Circuit Tutor:
A Computer-Aided Learning Package For Electrical Engineering

L. Potgieter

MSc Thesis

Department of Electrical Engineering
University of Cape Town
November 1988

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

ACKNOWLEDGEMENTS

I would like to record my thanks to the following:

Mr John Greene from the Dept. of Electrical Engineering UCT, who supervised this dissertation, for his guidance and advice.

My wife Nicky, for her encouragement and support and for proofreading this dissertation.

Finally to the Dept. Post and Telecommunications, who generously allowed me leave to complete this dissertation.

Synopsis

The development of Circuit Tutor, the subject of this dissertation, resulted from a conviction that computers can further enrich the Electronic Engineering curriculum.

After an investigation into the different roles of the computer in education the use of modelling and simulation was selected as an effective Computer Aided Learning method.

It was realised that the development of any non-trivial simulation program is however not an easy task. The programmer must not only model the circuit behaviour, but also write the man-machine interface (MMI). The main goal of Circuit Tutor was to provide a ready-made simulation environment which makes effective use of the graphics capabilities of the microcomputer for the simulation of a whole class of electrical circuit simulations. To facilitate rapid prototyping the installer is provided with:

(1) a man-machine interface which provides the user with a graph, 3 meters, a circuit diagram of the circuit, a menu facility, windows to view circuit parameters and outputs;

(2) a program scheduler;

(3) a library of maths functions, including Gauss-Jordan elimination of complex matrices; and

(4) Circuit Draw: a utility to draw a circuit diagram.

Particular emphasis was placed on the design of the user's interface. It has been possible to restrict the effort to link in a new circuit model to 3 modifications to the man-machine interface (MMI) part of the program.

Present software and MMI design were investigated. Circuit Tutor and Circuit Draw were developed using modular software design techniques. A modular design chart similar to that proposed by Wiener (1984) was found to be useful during the design stages of both Circuit Tutor and Circuit Draw. Available computer languages for the IBM PC were evaluated and Turbo Pascal selected, as it offered most of the features necessary for the implementation of a modern, modular software design. Four circuits were implemented to serve as examples.

The documentation was structured in a manner appropriate to a software project:

Part 1 gives an introduction to computers in education and provides the rationale for the use of simulation. A brief overview of Circuit Tutor and Circuit Draw is presented.

Part 2 contains the User's Reference Manual for Circuit Tutor and the Circuit Draw Utility.

Part 3 contains the Designer's Reference Manual for Circuit Tutor and the Circuit Draw Utility.

TABLE OF CONTENTS

PART 1: Computer Aided Learning and Circuit Tutor: An Overview

1 INTRODUCTION TO COMPUTERS IN SCIENCE EDUCATION 1-1

1.1 Terms and definitions 1-1
1.2 Tutorial CAL 1-3
1.3 Laboratory CAL 1-5
1.3.1 Models and simulations 1-6
1.4 NDPCAL 1-7

2 THE CIRCUIT TUTOR SYSTEM 1-8

2.1 Goals of Circuit Tutor 1-8
2.2 Selection of a CAL method for Circuit Tutor 1-8
2.3 Choice of hardware 1-10
2.4 General requirements of a man-machine interface 1-10
2.5 Overview of Circuit Tutor features 1-11
2.6 Expanding Circuit Tutor 1-11
2.7 Evaluation of Circuit Tutor 1-16
2.8 Possible improvements and enhancements to Circuit Tutor . . . 1-18

BIBLIOGRAPHY 1-19

PART 2: Circuit Tutor User's Reference Manual

1 INTRODUCTION 2-1

1.1 What is Circuit Tutor 2-1
1.2 Structure of this document 2-1

2 USING CIRCUIT TUTOR 2-3

2.1 Hardware requirements 2-3
2.2 Distribution diskette files 2-3
2.3 Starting Circuit Tutor 2-4
2.4 Selecting a circuit 2-4
2.5 Circuit Tutor display 2-7
2.6 Modifying circuit parameters 2-8
2.7 Monitoring the circuit outputs 2-11
2.7.1 Using the meters 2-11
2.7.2 Using the graph 2-13
2.8 Circuit Tutor menu reference 2-13

3 USING THE CIRCUIT DRAW UTILITY 2-18

3.1 System requirements 2-18
3.2 The Circuit Draw distribution diskette 2-18
3.3 Starting Circuit Draw 2-19
3.4 Drawing diagram symbols 2-21
3.5 Drawing diagram text 2-23
3.6 Defining symbols (graphics font editor) 2-23
3.7 Circuit Draw menu reference 2-26

4 ADDING A CIRCUIT TO CIRCUIT TUTOR 2-28

4.1 Creating the unit 2-28
4.1.1 IdentifyCct statement 2-29
4.1.2 InitCct statement 2-29
4.1.3 SolveCct statement 2-30
4.2 Modifying Circuit Tutor 2-33

APPENDIX A. CIRCUIT TUTOR MESSAGES 2-34

APPENDIX B. CIRCUIT TUTOR INSTALLATION GUIDE 2-36

PART 3: Circuit Tutor Designer's Reference Manual

---

1 SPECIFICATION FOR CIRCUIT TUTOR 3-1

1.1 Specification methodology 3-1
1.2 General goals 3-1
1.3 Functional requirements 3-3
1.4 Non-functional requirements for Circuit Tutor 3-4

2 DESIGN OF CIRCUIT TUTOR 3-6

2.1 Design of the VDU lay-out of Circuit Tutor 3-6
2.2 Selection of a VDU format 3-8
2.3 Detailed VDU lay-out 3-11
2.4 Overview of current software design methodologies 3-11
2.5 Criteria for choosing a programming language 3-11
2.6 Choosing a computer language 3-14
2.7 Choosing a design methodology 3-17
2.8 Identifying the main objects 3-18
2.9 Preparing a modular design chart 3-22

3 CIRCUIT TUTOR UNITS IN DETAIL 3-24

3.1 The main (program) unit 3-25
3.1.1 Description of routines 3-26

3.2 The Graph unit 3-29
3.2.1 Description of routines 3-30
3.3 The Meters unit 3-33
3.3.1 Description of routines 3-34
3.4 The Pars unit 3-36
3.4.1 Description of internal data 3-37
3.4.2 Description of routines 3-38
3.5 The CctDgm unit 3-45
3.5.1 Co-ordinate systems used by CctDgm unit 3-46
3.5.2 Description of routines 3-48
3.6 The UserMsgs unit 3-51
3.6.1 Description of internal data 3-52
3.6.2 Description of routines 3-53
3.7 The Chr8x8 unit 3-57
3.7.1 Exported data 3-57
3.7.2 Description of routines 3-58
3.8 The Misc unit 3-60
3.8.1 Description of routines 3-60
3.9 The Outputs unit 3-64
3.6.1 Description of internal data 3-65
3.6.2 Description of routines 3-65
3.10 The Menus unit 3-69
3.10.1 Exported data 3-71
3.10.2 Description of routines 3-76
3.11 The MmiGloblals unit 3-83
3.11.1 Miscellaneous constants 3-83
3.11.2 Graphics window constants 3-83
3.11.3 Graphics co-ordinate system constants 3-84
3.11.4 Screen lay-out constants 3-84
3.11.5 Variables 3-84
3.11.6 Type identifiers 3-85
3.11.7 Exported function 3-86
3.12 The Ascii unit 3-87
3.13 The Maths and CMaths units 3-88
3.13 Description of exported data and routines 3-88
3.14 Other units 3-90

4 DESIGN OF THE CIRCUIT DRAW UTILITY PROGRAM 3-91

4.1 General goals 3-91
4.2 Specifications for Circuit Draw 3-91
4.3 Design of the VDU lay-out 3-92
4.4 Design of the program 3-92
4.4.1 Preparing a modular design chart 3-98

