IX and establish own
LD IX, 0
ADD IX, sp
LD C, $FEH ; port Address of PIO 0
LD A, $FH ; select operating mode (out put)
Out (C), A
LD C, $FFH ; program port B of PIO 0
LD A, $FH
Out (C), A
LD C, $FCH ; select port A for pattern
LD A, (IX+6); get 1st byte
Out (C), A ; send pattern out
LD C, $FDH ; select port B for group
LD A, (IX+4); get 2nd byte
Out (C), A ; send group out
pop IX ; restore caller's IX
pop HL ; get return address
pop DE ; deallocate in parameters
Jp (HL)
```

**end panio**

End patout
**Fig. II.1.**  
System parts

**Fig. II.2.**  
Interface
Most microcomputer applications require some kind of counting or timing whether it is for the provision of time delays in simulating a large system, 'time out's' in a communication controller, or counting the number of cans which pass a photoelectric cell. We shall use the CTC as a timer to enable a sequential interrupt to the main interpreter program, during running, in order to achieve the 3 blinking modes i.e. Mode Fast (8 Hz), Mode Medium (2 Hz), Mode Slowly ($\frac{1}{2}$ Hz).

The CTC provides the following features:

- Operation of each channel in either timer or counter mode;
- Triggering by either system clock or external asynchronous clock;
- Zero Count (Timeout output on channels 1 and 2);
- A Readable down counter indicating the number of counts to zero;
- Programmable, nested interrupts (mode 2) on all four channels in both counting and timing modes.

The CTC is programmed by writing to the channel control register, but before we start discussing how to program then CTC, we ought
to know the IC (chip) itself.

Fig. II.4 shows the construction of this chip, it contains of a bus interface to the Z-80 CPU, Internal Control Logic, four sets of Counter/Timer Channel Logic, and Interrupt Control Logic. The CTC has the capability of generating a unique interrupt vector for each separate channel (for automatic vectoring to an interrupt service routine). The 4 channels can be connected into four contiguous slots in the standard Z80 priority chain with channel number has the highest priority. The CPU bus interface logic allows the CTC device to interface directly to the CPU with no other external logic.

Now we get to know the parts of the Z-80 CTC, let us direct our attention to the construction of the channel.

II.5 Structure of the Channel Logic.

Fig. II.5 shows the channel logic structure which is composed of 2 registers, 2 counters and control logic.
The registers are an 8-bit Time Constant Register and an 8-bit Channel Control Register. The counters are an 8-bit CPU-readable Down Counter and an 8-bit Prescaler.

11.6 C T C. Operating Modes.

Conceptually, the CTC may be viewed as one of four channels, each of which functions as in the diagram below:

By loading appropriate information into the Control Register, the channel's operating mode is determined. These are two modes, Counter / Timer mode.

For our application we shall use the Timer Mode, where the system clock is used to trigger the down counter. The clock is fed through a prescaler, which can be either 16 or 256, and hence the period of the timer is:

\[ t_c \times P \times T_c \]

- \( t_c \) is the system clock period
- \( P \) is the prescale (16 or 256)
- \( T_c \) is the initial value of the contents of the down counter.

The timing sequence can be triggered-off either by a pulse on the appropriate CCK/TRG line or under software control via the control register.
Again, when Zero is reached ZC/TO is output high and interrupt is available. When Zero Timeout is reached, the down counter register will automatically be reloaded and, if timing is under software control, the time period will again be counted down. If the ZC/TO output from timer is used as a trigger for a down counter the actual length of the timer can be increased to

\[ 256 \times t_c \times P \times Tc \]

which is about 6.7 secs. for a Z-80, and 4.2 secs. for a Z-80A.

II.7 CTC programming

II.7.1 Channel Control Word.

Programming the CTC as a timer can be achieved by loading the Channel Control Word as follows:

\[ D_1 \ D_6 \ D_5 \ D_4 \ D_3 \ D_2 \ D_1 \ D_0 \]

\[ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \]

\[ \equiv \ \text{\$B5H} \]

The explanation of such a byte will be explained as follows:

Setting bit Zero to unity indicates a channel control command, and this is to be stored in the channel control register.

The first three bits of the control register are

\[ D_2 \ D_1 \ D_0 \]

\[ 1 \ 0 \ 1 \]

channel stop counting until Tc loaded Time Constant follows.

Setting the second bit, forces us to provide a Time Constant as the next byte written to the selected channel. The Time Constant register also stores the value to be 'counted down', all values should be programmed in 1's complement form, i.e. \( P \) is equivalent to a Time Constant of 256, (see later calculating the Time Constant).
These next three bits are used in time mode only.
Bit 3 selects the cycle on which the timer operation starts and depends upon the setting of bit 2 as shown in the table below:

<table>
<thead>
<tr>
<th>bit 3</th>
<th>bit 2</th>
<th>interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>start (or restart) operation at next machine cycle</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>start at next machine cycle, following loading of down counter</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>start operation after trigger makes specified transition</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>start operation after down counter loaded and trigger has made specified transition</td>
</tr>
</tbody>
</table>

Bit 4 specifies the trigger operation: unity for a positive sloping trigger zero for a negative slope. The prescaler is specified by bit 5:

Lastly, the operating mode and interrupt enable can be specified as shown below:
II.7.2 Time Constant Calculation

To achieve the highest blinking mode, 8Hz, we are obliged to choose the interrupt period every 1/16 sec.