5 THE CIRCUIT DRAW UNITS IN DETAIL 3-99

5.1 The main (program) unit 3-99
5.1.1 Description of routines 3-100
5.2 The Grid unit 3-103
5.2.1 Description of routines 3-103
5.3 SymTable unit 3-107
5.3.1 Description of routines 3-108

5.4 Diagram unit 3-109
5.4.1 Description of exported variable 3-110
5.4.2 Description of routines 3-110
5.5 DrawText unit 3-113
5.5.1 Description of exported constant 3-114
5.5.2 Description of routines 3-114
5.6 DrawMisc unit 3-116
5.6.1 Description of routines 3-117
5.7 HgcPlus unit 3-118
5.8 DrawGlobals unit 3-118
5.8.1 Miscellaneous constants 3-119
5.8.2 Graphics window constants 3-119
5.8.3 Types and variables 3-120

BIBLIOGRAPHY 3-121

Part 1

Computer Aided Learning and Circuit Tutor: An Overview

TABLE OF CONTENTS

1 INTRODUCTION TO COMPUTERS IN SCIENCE EDUCATION 1-1

1.1 Terms and definitions 1-1
1.2 Tutorial CAL 1-3
1.3 Laboratory CAL 1-5
1.3.1 Models and simulations 1-6
1.4 NDPCAL 1-7

2 THE CIRCUIT TUTOR SYSTEM 1-8

2.1 Goals of Circuit Tutor 1-8
2.2 Selection of a CAL method for Circuit Tutor 1-8
2.3 Choice of hardware 1-10
2.4 General requirements of a man-machine interface 1-10
2.5 Overview of Circuit Tutor features 1-11
2.6 Expanding Circuit Tutor 1-11
2.7 Evaluation of Circuit Tutor 1-16
2.8 Possible improvements and enhancements to Circuit Tutor . . . 1-18

BIBLIOGRAPHY 1-19

1

INTRODUCTION TO COMPUTERS IN SCIENCE EDUCATION

Since computers were invented some 40 years, this technology has advanced at an unprecedented pace. Initially computers were large, difficult to operate, and expensive. Rapid advances in computer hardware technology have however resulted in computers becoming smaller, friendlier, more powerful, and cheaper. With the advent of powerful microcomputers, computers have finally reached the man in the street. Educational institutes can now for the first time afford to employ computers on a large scale. In 1987, for example, the number of microcomputers at U.C.T. had increased to over 700, whilst more than 300 students and lecturers have bought microcomputers [Cousins, 1987, p.7].

Unfortunately software development technology has not kept pace with the advances in computer hardware technology. The production of any non-trivial program is still a labour intensive task, and therefore very expensive. One of the biggest problems still facing educational institutes today, is the unavailability of affordable, good quality software [Broster, 1981].

The computer is extremely versatile: it can be used to do number-crunching, data processing, data storage and retrieval, simulation and animation to name a few. Due to this versatility computers are used in virtually all aspects of education. It is therefore convenient to attempt a classification of the computer in SCIENCE EDUCATION as this thesis is concerned mainly with the computer in this sphere of education.

1.1 Terms and definitions

Computer Based Learning (CBL). The computer is used for the purpose of education either to manage learning (CML) or to assist learning (CAL). *Computers used by university/school administrative departments for administrative purposes: staff records, payroll etc., are excluded.*

Computer Managed Learning (CML). The computer is used to manage teaching. Pure CML applications therefore assist the tutor with his management of teaching, but do not participate **directly** in the teaching process. CML can assist the tutor in tasks such as test marking and analysis, routing students through a course of study, and record keeping.

Computer Aided Learning (CAL). Two forms of CAL can be distinguished in science education namely tutorial CAL and laboratory CAL.

- **Tutorial CAL.** The computer is used to perform some or all of the tutoring functions normally performed by a human instructor/tutor. This mode of the computer will also be termed **Computer Aided Instruction (CAI)**. CAI may be combined with CML, for example during a CAL tutorial the computer can evaluate the performance (CML testing function) of the student and then decide which tutorial the students should be routed to next (CML routing function).

- **Laboratory CAL.** This involves using the computer as a tool or learning resource. The term 'laboratory CAL' is used to distinguish this use of the computer from tutorial CAL. As a tool, the computer can be used for numerical analysis; equation solving; computer aided design (CAD); data storage and retrieval; and modelling and simulation. The student may write a program for the computer or use a pre-written program, for example a spreadsheet program.

- **Tutorial/laboratory CAL.** This is essential a combination of the above two categories. In science education this mode often entails combining simulation with tutorial CAL. For example, Alfred Bork who is one of the pioneers of CAL, has for some time developed simulated laboratory experiments with a tutoring element.

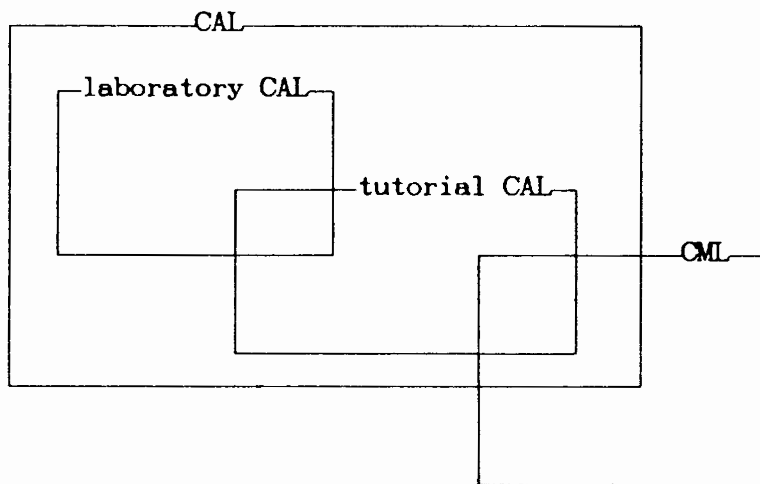


Figure 1-1. Classification of CBL applications in science education.

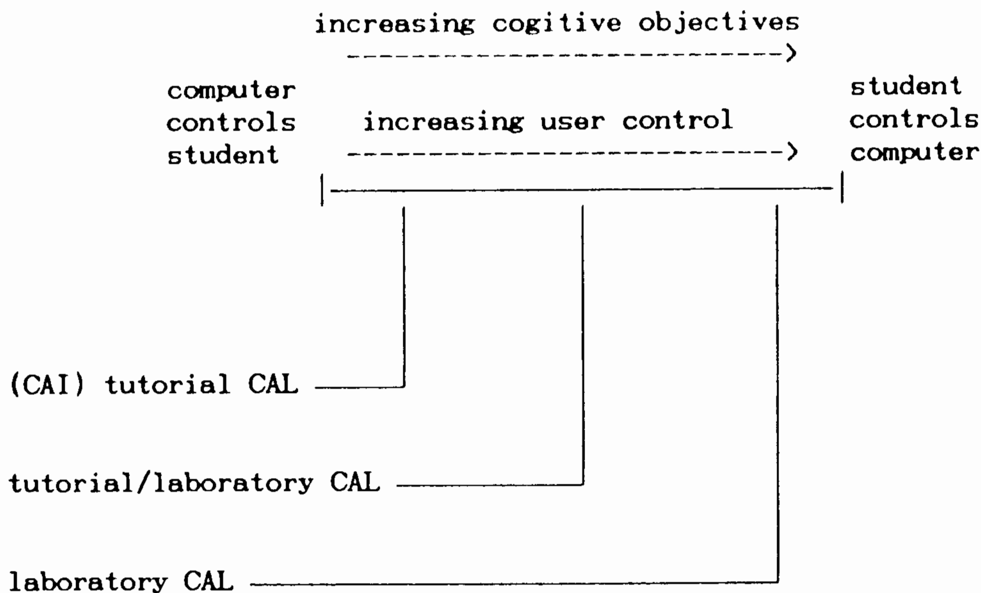


Figure 1-2. Classification of CAL by interaction type.