\[ T_i = \frac{1}{8} = \frac{2}{16} \]

One period of 8Hz means that we have to create an interrupt by changing state.

\[ T = \frac{2}{16} \]

\[ t \ (\text{sec}) \]

\[ \text{int} \]

\[ \text{active} \]

\[ T_c \]

\[ T \]

\[ \text{int} \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]

\[ T_c \]

\[ \text{active} \]
II.7.3 What is an interrupt?

The purpose of an interrupt is to allow peripheral devices to suspend CPU operation in an orderly manner and force the CPU to start a peripheral service routine. Usually this service routine is involved with the exchange of data, or status and control information, between the CPU and the peripheral. Once the service routine is completed, the CPU returns to the operation from which it was interrupted.

Fig. II.6 shows figuratively the communication principle between the interpreter program and the interrupt routine, by creating an interrupt to the interpreter program, the address of the last executed instruction should be saved and a start command to the interrupt routine must be delivered, while the interrupt routine is running, the main program waits for its saved address to continue running. This means that by discussion (demanding, and answering) of the CTC and the CPU respectively the running flow will survive. For that sake we need to use Interrupt Mode 2.
With this mode we (programmers) should maintain a table of 16 bits starting addresses for every interrupt service routine. This table may be located anywhere in memory. When an interrupt is accepted, a 16 bit pointer must be formed to obtain the desired service routine starting address from the table. The upper 8 bits of this pointer is formed from the contents of the I register. The I register must have been previously loaded with the desired value by us i.e.

\[
\text{LD I, A}
\]

The CPU also resets clear the I register so that it is initialized to zero. The lower eight bits of the pointer must be supplied by the interrupting device; actually seven bits where

\[
\text{interrupt service routine address table}
\]

the least significant bit is zero, to guarantee an even location as starting address. Note that the pointer is used to get two adjacent bytes to form a complete 16 bit service starting address.

\[
\begin{array}{c}
\text{MEMORY} \\
\text{Z-80 CPU} \\
\text{PC} \\
\text{SP} \\
\text{IREG}
\end{array}
\]

\[
\begin{array}{c}
\text{JUMP TABLE} \\
\text{(start address of the interrupt routine)}
\end{array}
\]

\[
\begin{array}{c}
\text{INTERRUPT VECTOR} \\
\text{CTC}
\end{array}
\]

\[
\text{ Loaded by the programmer Fig. III.7.}
\]
Once the CTC generates the lower portion of the pointer and places the vector on the data bus in response to an interrupt acknowledge (1), the CPU automatically pushes the program counter onto the stack (1a), and obtains the starting address from the table (3), and does a jump to this address (4). Fig. III.7 shows the sequence of events for vector processing.

II.7.4 Programming the Interrupt Vector Register.

The high order 5 bit of the Interrupt Vector must be written to the CTC in advance as part of the initial programming sequence. To do so, the CPU must write to the I/O port address corresponding to the CTC channel 0. Bits 1, 2 are not used when loading this vector. At the time when the interrupting channel must place the Interrupt Vector on the Z80 Data Bus, the Interrupt Control Logic of the CTC automatically supplies a binary code in bits 1 and 2 identifying which of the four CTC channels is to be serviced.

The "interrupt vector register" is then

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0
\end{array}
\]

F8H

RANDOMLY SELECTED CHANNEL
BY THE PROGRAMMER INITIAL VALUE
Section III.

Demoprog.

After we have discussed the necessary hardware parts needed for the demonstration of lessons, aimed to explain the PRX apparatus, we would like to represent the user's language "Demoprog".

The "Demoprog" has been designed to reflect the salient features of real machines, of several computers. It shows the micro steps of data transfer between one register to another, from memory to register or the other way around.

Specific cycles performed by the machine and the whole machine instructions, can be programmed and demonstrated. All addressing techniques, and all arithmetical operations executed by the machine, can also be didactically demonstrated. The unique possibility is of course to demonstrate a whole program, in our case the PRX program, and in the future the microcomputer programs, which are going to be used as an interface of the digital telephone central station, controlled by its own computer TCP 36.

We need to define the "Demoprog" due to our specific applications. The rules are written in BNF notation.

The "Demoprog" consists of two parts:
1. Program part
2. Control part

An elegant way to define the user's language, can be done by using the BNF notation. (Backus-Naur-Form)
III.1. Definition of The "DEMOPROG" User's Language

<DEMOPROG>::= <PROGRAM>
   <CONTROL PROGRAM>

<PROGRAM>::= <PROGRAM HEADING> <BLOCK>

<PROGRAM HEADING>::= BEGIN <PROGRAM NAME>
   {<FORMAL PARAMETER LIST>}*
   {<DECLARATION>}

<PROGRAM NAME>::= <IDENTIFIER>

<IDENTIFIER>::= <LETTER>{<LETTER> | <DIGIT>}*

<LETTER>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<DIGIT>::= 0|1|2|3|4|5|6|7|8|9

<FORMAL PARAMETER LIST>::= <IDENTIFIER>{, <IDENTIFIER>}

<DECLARATION>::= NAME <VARIABLE>=<IDENTIFIER>

<DECLARATION>::= NAME <VARIABLE>=<IDENTIFIER>

<BLOCK>::= {<FLOW PART><LABEL PART><STATEMENT PART>}END

<FLOW PART>::=<EMPTY>@<FLOW MODE>

<EMPTY>::=

<FLOW MODE>::= F1 | F2 | F3 | F4 | F5

<LABEL PART>::= <EMPTY> | /<IDENTIFIER>

<STATEMENT PART>::= <MODE CHANGE STATEMENT>
   | <ASSIGNMENT STATEMENT>
   | <CODING STATEMENT>
   | <COMMAND STATEMENT>

<MODE CHANGE STATEMENT>::=<VARIABLE>:<MODE>

<ASSIGNMENT STATEMENT>::=<VARIABLE>=<EXPRESSION>:<MODE>

<VARIABLE>::= <IDENTIFIER> | <IDENTIFIER><SELECTOR>

<EXPRESSION>::= <SIMPLE EXPRESSION>
   | <SIMPLE EXPRESSION><OPERATOR><SIMPLE EXPRESSION>

<SIMPLE EXPRESSION>::=<PATTERN> | <VARIABLE>

<PATTERN>::=#<HEX NUMBER>
   | #<OCT NUMBER>
   | #<BIN NUMBER>
   | #<DEC NUMBER>
<HEX NUMBER>::=H{<HEX DIGIT>}+4
<HEX DIGIT>::=0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
<OCT NUMBER>::=Q{<OCT DIGIT}>+6
<OCT DIGIT>::=0|1|2|3|4|5|6|7
<BIN NUMBER>::=B{<BIN DIGIT}>+16
<BIN DIGIT>::=0|1
<DEC NUMBER>::=D{<DIGIT}>+5

SELECTOR>::=[<ELEMENT>{, <ELEMENT>}]}
<ELEMENT>::=<NUMBER>{.. <NUMBER>}^1
<NUMBER>::=<DIGIT> | <DIGIT><NUMBER>

<OPERATOR>::=<ADDING OPERATOR>

<ADDING OPERATOR>::=+-

<LOGICAL OPERATOR>::=<AND> | <OR> | <XOR>

<AND>::=&
<OR>::=*
<XOR>::=$

<MODE>::=MZ | MC | MF | MM | MS | MT | MI | MCPY

<CODING STATEMENT>::=<IDENTIFIER>=COD<PATTERN>:<MODE>|<IDENTIFIER>=DEC<PATTERN>:<MODE>

<COMMAND STATEMENT>::=CLEAR(<IDENTIFIER>{, <IDENTIFIER>})

CLEAR ALL|
SET ALL

<CONTROL PROGRAM>::=RECALL<LABEL PART>|BYE|
|f1 | f2 | f3 | f4 | f5
Some examples of the statements used in the Demoprog.

1. \(<\text{MODE CHANGE STATEMENT}\)::=\(<\text{VARIABLE}):(\text{MODE})\)
\(<\text{VARIABLE}>::=\(<\text{IDENTIFIER}>\)\)
\(<\text{IDENTIFIER}> \text{ <SELECTOR}>\)
\(<\text{SELECTOR}>::=\(<\text{ELEMENT}>\{ , \text{<ELEMENT>}\}\)
\(<\text{ELEMENT}>::=\(<\text{NUMBER}>\{..\text{<NUMBER>}\}\)\)
\(<\text{NUMBER}>::=\(<\text{DIGIT}>\)

Examples

1.a. IR : MM
IR takes new mode the MM mode (mode medium)

1.b. IR[1,4,10..16] : MF
The bits 1,2,10 till 16 keeps their value and change their
to MF mode (mode fast)

2. \(<\text{ASSIGNMENT STATEMENT}\)::=\(<\text{VARIABLE}):(\text{EXPRESSION}):(\text{MODE})\)
\(<\text{EXPRESSION}>::=\(<\text{SIMPLE EXPRESSION}>\)\)
\(<\text{SIMPLE EXPRESSION}>::=\(<\text{OPERATOR}>\text{<SIMPLE EXPRESSION}>\)\)
\(<\text{SIMPLE EXPRESSION}>::=\(<\text{PATTERN}>\)\)
\(<\text{VARIABLE}>\)

Examples

2.a. IR=#A3D1 : MM
Load IR with pattern A3D1 hexadecimal and change its mode to MM

2.b. IR=AR : MM
Load IR with the content of AR and set its mode to MM

2.c. IR=UR+VR:MF
Load IR with the sum of UR and VR and set its mode to MF

2.d. IR=UR+3 : MM
Load IR with the content of UR added to 3 and set its mode to MM
2.e. \[SC = VR[1..4] : MM\]
Load SC with the bits 1 to 4 of the vr register and set its mode to MM

2.f. \[IR[1..4] = VR[9..12] \ : \ MF\]
Load the bits 1 to 4 of IR with the content of the bits 8 to 11 of VR, and set its mode to MF

3. \(<\text{DECLARATION})::=\text{NAME} \langle\text{VARIABLE}\rangle = \langle\text{IDENTIFIER}\rangle\]

Example

3.a. \text{NAME} IR[1..8] = PC
the bits 1 to 8 of register IR takes a new name PC

4. \(<\text{CODING STATEMENT})::=\langle\text{IDENTIFIER}\rangle = \langle\text{COD<PATTERN}\rangle: \langle\text{MODE}\rangle\]
\[\text{DEC<PATTERN}: \langle\text{MODE}\rangle\]

Examples

4.a. if the following statement is already existing,
\[SC = \text{#B101 : MF}\]
which means that the register SC is loaded with 101 binary and its mode is MF, then by writing this statement:
\[UR = \text{DEC SC} : \text{MM}\]
the register UR will be loaded with the decoding value of SC namely 0010000 and its mode will be the MM mode.