Figure 1-2 demonstrates the control the user has over the computer. In lower forms of tutorial CAL, where the main purpose is information retention, for example practice-and-drill CAI, the student has little control over the computer. For laboratory CAL, which is mainly concerned with the attainment high-level cognitive objectives, the student has full control over the computer. For tutorial/laboratory CAL the user has some control over the session. For example, Bork comments on his MOTION program, which uses simulation and tutorial CAL: "The program does not 'drive' the student; rather, it persuades students to take an active lead" [Bork, 1981, p. 112].

1.2 Tutorial CAL (CAI)

Computer Aided Instruction is directly descended from the programmed learning and teaching-machine movement of the early 1960s. Essentially the computer attempts to simulate an instructor/tutor. The level of sophistication of a CAI program, depends on the complexity of the instructor/tutor and student models used by the program. In low-end applications, for example non-adaptive practice-and-drill CAI, the instructor and student models are crude or virtually non-existant.

For more sophisticated tutorial CAL applications however, artificial intelligence techniques can be used to simulate a human tutor; or the program can keep a record on the achievement of the student to determine the students progress and to route the student through the subject matter, based on his level of comprehension. Or tutorial CAL can make use of other media, for example the Videodisc, to enhance the tutorial.

Control Data Corporation (CDC), have been one of the earliest manufacturers of CAI hardware and software. CDC has invested some $600 million in 20 years on the development of their PLATO system, which was originally funded by the US government and developed at the University of Illinois [Beech,1983]. A modern (early 1980s) PLATO installation typically comprises of :

- a networked mainframe computer with several intelligent terminals connected to it;
- the TUTOR author language for developing CAI programs;
- several software development tools to assist the programmer/author, for example an interactive graphics editor which allows the programmer to design graphics and then translate these graphics to equivalent TUTOR statements; and a utility called PLM (for PLATO Learning Management) which allows an author to design a course as a collection of modules, without the need for the author to use the TUTOR language;
- terminals fitted with a touch-sensitive graphics visual display unit (VDU) to facilitate ease of input. Terminals are connected via telephone lines to the mainframe computer, but due to their built-in intelligence they can perform certain functions in the stand-alone mode;

CAI applications are normally developed using special **author languages** such as Control Data Corporation's TUTOR language for their PLATO system [Beech, 1983, pp. 5-13]. The main goal of these author languages is to facilitate rapid program development by the tutor/author. Nowadays author languages are also available for microcomputers although they do not offer all the facilities of a mainframe author language such as CDC's TUTOR, or IBM's COURSEWRITER languages.

CAI is said to have these advantages:

- Economical. It is normally argued that the computer can reduce the number of teachers required in education by virtue of its capability to perform some or all of the tutor's functions. This rationale is normally cited for courses where the enrolment figure is very low, i.e. where only a few computers are required (as opposed to one human tutor) [Suppes, 1980, p. 250].
- Individualism. The response from each individual student can be used to determine the next material to be presented. It is therefore possible to tailor the content of the course to match the student needs. Another benefit derived from individualism is that the pace can be matched to the rate of learning of the student.

- **Interactiveness.** This advantage stems from the individualism afforded to the user. Often students will not interrupt a human tutor if they do not comprehend the material being presented, perhaps for fear of being considered stupid. But a well-designed tutor program will allow the user to interrupt the program should additional explanation be required.

- **Shorter course durations.** Although it is a debateable issue some defenders of CAI claim that course durations can be typically 30% less. Little evidence is however available to prove or disprove this statement.

Although CAI has been used in commerce for the training of skills such as salesmanship, touchtyping, and financial accounting with some success (using systems such as PLATO) there is little evidence that this method is more effective than traditional teaching methods when applied to science education at higher educational institutes.

1.3 Laboratory CAL

Although CAI is sometimes regarded as being synonymous with CAL a survey conducted in the U.S.A. in 1980 indicated that CAI constituted less than 10% of all CAL applications (excluding the use of the computer as the object of study) [Skyrme,1981]. The majority of applications used the computer as a learning resource for data retrieval and analysis; problem solving; and games, modelling and simulation. As mentioned the term 'laboratory' CAL is used to define this role of the computer. The following uses of the computer fall into this category:

- **Models and simulation.** This use of the computer will be further expanded in a later section. Briefly simulation may be used by the tutor to demonstrate certain ideas or processes similar to a laboratory experiment; or students may themselves do the investigating with varying degrees of freedom; and in some cases students may create and test their own models, written in a computer language such as Pascal.

- **The computer as 'super-calculator'.** The computer relieves the student of inauthentic labour (work not contributing directly to learning). For example the computer may be used to provide statistical information on large amounts of data, thereby freeing the student to do authentic labour, for example evaluate the processed data.

- **Browsing.** This is the electronic equivalent of browsing in a library, with the advantage that it can be much faster. This approach however, requires large amounts, often gigabytes, of on-line storage. With the advances in local area network (LAN) and optical disk technology, this use of the computer could gain popularity.

1.3.1 Models and simulation

What exactly is understood by models and simulations? A model is a representation which exhibits some, or all, of the properties of a real object. For example, a scaled down replica of an experimental aircraft may be built to represent the full-scale aircraft. This model represents the full-scale aircraft and will exhibit some of its properties. In computer simulation a model is normally a set of equations that represent an object (e.g. the Ebers-Moll models for a transistor).

Simulation is the process of actually using the model, or testing it. Bloomer formally defines simulation as "an operational representation of central features of reality" [Ellington,1981].

In science education simulation is especially useful when it is used as a laboratory substitute. Simulation should however, not be regarded as a replacement for all laboratory work; it should only be used when the simulation offers certain definite advantages to real laboratory work. For example [Maddison, 1982, pp. 111-112]:

- when a laboratory experiment is not practical or impossible;
- when the cost of a real experiment is beyond the means of the institution;
- when it is not possible to observe a phenomenon in real-time. For example plant growth;
- when a real laboratory experiment is time-consuming;
- when an experiment is dangerous;
- when a number of reruns are required. It may be necessary to repeat the experiment several times with different parameters to arrive at a conclusion;
- when idealised conditions are required. The experiment can tailored to achieve pedagogical goals. Certain phenomena, for example friction, can be ignored to create what Bork terms *controllable worlds*. A good example of this is Bork's $F=ma$ controlled world of his MOTION program [Bork, 1981, p.112].

Studies have indicated that simulation is no more effective than other teaching methods for teaching the basics of a subject (low-level cognitive objectives). But for achieving certain high-level cognitive objectives, for example, decision-making and hypothesising, it is more successful than traditional methods. The following list of reasons why simulations are useful in science education are extracted from Bork (1981) and Ellington (1981):

- Simulations can be used for teaching about modelling and simulation, to acquaint students with this very important research tool.
- Simulations (which are well-designed) can have a strong motivational effect on students.
- It is possible to simulate situations which are difficult or impossible to obtain in real life.
- Simulation is suitable for the development of high-level cognitive ability such as hypothesising and decision-making; and develop initiative and powers of creative thought.
- The experiment can be tailored to meet needs of the exercise. For example when idealised conditions are required.
- Simulation is effective for reinforcement or demonstration of principals taught in the classroom.