4.b. if the following statement is already existing,
\[UR = \text{#B1000000} : \text{MF}\]
which means the UR register is loaded with the mentioned binary value and is blinking with the MF mode, the following statement:
\[SC = \text{COD UR} : \text{MF}\]
will load the register SC with the coding value of UR namely 111 and sets it mode to MF.

5. \(<\text{COMMAND STATEMENT}: = \text{RECALL } \langle\text{LABEL PART}\rangle |\)
\[\text{CLEAR}(\langle\text{IDENTIFIER}\rangle\{,\langle\text{IDENTIFIER}\rangle\})|\]
\[\text{CLEAR ALL} |\]
\[\text{SET ALL} |\]
\[\text{BYE}\]

Examples

5.a. RECALL LDA1
the label part of the inputfile is recalled

5.b. CLEAR (UR,WR)
the registers UR,WR are set to zero

5.c. CLEAR ALL
initialize the whole panel

5.d. SET ALL
set all lamps of the panel on

5.e. BYE
close and save inputfile and write to console
'PRINT DEMO \langle\text{COURSE NAME}\rangle' and go to RIO '%%' command.
III.2. Language Structures and Compilers.

We aim to develop the compiler of the Demoprog, using systematic programming technique in the high programming language PLZ. In this respect, it constitutes a welcome application of the program and data structuring disciplines exposed and elaborated in this section.

We shall start by describing language composition and will then concentrate exclusively on simple structures that lead to the modular translator.

Every language is based on a vocabulary. Its elements are ordinarily called words; in the realm of formal languages, however, they are called (basic) symbols. It is characteristic of languages that some sequences of words are recognized as correct, well-formed sentences of the language and that others are said to be incorrect or ill-formed. What is it that determines whether a sequence of words is a correct sentence or not? It is the grammar, syntax, or structure of the language. In fact, we define the syntax as the set of rules or formulas which defines the set of (formally correct) sentences. More importantly, however, such a set of rules not only allows us to decide whether or not a given sequence of words is a sentence, but it also provides the sentences with a structure which is instrumental in the recognition of a sentence's meaning. Hence, it is clear that syntax and semantics (= meaning) are intimately connected. The structural definitions are therefore always to be considered as auxiliary to a higher purpose. This, however, must not prevent us from initially studying structural aspects exclusively, ignoring the issues of meaning and interpretation.

Take for example, the sentence:

Girls sleep

The word "Girls" is the subject and "sleep" is the predicate.
This sentence belongs to the language that may, for instance, be defined by the following syntax.

\[
\begin{align*}
\langle \text{sentence} \rangle & ::= \langle \text{subject} \rangle \langle \text{predicate} \rangle \\
\langle \text{subject} \rangle & ::= \text{girls} | \text{boys} \\
\langle \text{predicate} \rangle & ::= \text{sleep} | \text{eat}
\end{align*}
\]

The meaning of these three lines is:

1. A sentence is formed by a subject followed by a predicate.
2. A subject consists of either the single word "girls" or the word "boys".
3. A predicate consists of either the word "sleep" or the word "eat".

The idea then is that a sentence may be derived from the start symbol \( \langle \text{sentence} \rangle \) by repeated application of replacement rules.

The formalism or notation in which these rules are written is called Backus-Naur-Form (BNF). The sentential constructs \( \langle \text{sentence} \rangle, \langle \text{subject} \rangle, \langle \text{predicate} \rangle \) are called non-terminal symbols, the words girls, boys, sleep, eat are called terminal symbols, and the rules are called productions.

The symbols:

\[
 ::= , ~ | 
\]

are called meta-symbols of the BNF notation.