Two of the biggest problems with current simulations are:

1. **Inadequate speed of computers.** Although computers have come a long way, there are still many simulations that require much more computing power, especially if the simulation is to be used interactively.

2. **Lack of commitment.** Because simulations often have no relationship to other course components, some students will will make more use of them than others who may even ignore simulation completely. To use simulation effectively therefore, requires more than just a casual treatment; simulations must be incorporated into the curriculum, to ensure that all students make equal use of this very important tool.

1.4 The National Development Programme in CAL (NDPCAL)

The NDPCAL in the U.K. has probably been one of the most significant milestones in CAL history and deserves some mention. The primary goal of NDPCAL was "to develop and secure the assimilation of computer assisted and computer managed learning on a regular institutional basis at reasonable cost" [Hooper, 1977, p. 15].

The programme, which lasted 5 years (1973 - 1977), comprised of 35 projects in total, which were classified as follows:

Classification	Quantitity
laboratory CAL	15
tutorial CAL	1
laboratory/tutorial CAL	4
CML	9
admininstration	4
other	2

Towards the end of the programme 13 880 students and 690 staff members from 172 institutes were either directly or indirectly involved with the programme, and Hooper (the director of NDPCAL) stated that 70% of the projects had been institutionalised (institutionalisation was defined as "the successful takeover of working CAL and CML systems on to the local budgets on a permanent basis after the period of external funding runs out.") [Hooper, 1977].

2

THE CIRCUIT TUTOR SYSTEM

2.1 Goals of Circuit Tutor

Circuit Tutor, the subject of this dissertation, has two main goals, namely:

1. to develop a computer program suitable for the teaching, demonstrating, or testing of a whole class of electrical/electronic circuits; and

2. that the program should be expandable and maintainable, to simplify the task of adding circuit models to Circuit Tutor.

2.2 Selection of a CAL method for Circuit Tutor

Circuit Tutor is concerned with the education of Electrical Engineering students at undergraduate level. It is therefore concerned with the achievement of higher level cognitive goals which are attainable by using SIMULATION (see section 1.3.1). Although simulation can be combined with CAI, it was decided not to include a tutoring element for the following reasons :

- the tutoring style built into Circuit Tutor may conflict with that of some tutors. Science teachers (which includes Electrical Engineering tutors) do not always have a high opinion of CAI. It was therefore decided to allow the tutor to use Circuit Tutor using his/her own tutoring style;

- it would be more difficult to expand Circuit Tutor. When a circuit is added (installed), the installer would have to specify a tutoring algorithm for the circuit;

- Circuit Tutor is not aimed at reducing the requirement for a human tutor. It is intended to supplement/enhance existing educational resources, not replace them;

- Circuit Tutor is intended to allow the user to freely explore the properties of the circuit by himself/herself, thereby simulating a laboratory environment rather than a classroom;

- lack of confidence in tutorial CAL. Little evidence could be found in the literature to suggest that CAI has achieved much success in science education at undergraduate level. This author's personal opinion of CAI is that it is not very suitable for the teaching of high level cognitive skills. On tutorial CAL, Hooper, the director of NDPCAL, remarks:

  > "Thus the issue of whether CAL is the most cost effective means of, for example, teaching decision making or individualised instruction has to be confronted. Generally speaking, CAL has not proved itself to be cost-effective in individualising instruction." [Hooper, 1978, p. 3)

Any simulation program can be logically divided into two parts:

- the one part models the behaviour of the real circuit; and
- the remainder of the program provides the man-machine interface (MMI) and other run-time support (e.g. the scheduling of the the program).

To accomplish expandability of the Circuit Tutor, i.e. to allow a circuit models to be added with ease, the MMI part of Circuit Tutor was logically separated from the rest of the program. Each circuit model is implemented as a Turbo Pascal unit (see figure 2-1). These units are described, in detail, in the Designer's Reference Manual.

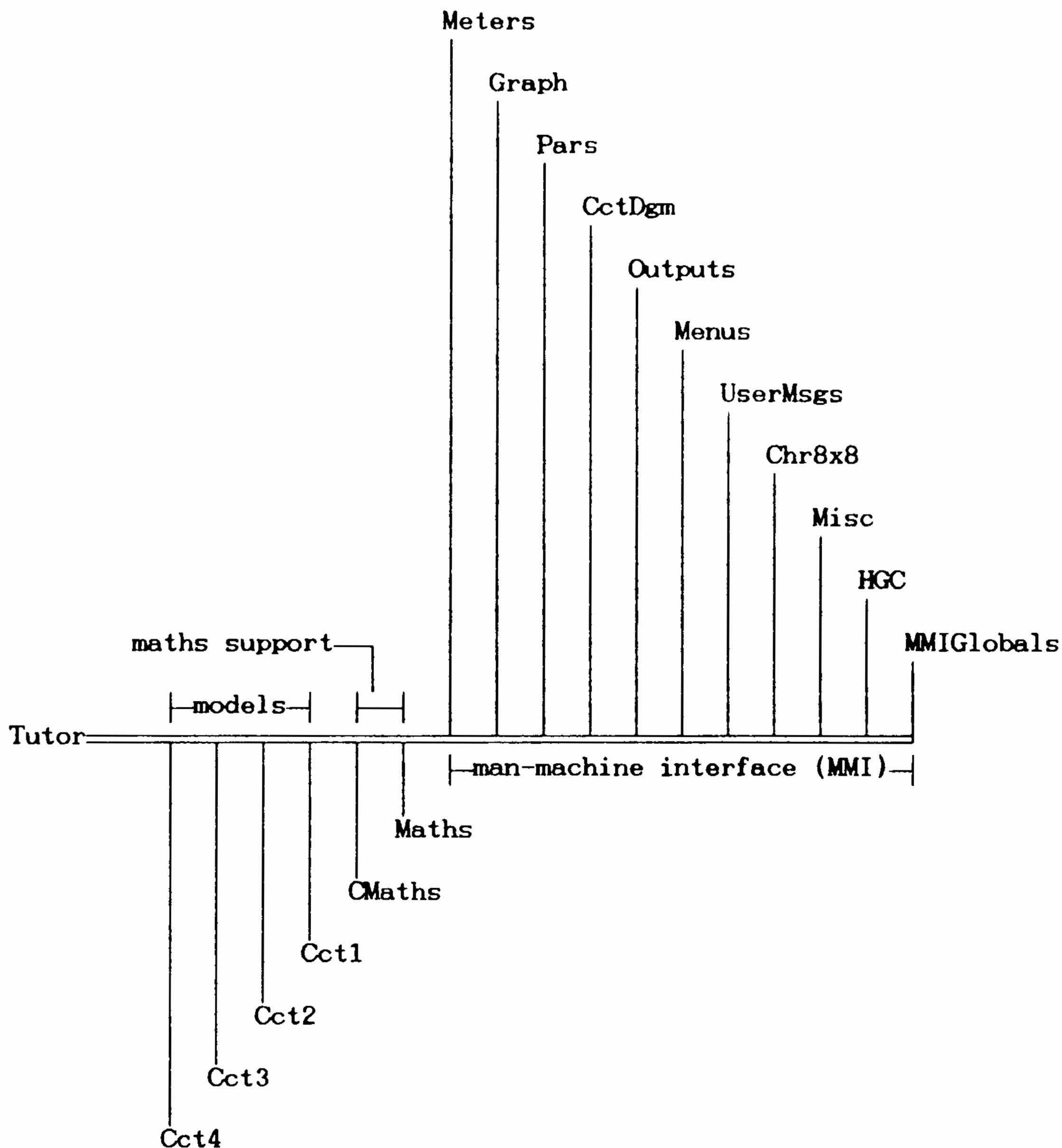


Figure 2-1. Modular software construction - MMI and models separated. Turbo Pascal v.4 unit identifiers indicated.