The language defined by this syntax consists of the four sentences:

\[
\begin{align*}
girls & \text{ sleep} \\
boys & \text{ sleep} \\
girls & \text{ eat} \\
boys & \text{ eat}
\end{align*}
\]

To build up one of the above mentioned sentences, we can follow the following sequence:
111.9

<sentence> — <subject> <predicate> —
girls <predicate> — girls sleep:

hence <sentence> — girls sleep, and since
girls sleep ∈ T^*

where T^* denotes the set of all sequences of symbols from T

T denotes the vocabulary of terminal symbols.

L denotes the language.

Not that <subject> and <predicates> occur in non-terminating steps only, whereas the terminating step must lead to a sequence that contains to one of the above mentioned sentences.

The grammatical rules are called productions, because they determine how new forms may be generated or produced.

For the sake of brevity, we use the capital letters for non-terminal symbols, the Greek letters to denote sequences of symbols.

We can now present the following mathematical definitions:

1. Let a language L = L(T,N,P,S) be specified by
   a. A vocabulary T of terminal symbols.
   b. A set of non-terminal symbols (grammatical categories).
   c. A set P of productions (syntactical rules).
   d. A symbol S (from N), called the start symbol.

2. The language L(T,N,P,S) is the set of sequences of terminal symbols € that can be generated from S according to rule 3 below.

\[ L = \{ \xi | S \rightarrow^* \xi \text{ and } \xi \in T^* \} \]

3. A sequence \( \sigma \) can be generated from a sequence \( \sigma_0 \) if and only if there exist sequences \( \sigma_1, \sigma_2, \ldots, \sigma_{n-1} \) such that every \( \sigma_i \) can be directly generated from \( \sigma_{i-1} \) according to rule 4 below:

\( (\sigma_0 \rightarrow^* \sigma_n) \rightarrow ((\sigma_{i-1} \rightarrow \sigma_i) \text{ for } i = 1 \ldots n) \)
4. A sequence $\gamma$ can be **directly generated** from a sequence $\xi$ if and only if there exist sequences $\alpha, \rho, \xi', \gamma'$ such that

a. $\xi = \alpha \xi' \beta$

b. $\gamma = \alpha \gamma' \beta$

c. $P$ contains the production $\xi' ::= \gamma'$

A language is said to be **context-free** if and only if it can be defined in terms of a context-free production set. A set of productions is context-free if and only if all its members have the form,

$$<\text{subject}>::=\xi \quad (\text{<subject> } \in \text{ N, } \xi \in (\text{N} \cup \text{T})^\ast)$$

i.e., if the left side consists of a single non-terminal symbol and can be replaced by $\xi$ regardless of the context in which $<\text{subject}>$ occurs. If a production has the form

$$\alpha <\text{subject}> \beta ::= \alpha \xi \beta$$

then it is said to be **context-sensitive** because the replacement of $<\text{subject}>$ by $\beta$ may take place only in the context of $\alpha$ and $\beta$. 
Section IV.

IV.1. Software Design.

As computer programs get larger, and are made to carry out more complex tasks, they become more complex and prone to errors in logic. To produce a design free of logical errors, it is necessary to remove all ambiguities and inconsistencies from the specification. However, in order that the specification should convey sufficient information for the product to satisfy the predetermined requirements, it must be couched in terms that are natural to the application it describes. We shall follow the steps of structured programming method, because the design of software becomes a rigorously ordered and staged transition from the high-level specification to the low-level implementation. The initial specification is divided into a number of functionally related subsystems. Each subsystem has its associated specification which itself is a subset of the total system specification. The reasons for a specific subdivision being adopted should be completely documented as part of the design process. Each subsystem may be considered to contain functionally related tasks such that the degree of interaction between these tasks is greater than that between tasks of different subsystems. Proceeding further, each task is divided into a number of steps which specify the processing to be carried out. It is important to specify the processing involved at each step in a language appropriate to the level of detail being considered. No attempt to use a specific computer language should be made until the refinement of a particular step is sufficiently detailed to allow an almost one-to-one correspondence between the description of a substep and a statement in the language.

The top-down refinement process structures the design according to well defined hierarchy. At the heart of the method is the need to divide the design into easily comprehensible units.
This not only helps to ensure that the program does what is was supposed to do at the outset, but it also demonstrates the fact to other programmers. A program that can be understood stands a far greater chance of being modified correctly. Stepwise refinement, in fact, forces the design into a rigorous modifiable structure. Where a modification or extension of the specification is necessary, and the subsequent decomposition steps are replaced by those pertinent to the new requirements. Indeed this is one of the most important wishes requested by designing the 'demoprog', hence, it appears that the demonstrating language should be able to fulfil the changing courses in the future which is the typical character of the training centre of P.I.T.T.C.: 'Philips' International Telecommunications Training Centre.'

After we have completed the definition of the 'demoprog' (see section III) we can try designing the software structure of the interpretor:

Main program.

After initialisation of the important variables used in the whole program, and as long as a certain codition is true, the program has to check up the syntax used by the user, in computer terms "the source", and ofcourse to interprete it. The interpretation will be executed if the syntax check agreed with the content of the source!
After interpretation the action must be ended.

Syntax analyzer (parser)

The syntax analyzer has an input and an output.

The input: is a row of abstract symbols supplied by the lexical scanner, in the variable called symbol.

The output: error if there is any, or an intermediate form of the demogrogram, with some variables needed as an input to the interpretor.
Lexical scanner.

The lexical scanner is assessable with two tasks, getting the `nextsymbol` and gathering the `nextcharacter`.

Next symbols

The `nextsymbols` (abbr.: `nextsym`) consists of an input and an output.

The input: receiving an "interesting" character handed by `nextcharacter` with variable `ch`.

The output: the abstract symbols, in the variable `sym`.

Next character

The `nextcharacter` (abbr.: `nextchr`) consists of an input and an output.

The input: ASCII characters as they are supplied by `getseq` procedure.

The output: an interesting character in the variable `ch`. 
Beside the main program, we need an interrupt routine to reach the several blinking modes (see also section IV), in other words the interrupt routine shall inspect an array of instructions which has been used by the user (written in "Demoprog" user's language) and send the right signals to the input/output port (see section III); precisely port A will serve the gate for the lamps, and port B will select sequentially the group of lamps.

We have to define the way of preparing the demoprogram (the users language program), before executing.

We shall make use of the existing file editing program supplied by the RIO operating system, to prepare the lesson written in the "Demoprog user's language", act from an existing file. We based ourselfs on the assumption of an existing file which has to be called by the main program.

How to create a file?

After initializing the RIO operating system, the user will find a brief displayed explanation, how to edit and execute the demoprogram.

We can create the file by editing it, as follows:

after system initialization the symbol will appear which means that the system is on RIO command, this intitles use to request the EDIT program as follows:

```
% EDIT <Lesson name>
EDIT
NEW FILE
INPUT

{here follows the lesson written in Demoprog User's Language}
Begin
.
.
end
```
by twice return, the system will go to the editing command which has the symbol >

changing and correcting the lesson can be fulfilled by the editing command (see text editor of the Z-80 system)

Quit {will turn us to RIO command}

Testing the lesson:

After creating the lesson, the user has the ability to test and demonstrate the lesson by using the following command.

% DEMO <LESSON NAME>

N.B. In the future the user has to specify the course name afterwards the lesson name, e.g.

% DEMO <LESSON NAME> ... <LESSON NAME>

Let us try now the stepwise refinement method to explain the algorithms of the "Demoprog" program, for this aim we shall use the high-programming language PLZ. Before we start, let's explain the word algorithms;

What is an Algorithm?

According to Webster's Dictionary, an "algorithm" is any special method for solving a certain kind of problem. Actually we use an established algorithm for carrying out almost any familiar task, though we rarely stop to think of the various component steps of algorithms. To be useful in the computing context, we'll have to narrow down the definition of an algorithm a bit further.

For our use we can define "Algorithm" as "a list of instructions for carrying out some process step by step". Although this definition avoids specifying the level of detail to be used in the list, we will find it useful to start with a list of
very rough and informal instructions, and then to start refining these instructions in designing a final detailed version of the algorithm in the form of a program.

The nature of the digital computer leads to the specification of solutions to problems using algorithm, i.e. using "algorithmic" notation. Since the computer can perform only one small step at a time, people have generally been led to concentrate more on the order in which the steps are taken, than on understanding how each step relates to the problem as a whole. Today we know that it is often better to reverse this process, leaving the order of processing to be determined after the various major steps of the task have been defined.

The problem is then to develop a program which interpret the lesson. The interpret program can be formulated as follows:

demo module
external ...
internal ....
main procedure
entry
"openfile"
"read line after line of file and store in memory"
"scan source"
"perform syncheck, context free & sensitive"
if
"no errors found"
then "interpret program"
"close file"
fi
end main

We now proceed to specify the various statements in greater detail. Refining "openfile", "read line after line of file and store in memory", "scan source" and "perform syncheck, context free & sensitive", we note that those instructions are a preparing phase to
the instruction "interpret program.

The "openfile", "close file" and "read line after line of file
and store in memory" will be executed automatically by the operat­ing system, when we make use of the PLZ stream I/O facility,
after declaring the following external procedure:

\[
\text{external open procedure } \text{(unit byte, filename-ptr}
\begin{align*}
\text{byte flag byte)} \\
\text{returns } (rcode byte) \\
\text{close procedure } \text{(unit byte)} \\
\text{returns } (rcode byte)
\end{align*}
\]

open and closing the file will be as follows respectively:

\[
\text{code := open (infile, filenameptr, input)} \\
\text{code := close (infile)}
\]

**IV.2 Scan Source Module**

This procedure has the task to read the characters with the
procedure nextch, and to hand it over to another procedure
called nextsym. In the scan procedure we need the following
declaration:

\[
\text{type} \\
\text{ alfa array [8 byte]} \\
\text{ global} \\
\text{idname alfa} \\
\text{intval integer} \\
\text{sym byte} \\
\text{interval } \\
\text{repr array 23 alfa := } [\text{['B','E','G','I','N','',''','']}, \\
\text{['B','Y','E','','','','','']}, \\
\text{ all reserved words }] \\
\text{used in "Demoprog" ]}
\]

\]
The variable "idname" has originally the array type which is used as a dynamic mechanism. It will be filled by the characters fetched by nextchr procedure, afterwards compared with the content of repr array, if the result is true then the variable "sym" will take the corresponding value which declared as "constant", the value of "sym" is necessary to the parser.