2.3 Choice of hardware

Circuit Tutor was implemented for the IBM PC microcomputer for the following reasons:

- this is the de facto standard in microcomputers;
- most South African universities, including UCT, have acquired large numbers of IBM PC or compatible microcomputers;
- IBM PC compatible microcomputers offers adequate performance at reasonable cost even with the poor exchange rate of the Rand;
- many students and lecturers have purchased IBM PC compatible microcomputers, and can therefore use Circuit Tutor at home;

One of the biggest problems with the IBM PC is the lack of a graphics standard. Several graphics adaptors are used for the IBM PC; including the CGA, MCGA, EGA, VGA, Hercules, and Olivetti M24. Although software can be written to run on all of the graphics standards the task of the programmer is complicated - Circuit Tutor was therefore implemented for the IBM PC with a Hercules Graphics Card installed. The Hercules Graphics Card was selected as it provides:

- acceptable resolution;
- is in wide-spread use;
- it is economically priced, providing a good resolution-price ration.

2.4 General requirements of a man-machine interface

Any man-machine interface making use of menus should meet the following minimum requirements to be acceptable. Some of these requirements were taken from Huckle (1981), p.108:

1. Extensive on-line help to be available at all times.

2. Error conditions to result in the display of error messages which suggest corrective action to be taken.

3. Special attention to be paid to the menu item identifiers to ensure clarity.

4. The use of computer jargon to be avoided.

5. Information to be properly spaced.

6. Menus should not contain an excessive number of items.

7. Data output must be presented in a manner that makes it easy to see, visualize and comprehend.

8. Data input required by the system must be simple and unambiguous.

2.5 Overview of Circuit Tutor features

One of the objectives of simulation is to motivate students. This can be achieved if the MMI makes use of interactive high-resolution graphics. Circuit Tutor provides the user with all the facilities necessary for selecting and testing a circuit. The program is fully menu driven with extensive on-line help.

A session normally commences with the user selecting a circuit from a list of available circuits (by means of the Circuit Tutor menu). After selection, a circuit diagram is drawn for the circuit, and the circuit parameters and their default values displayed in CIRCUIT PARAMETERS window.

The user can then modify the parameters by means of:

- the + and - keys for incremental changes. The circuit parameter is first selected by moving the CIRCUIT PARAMETERS window cursor to the parameter, and then pressing + to increment the parameter or - to decrement the parameter. Alternatively the user can temporarily remove the menu and display a cursor on the circuit diagram. This cursor can be moved to the component associated with the parameter to be modified, and the + and - keys used as before;

- the menu for other changes. For each circuit parameter the user may modify the minimum, maximum, step (incremental), or present (current) value. The menu for any circuit parameter can be selected (1) in the normal way: i.e. by selecting the parameter from the PARAMETERS menu; or (2) at any other time by pressing a function key to display the menu for the parameter selected in the CIRCUIT PARAMETERS window.

After a parameter is selected Circuit Tutor will recalculate the outputs for the circuit (this facility can be disabled). The new circuit output values can be monitored by (1) means of the CIRCUIT OUTPUTS window; (2) the meters; (3) the graph; or (4) the lin/lin graph.

2.6 Expanding Circuit Tutor

To expand Circuit Tutor i.e. to add a circuit to Circuit Tutor, entails:

1. Adding the mathematical model for the circuit. This is accomplished by creating a Turbo Pascal unit (e.g. Cct2) to solve the parameter-output relationship for the circuit. Two units, Maths and CMaths, which contain routines for matrix manipulation (including Gauss-Jordan elimination of complex matrices) are provided for rapid prototyping of circuit models.

2. Creating a circuit diagram for the circuit by using the Circuit Draw utility program. Circuit Draw allows the programmer to interactively draw a circuit diagram, add text to the circuit diagram, define new circuit diagram, and much more. The Circuit Draw utility is described, in detail, in the User's Reference Manual and the Designer's Reference Manual.

3. Modifying the MMI part of Circuit Tutor and then recompile the entire program. As the MMI part is logically separate from the circuit models only minor modifications are necessary.

Two examples of units to model the parameter-output relationship for a circuit will now be presented to demonstrate the ease with which Circuit Tutor can be expanded. Both of these unit are included with the Circuit Tutor package. The first unit, Cct2, implements circuit number 2 - a simple capacitor charging circuit. It can be observed that the programmer adding a circuit model need not be concerned with the MMI part of the program. The programmer simply has to use

- the *DefCct* procedure to tell Circuit Tutor the name of the circuit and where the circuit diagram file (created with the Circuit Draw utility) can be found;

- the *DefParameter* to tell Circuit Tutor which parameters are modifiable by the user; and

- the *DefOutput* procedure to tell Circuit Tutor which outputs the user may observe.

The *CctResult* variable is used to signal to Circuit Tutor whether or not the calculated outputs are valid. If the outputs are valid then CctResult=0, and Circuit Tutor will display the outputs; if CctResult is set to another value an error message will be displayed to the user, for example if CctResult=2 then message number 2 (from the Circuit Tutor messages file TUTOR.MSG) will be displayed to the user, suggesting corrective action(s) to be taken.

The second example, Cct4, which implements a sixth order Chebychev filter, makes use of the Gauss-Jordan elimination procedure provided by the CMaths unit.

```
(* UNIT TO IMPLEMENT CIRCUIT NO. 2 - CAPACITOR CHARGING CIRCUIT *)

unit Cct2;
interface
uses
  MmiGlobals,
  CctDgm,
  Outputs,
  Pars;
procedure SolveCct2(option: CctSolveOption);
implementation
var
  E,R,C,t,i,Vc :real;
procedure SolveCct2{(option: CctSolveOption)};
begin
  case Option of
    IdentifyCct:
      DefCct('2. Capacitor charging circuit','cct2.dgm');
    InitCct:
    begin
      DefParameter(t,'t',    0.0,    0.0,   100.0,    1.0,  5,5);
      DefParameter(R,'R', 1.0e+6, 0.1e+6,   10e+6, 0.1e+6,  7,5);
      DefParameter(C,'C',   1e-6,   1e-6,  100e-6,   1e-6,  9,6);
      DefParameter(E,'E',    1.0,    1.0,    10.0,    1.0,  3,6);
      DefOutput(t,'t',0.0);
      DefOutput(i,'i',0.0);
      DefOutput(Vc,'Vc',0.0);
    end;
    SolveCct:
      If C<>0 then
      begin
        CctResult:=0;                    (* OK - NO ERRORS *)
        i := (E/R) * exp(-t/(R*C));
        Vc:= E - i*R;
      end
      else
        CctResult:=1                     (* ERROR NO. 1 *)
  end{case}
end{SolveCct2};
end{Cct2}.
```

```
(* UNIT TO IMPLEMENT CIRCUIT NO. 4 - 6TH ORDER CHEBYCHEV LOW-PASS FILTER *)

unit Cct4;
interface
uses
  CMaths,
  MmiGlobals,
  Misc,
  CctDgm,
  Outputs,
  Pars;
procedure SolveCct4(Option: CctSolveOption);
implementation
var
  Vs, w, f, LogF, Rs, Gs, Rout, Gout,
  C1, C3, C5, L2, L4, L6,
  Yc1, Yc3, Yc5, YL2, YL4, YL6,
  v1, v2, v3, Vout, G: real;
  A: CBigMatrix;
  GJerror: Boolean;      { Calculated outputs are invalid !! }
procedure SolveCct4{(Option: CctSolveOption)};
begin
  case Option of
    IdentifyCct:
      DefCct('4. Chebyshev low-pass filter (6th order).','cct4.dgm');
    InitCct:
    begin
      DefParameter(LogF,'log(f)',     3.0,     0.10,     10.0,     0.1,  2,5);
      DefParameter(Vs   ,'Vs',    1.000e+0,  1.0e+0,    10e+0,     0.1,  3,5);
      DefParameter(Rout,'Rout',10.000e+3,  1.0e+3,   100e+3, 1000.0, 13,5);
      DefParameter(Rs   ,'Rs',   10.000e+3,  1.0e+3,   100e+3, 1000.0,  4,3);
      DefParameter(C1   ,'C1',    2.588e-9,  0.1e-9,   100e-9, 0.1e-9,  6,5);
      DefParameter(C3   ,'C3',    9.660e-9,  0.1e-9,   100e-9, 0.1e-9,  8,5);
      DefParameter(C5   ,'C5',    7.071e-9,  0.1e-9,   100e-9, 0.1e-9, 10,5);
      DefParameter(L2   ,'L2', 707.100e-3,  0.1e-3, 1000e-3,    1e-3,  7,3);
      DefParameter(L4   ,'L4', 966.000e-3,  0.1e-3, 1000e-3,    1e-3,  9,3);
      DefParameter(L6   ,'L6', 258.800e-3,  0.1e-3, 1000e-3,    1e-3, 11,3);
      DefOutput(LogF,'log(f)',3.0);
      DefOutput(f ,'f', 0.0);
      DefOutput(w ,'w', 0.0);
      DefOutput(v1,'v1',0.0);
      DefOutput(v2,'v2',0.0);
      DefOutput(v3,'v3',0.0);
      DefOutput(Vout,'Vout',0.0);
      DefOutput(G, 'G',0.0);
    end;
```

```
    SolveCct:
    begin
      f   := Exponent(10,LogF); w:= 2*Pi*f;
      Gs  := 1/Rs; Gout:=1/Rout;
      YL2:=-1/(w*L2); YL4:=-1/(w*L4); YL6:=-1/(w*L6);
      Yc1:= w*C1; Yc3:= w*C3; Yc5:= w*C5;
      (* SET UP MATRIX ELEMENTS *)
      CAss(A[1,1],Gs,Yc1+YL2);      (* Re(A[1,1]):=Gs ; Im(A[1,1]):=Yc1+YL2 *)
      CAss(A[1,2],0,-YL2);          (* Re(A[1,2]):=0  ; Im(A[1,2]):=-YL2    *)
      CAss(A[1,3],0,0);
      CAss(A[1,4],0,0);
      CAss(A[2,1],0,-YL2);
      CAss(A[2,2],0,Yc3+YL2+YL4);
      CAss(A[2,3],0,-YL4);
      CAss(A[2,4],0,0);
      CAss(A[3,1],0,0);
      CAss(A[3,2],0,-YL4);
      CAss(A[3,3],0,Yc5+YL4+YL6);
      CAss(A[3,4],0,-YL6);
      CAss(A[4,1],0,0);
      CAss(A[4,2],0,0);
      CAss(A[4,3],0,-YL6);
      CAss(A[4,4],Gout,YL6);
      CAss(A[1,5],-Vs*Gs,0);
      CAss(A[2,5],0,0);
      CAss(A[3,5],0,0);
      CAss(A[4,5],0,0);
      CGaussJordan(A,4,5,GJerror);  (* SOLVE MATRIX USING GAUSS-JORDON ELL. *)
      if GJerror then
        CctResult:=1                (* ERROR NO. 1 *)
      else
      begin
        CctResult:=0;               (* OK - NO ERRORS *)
        v1   := CAbs(A[1,5]);
        v2   := CAbs(A[2,5]);
        v3   := CAbs(A[3,5]);
        Vout := CAbs(A[4,5]);
        G    := 20*Log(Vout/Vs)
      end
    end
  end{case}
end{SolveCct4};

end{Cct4}.
```

Although four circuit models are included with Circuit Tutor, it was not the intention to provide a whole library of circuit models. Rather the objective was to develop a general purpose man-machine interface to be used for demonstrating a large number of electrical/electronic circuits. The design therefore concentrated on the quality of the man-machine interface. Any circuit can be added to Circuit Tutor as no assumptions are made regarding the functionality of the circuit. Provided a circuit has user-modifiable parameters, and observable outputs, it can be added, with very little effort.

To aid the programmer to add a circuit the following tools are provided:

- the *DefCct*, *DefParameter*, and *DefOutput* procedures. The procedures permits the user to easily identify (declare) the circuit to Circuit Tutor;
- the *CctResult* variable to facilitate exception handling;
- the Circuit Draw utility to interactively draw a circuit diagram for a circuit; and
- maths support provided by the *Maths* and *CMaths* units.

In addition Turbo Pascal version 4 is used for implementing the design as it supports a modular design, thereby separating the man-machine interface part of the program from the units to implement the circuit model. In addition, most students and lecturers are familiar with Pascal.

At run-time Circuit Tutor provides a student with all the facilities to interactively view or modify circuit parameters, whilst observing the corresponding outputs. Circuit parameters can be modified incrementally or by means of the menu, for other changes. For each circuit parameter the user may modify the default minimum and maximum values, although this should be done with caution as the minimum and maximum values serve mainly to determine the permissible range of values for which valid outputs can be solved.

Circuit outputs can be monitored by means of the CIRCUIT OUTPUT window, the menu, the meters, the graph or any combination of these methods.

During the development of Circuit Tutor prototype copies were evaluated on an informal basis by engineering students; engineering colleagues; and my supervisor, Mr Greene. During these evaluations only circuit number 1 was implemented. The main focus of these informal evaluations was however, not to to provide an impressive circuit, but rather to improve the man-machine interface. Several modifications were implemented, mainly to the menu structure, menu item identifiers, and the on-line help facility. Almost all the comments received during these informal evaluations were positive, and indicated a high level of acceptance. The simulated analogue meters appeared to appeal to most of the students, as this gave them a 'feel' for the circuit (a simple transistor circuit implemented using the static Ebers-Moll model).

As Circuit Tutor can be classified as a pure laboratory CAL package, it can be used in many ways by the lecturer - its application was purposely left open-ended for flexibility. It may be used for demonstrating circuits during lectures; as a laboratory-substitute during a normal tutorial; or it can be used to teach students about modelling. The educational worth of Circuit Tutor - and the evaluation to determine its educational worth - therefore depends on the way that the tutor decides to use Circuit Tutor.

Although the informal evaluation of Circuit Tutor proved valuable in the design stages, a more formal evaluation would have to be carried to determine the level of acceptance, and the attitudes of users towards Circuit Tutor.

A formal evaluation of any program, especially a CAL program, is no trivial task if worthwhile results are to be obtained. In fact it was felt that an formal evaluation of Circuit Tutor would be a time-consuming activity and therefore beyond the scope of this dissertation.

In conclusion, it is felt that Circuit Tutor achieved the goals relating to functionality, expandability and maintainability of the program. Informal evaluations indicated that users had positive attitudes towards Circuit Tutor. These informal evaluations, would however have to be followed up with a more formal evaluation to determine the educational worth of the program. Such an evaluation would however, have to be undertaken after it has been decided in what context Circuit Tutor was to be used. Such an evaluation could possibly be undertaken as an undergraduate dissertation.

2.8 Possible improvements and enhancements to Circuit Tutor

1. Write versions for other graphics adaptors. The current version 1.04/hgc of Circuit Tutor will only run on the Hercules Graphics Card.

2. Convert to Turbo Pascal version 5 when available. As Turbo Pascal support overlay procedures SolveCct<x> procedures can be converted to overlay procedures, thereby allowing a virtually unlimited number of circuits to be added to Circuit Tutor.