nextchr procedure
entry
ch:= getch; putch(ch)
if rcode="endoffile" then ch= EOF fi
if ch= CR or if ch= TAB then ch:= ' ' fi
end nextchr

n.b. ch:= getch and putch(ch) are procedures which can be found in the main lexical scanner program.

nextsym procedure
entry
"after skipping blanks and reading comment"
if ch
  case 'A',..., 'Z'
  then I:=0
  do idname[i]:=ch
     i:=i+1; nextchr
     if (i=8 or if ch 'A' or if '9' ch)
        and if (ch '0' or if '9' ch)
        then exit
        fi
  od
  do
  "fill blanks in the rest of idname array if the read sym is less than eight character"
  od
do
  "if we meet ch \notin \{ch'[A'...'Z' or '0'...'G']\} then exit"
  "now we give sym a value called identifier"
  "and we compare the content of idname variable with our data structure constructed by name repr".
  "if we find one then we recognise the corresponding value to variable sym".
od

\textbf{case} \\
\textbf{then nextchr}
if ch
  \textbf{case} 'B' \textbf{then} "scanbinarynumber" \\
  \textbf{case} 'Q' \textbf{then} "scanoctalnumber" \\
  \textbf{case} 'X' \textbf{then} "scan hexnumber" \\
  \textbf{case} 'D' \textbf{then} "scan decnumber"
else nextdel
fi
end nextsym
For the sake of brevity, we shall explain the "Scanbinarynumber" only, indeed the four scan numbers algorithm are similar.

```plaintext
  case 'B'
  the intval:=0; sym:=binnum
  count:=1; nexchr
  do if ch /\ [ch | '0', '1'] or if count > 16
    then exit
    fi
  intval:=(2^intval) + (integer ch-48)
  fi
  ! every letter we meet its value must correspond with:
  ! its weight, this will be achieved by (2^intval),
  ! further the binary value should be changed to its
  ! decimal one, the number 48 is the ASCII base number,
  ! we know that '0' in ASCII is 30H this means in bi-
  ! nary 00110000. This binary number has 48 decimal, i.e.
  ! so every character we meet which is an element of
  ! the set of numbers, can be changed to its decimal
  ! value by first changing the type of the ch to in-
  ! teger and reduced by the 48 value:
  nexchr
  count:= count + 1
  od
  do it if ch /\ [ch | '0', '1'] then exit fi
  nextchr
  od
  if count > 16 then error (illegal binnumber) fi
```

If the selecting statement "case statement" couldn't find the given character within the nextsym procedure, we invoke "next-del procedure" expecting that the fetched character will be perhaps a delimiter.
The "nextdel" procedure has the task to select all used delimiters and present the "sym" variable its corresponding value:

```plaintext
nextdel procedure
  entry
    if ch
      case '!' then sym:= exlmsym
      case '#' then sym:= numsym
      etc.
      else "most probably the fetched character doesn't exist in the chosen character set"
    fi
  end nextdel
end demo
```
IV.3 Parse (syntax analyser) Module

The task of language translators or processors is primarily not the generation but the recognition of sentences and sentence structure.

This implies that the generating steps which lead to a sentence must be reconstructed upon reading the sentence, and that its generation steps must be retraced. This is generally a very complicated and sometimes even an impossible task. Its complexity intimately depends on the kind of production rules used to define the language. It is the task of the theory of syntax analysis to develop recognizing algorithms for languages with rather complicated structural rules. Here, however, the syntax of the "Demoprog" has not of that complexity, but neither preventing us of using the top down parsing method.

A first consequence of the basic efficiency requirement is that the choice of every analysis step must depend only on the present state of computation and on a single next symbol being read. Another most important requirement is that no step will have to be revoked later on. These two requirements are commonly known under the technical term one-symbol-lookahead without backtracking.

There are two essentially different techniques that can be applied. One is to design a general top-down parsing program, where particular grammars are to be supplied in the form of some data structure, on the basis of which the program operates. This parser is in some sense controlled by the data structure; the program is then called table driven. The other one, the one we are going to apply, is to develop a top-down parsing program, constructed systematically, and mapped a given syntax into a sequence of statements, i.e. into a program.

It is advantageous to represent the given syntax by a so-called recognition or syntaxgraph. This graph reflects the flow of control during the process of parsing a sentence, a general view
of the syntaxgraph can be seen in appendix.

It must be clear for us by applying this method, that the goal of the parsing process will be known from the start. The goal is to recognize a sentence, i.e., a sequence of symbols generatable from the start symbol.

Let us start from the top, we have seen from the definition of the "Demoprog", that it consists of:

1. Program
2. Control program

Before we start discussing the parser procedures, it is worthwhile to give a view of the data structure. We need some important specifications to define the panel, first of all the name of the registers, starting number of every register and of course the length of the register. For that sake we define a record with two fields;

```
Type
record [name array [8 byte]
    lampnvar, lengthvar byte
]

internal
regtable array [48 lampinfo] = [ [1:eqname reg'],0,16] [ [2:eqname reg'],16,16]
    lampreg, lengthreg byte
```

All parsing procedures assume that the first symbol of a recognizing construction is represented in the variable "sym" which has been discussed in the lexical scanner.

1. The program procedure actually consists of invoking the two procedures "programheading" and "block".
Programheading

<table>
<thead>
<tr>
<th>Program heading</th>
<th>procedure</th>
<th>entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminal (beginsym) ; terminal (identifier)</td>
<td>if sym = opensym then param-list fi</td>
<td>if sym = namesym then nextsym; lampreg, lengthreg := variable</td>
</tr>
<tr>
<td>terminal (eqsym); terminal (identifier) fi</td>
<td>end programheading</td>
<td></td>
</tr>
</tbody>
</table>

Block

<table>
<thead>
<tr>
<th>Block procedure</th>
<th>entry</th>
<th>do</th>
</tr>
</thead>
<tbody>
<tr>
<td>if sym = flowsym then nextsym; newflow := flowmode fi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if sym = labesym then nextsym; label-part fi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>statement if sym = endsym then exit else error(14) ; exit fi</td>
<td>end block</td>
<td></td>
</tr>
</tbody>
</table>
We have invoked the procedure statement in the block procedure, how does this procedure looks like? Well, actually this is the heart of the "demoprog", it terminates with semicolon or carriage return symbol.

```plaintext
statement procedure
  entry
    if sym=identifier
      then lampreg,lengthreg:=variable
      if sym=colonsym
        then nextsym; newmode:=mode
      else
        if sym=eqsym
          then nextsym
          if sym=codsym or if sym=decsym
            then nextsym; pattern
              terminal(colonsym)
              newmode:=mode
          else value:=expression
            terminal(colonsym)
            newmode:=mode
        fi
      fi
    else
      error(9)
    fi
else
  fi
```

IV.15.
if sym=clearsym
    then nextsym;param_list
else
    if sym=clearallsym
        then newmode:=mode
    else
        if sym=setallsym
            then newmode:=mode
        else error(13)
fi
fi

end statement

The procedure variable was necessary in the statement procedure, which delivers the number of the first register and its length;

variable procedure
returns (lampnovar,lengthvar byte)
local i,j byte
entry
    j:=0
    do
        if j=49 then exit fi
        i:=0
        do
            if regtable[j].name[i]<idname[i]
                then exit
            fi
            if i=7 then
                lampreg:=regtable[j].lampnovar
                lengthreg:=regtable[j].lengthvar
                exit
            fi
            i:=i+1
        od
    j:=j+1
    od

end variable

We have tried to give an impression on the way of programming, during the parsing procedures, a further detail can be found in the attached programs.

2. Control program.

The control program has the task to act independently, on the main program. A kind of GOTO statement has been provided to
enable the user during demonstrating, to repeat the demonstration in one or other parts of the lesson. This command statement is called RECALL statement. The user can quit demonstrating by using the command word BYE.

The control program is provided by five flow modes. By means of these flow modes the teacher can define demonstration steps, it consists of step by step flow modes (f1) which is defined for the programmer as a default value, while the rest can be used in random places in the program. (Note: that the user shall make use of the ASCII characters F1, F2, F3, F4, F5 within the program, while the corresponding function push button f1, f2, f3, f4, f5)

The functions button has been chosen for the sake of simplility and quickness during the demonstration.

The syntax graph will be as follows:
Appendix A.

Demoprog Syntax Diagram

Program

Programheading

BEGIN → Progname → Param_list → NAME → Variable → ident → statement → END

Block

Flow Mode

Label_Part

Statement

variable → expression → mode → CLEAR → CLRALL → SETALL

COD → Pattern → DEC → Param_list
Conclusion:

At the early stage of developing my subject, I took special care to achieve an applicable system, after all, the project has been supplied financially by the PITTC.

I believe the simplicity of the DEMOPROG rules is important to encourage the lecturer of frequent use. Although it is a bit exaggerating to use a software developing system Z-80 for the sake of teaching only, one can advise that the next step which has to be taken, to develop a kind of time sharing procedure to make the most of the microprocessor.

I would like to thank prof. A. Heetman and Ir. J.A. Samwel who gave the opportunity to perform this subject. I am very grateful to Ir. Engbers, Ir. Kemper, Ir. v/d Berg for their advice on the general layout. I appreciate the time prof. Kruseman Aretz and Ir. Hemerik could find to help me on software problems. From the side of the PITTC I would like to thank mr. P. Dane for his practical advice.
References:

   David Gries

2. Algorithm+Data Structure = Programs 
   Niklaus Wirth

   Niklaus Wirth

   Robert M. Graham

5. Microprocessors. 
   Rodnay Zaks

   Tod Snook

7. Soft & Hardware of the microcomputer system Z-80 Zilog.