3. Convert to Modula-2 to enable procedures in the Maths and CMaths units to pass funtions as parameters.

4. Extend the graph facility to give the user the option of specifying either log/lin or lin/lin axis for the graph.

5. Give the user the option of exchanging the three meters for a second graph.

1. Alty, J.L., The Impact of Microtechnology : A Case for Reassessing the Roles of Computers in Learning. In Computer Assisted Learning - Selected Papers from the CAL 81 Symposium (Edited by P.R. Smith). Pergamon Press, Oxford, 1981.

2. Beech, G., Current Aspects of Program Exchange - Costs and Benefits. In Computer Assisted Learning in Science Education (Edited by G. Beech). Pergamon Press, Oxford, 1978.

3. Beech, G., Computer Based Learning : Practical Microcomputer Methods. Sigma Technical Press, Cheshire, 1983.

4. Bork, A., Learning with Computers. Digital Press, Bedford, Mass., 1981.

5. Broster, P.R., Microcomputers and the Teaching of Chemistry in English Schools. M.Sc. dissertation, University of York, December, 1981.

6. Cousins, D.A., Design of a Computerised Electronics Tutoring System. B.Sc. thesis, U.C.T., 1987.

7. Ellington, H.I., et al., Games and Simulations in Science Education. Kogan Page, London, 1981.

8. Geerdts, C.D., Softwire: An Interactive, Computer-Based System for Enhancing Learning in Electrical Engineering, Using Simulation. M.Sc. dissertation, U.C.T., September 1987.

9. Hinton, T., Computer Assisted Learning in Physics. In Computer Assisted Learning in Science Education (Edited by G. Beech). Pergamon Press, Oxford, 1978.

10. Hooper, R., The National Programme in Computer Assisted Learning : Final Report of the Director. Council for Educational Technology, 1977.

11. Hooper, R., Computers in Science Teaching - An introduction. In Computer Assisted Learning in Science Education (Edited by G. Beech). Pergamon Press, Oxford, 1978.

12. Huckle, B., The Man-Machine Interface : Guidelines for the Design of the End-User/System Conversation. Savant Research Studies, Lancashire, 1981.

13. Jenkin, J.M., Some Principles of Screen Design and Software for their Support. In Computer Assisted Learning - Selected Papers from the CAL 81 Symposium (Edited by P.R. Smith). Pergamon Press, Oxford, 1981.

14. Joachim, P.E. and Wedekind, Computer Aided Model Building and CAL. In Computer Assisted Learning - Selected Papers from the CAL 81 Symposium (Edited by P.R. Smith). Pergamon Press, Oxford, 1981.

15. Maddison, A., Microcomputers in the Classroom. Hodder and Stoughton, London, 1982.

16. McKenzie, J., Interactive Computer Graphics for Undergraduate Science Teaching. In Computer Assisted Learning in Science Education (Edited by G. Beech). Pergamon Press, Oxford, 1978.

17. Plomp, T. (ed.), et al., Computer-Assisted Learning for Europe. Proc. Conference of the European Commission on The Development of Educational Software. Elsevier Science Publishers, Amsterdam, 1987.

18. Sloan, D. (ed.), The Computer in Education : A Critical Perspective. Teachers College Press, New York, 1984.

19. Suppes, P., The Future of Computers in Education. In The Computer in the School: Tutor, Tool, Tutee. Teachers College Press, New York, 1980.

19. Taylor, R.P.(ed.), The Computer in the School: Tutor, Tool, Tutee. Teachers College Press, New York, 1980.

Part 2

Circuit Tutor
User's Reference Manual

Version 1.04/hgc
October 1988

1 INTRODUCTION 2-1

1.1 What is Circuit Tutor 2-1
1.2 Structure of this document 2-1

2 USING CIRCUIT TUTOR 2-3

2.1 Hardware requirements 2-3
2.2 Distribution diskette files 2-3
2.3 Starting Circuit Tutor 2-4
2.4 Selecting a circuit 2-4
2.5 Circuit Tutor display 2-7
2.6 Modifying circuit parameters 2-8
2.7 Monitoring the circuit outputs 2-11
2.7.1 Using the meters 2-11
2.7.2 Using the graph 2-13
2.8 Circuit Tutor menu reference 2-13

3 USING THE CIRCUIT DRAW UTILITY 2-18

3.1 System requirements 2-18
3.2 The Circuit Draw distribution diskette 2-18
3.3 Starting Circuit Draw 2-19
3.4 Drawing diagram symbols 2-21
3.5 Drawing diagram text 2-23
3.6 Defining symbols (graphics font editor) 2-23
3.7 Circuit Draw menu reference 2-26

4 ADDING A CIRCUIT TO CIRCUIT TUTOR 2-28

4.1 Creating the unit 2-28
4.1.1 IdentifyCct statement 2-29
4.1.2 InitCct statement 2-29
4.1.3 SolveCct statement 2-30
4.2 Modifying Circuit Tutor 2-33

APPENDIX A. CIRCUIT TUTOR MESSAGES 2-34

APPENDIX B. CIRCUIT TUTOR INSTALLATION GUIDE 2-36

1

INTRODUCTION

1.1 What is Circuit Tutor

Circuit Tutor is in essence a man-machine interface. to facilitate the simulation of electrical circuits. Circuit Tutor is not a general purpose simulation package. Every circuit simulated by Circuit Tutor has to be added by writing a Turbo Pascal Unit to map the input/output relationship of the circuit. The circuit diagram for the circuit to be added is drawn using the utility Circuit Draw.

Circuit Tutor provides an integrated environment for testing a circuit. Apart from the input/output relationship of a circuit, no other programming effort is required to add a circuit.

A summary of Circuit Tutor features follow:

- Circuit Tutor can be used to teach basic electrical circuits to students. The user of the program need not be concerned with the inner workings of the program - i.e. how the circuit is solved.

- Circuit Tutor provides a simple user interface with a context sensitive help facility. All the functions of Circuit Tutor can be accessed by means of the menu.

- Circuit parameters can be modified quickly by means of the + and - keys. Parameters can be selected very quickly from the Parameters window or by direct positioning of a 'cursor' on the circuit diagram.

- Circuit outputs can be monitored by means of three simulated meters or by plotting outputs on the graph.

- It is extremely simple to add a circuit to Circuit Tutor. This is accomplished by first drawing a circuit diagram using the Circuit Draw utility, and then writing a Turbo Pascal Unit to solve the input/output relationship of the circuit.

1.2 Structure of this documentation

The documentation has been structured for the three categories of users namely:

- Users only interested in using Circuit Tutor need only read the first two chapters of this document.

2

USING CIRCUIT TUTOR

Circuit Tutor provides an integrated environment to test electrical/electronic circuits. The following features are provided:

- access to a circuit diagram for the circuit;
- a menu to manipulate all functions;
- quick modification of parameters by means of the Quick mode;
- three simulated analogue/digital meters;
- a graph to plot calculated circuit outputs.

2.1 Hardware requirements

Circuit Tutor has been developed for the IBM PC/XT or compatibles with the following hardware and software installed.

- At least 170 kilobyte free RAM.
- Hercules Graphics Card.
- MS-DOS version 3.xx

2.2 Distribution diskette files

Circuit Tutor and all it's associated files fit on one 5 ¼ inch floppy diskette. The Circuit Tutor files are:

- tutor.exe — Circuit Tutor executable file.
- tutor.msg — Welcome, help and error messages for Circuit Tutor.
- .dgm files — Contain the circuit diagrams e.g. cct1.dgm.
- error.msg — Graphix toolbox error messages.
- .fon files — Graphix toolbox character font files.

It is essential that all the Circuit Tutor files are located in the active directory before running Circuit Tutor. If any of the files are missing the program will display an error message and halt.

2.3 Starting Circuit Tutor

The first step is to boot up the personal computer with a copy of MS-DOS. It should be noted that the Circuit Tutor distribution diskette is not bootable. If you have a single or dual floppy computer it would be convenient to have Circuit Tutor on a bootable floppy diskette. This is accomplished by copying Circuit Tutor files to a floppy diskette that was formatted with the MS-DOS FORMAT/S command. If you have a hard disk computer it will be most convenient to create a directory, e.g. *CctTutor* for the hard disk and then copy all the Circuit Tutor files to the hard disk before executing Circuit Tutor.

As mentioned before, it is essential that all the Circuit Tutor files are located in the active directory before starting the program. For example, if the Circuit Tutor files are stored in the *CctTutor* directory of drive A, the following DOS commands will start the program:

a:
cd \CctTutor
tutor

If the hardware and software has been properly installed Circuit Tutor will display the following initialisation message:

Cct Tutor ver. 1.04
Developed by Leon Potgieter, 1988/89
Initialising - please wait ...

After a few seconds this message is replaced by the Circuit Tutor welcome message. The screen should now resemble figure 2-1. The welcome messages - as all messages - are cleared by pressing *Esc* or *F1*.

2.4 Selecting a circuit

The first step would normally be to select a circuit. A circuit is selected from the *SELECT CIRCUIT* menu (menu number 1.1). It should be noted that all menus except the main menu are numbered to indicate it's position in the menu hierarchical structure. The following step must be followed to select a circuit:

- From the main menu select the first item i.e. *CIRCUIT*. A menu menu item is selected using the up and down cursor and pressing *Enter*, although it is not necessary in this case as *CIRCUIT* is the first item. Selecting *CIRCUIT* has no affect other than displaying the *CIRCUIT* menu. Menu items which has this function is capitalised to distinguish them from other menu items.

- From the *CIRCUIT* menu select *SELECT CIRCUIT*. This causes the *SELECT CIRCUIT* menu - which contains a list of the available circuits - to be displayed.

CIRCUIT PARAMETERS

F9/F10 to select
+/- to incr/decr

CIRCUIT OUTPUTS

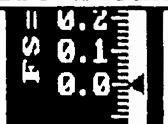
F7/F8 to page
21:44 410kB free

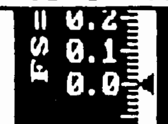
WELCOME TO CIRCUIT TUTOR

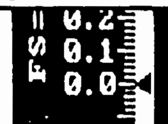
If you are a first time user please note
that help can be obtained by
pressing F1.

If you want to know more about Circuit
Tutor press F1 now or press the ESCAPE
(or ENTER) key to clear this message and
continue Circuit Tutor.

ESC/ENTER to clear.







** MAIN MENU **
CIRCUIT
METERS
GRAPH
quit program
press (F1) for help

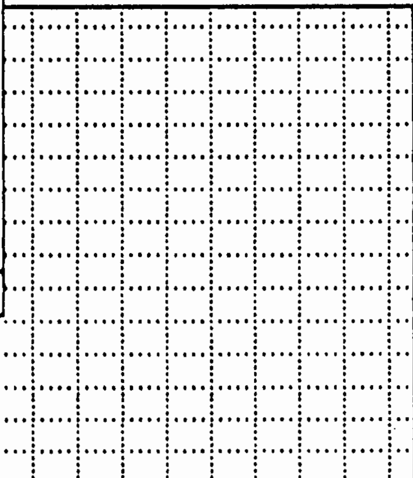
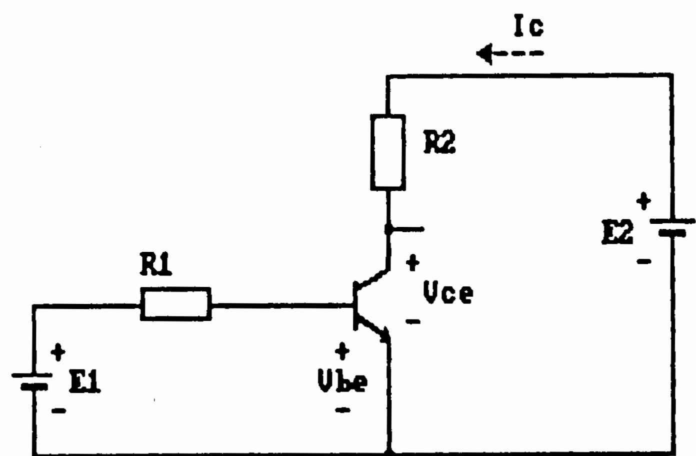


Figure 2-1. Circuit Tutor display after initialisation.

CIRCUIT PARAMETERS	
R1	1.000e+03
R2	8.300e+03
E1	0.000e+00
E2	10.00e+00
T	293.0e+00
Q:Icbo	1.000e-09
Q:Iebo	1.000e-09
Q:hRC	10.00e+00
Q:Eta	2.000e+00
Q:hFE	100.0e+00

F9/F10 to select
+/- to incr/decr

1. Simple transistor circuit



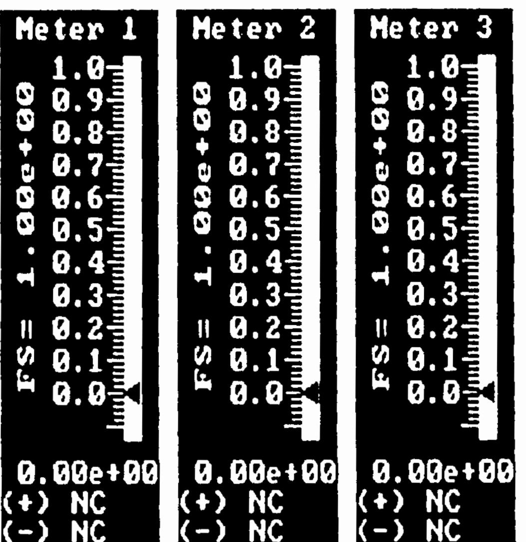
(1) CIRCUIT

SELECT CIRCUIT
PARAMETERS
OUTPUTS
quick mode <F3>
solve circuit <F2>
reset circuit
auto-solve : ON
press <F1> for help

CIRCUIT OUTPUTS

Vbe
Vce
Ib
Ic

F7/F8 to page
18:57 404kB free



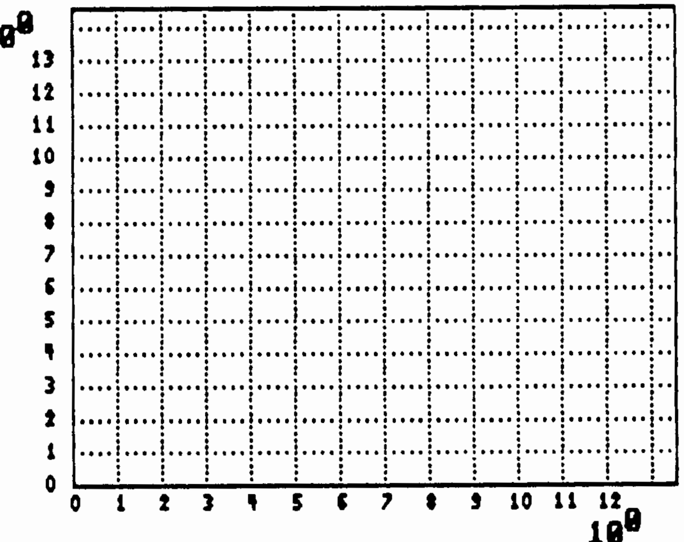


Figure 2-2. Circuit Tutor display after selecting circuit number one.

- Selecting a circuit from the *SELECT CIRCUIT* menu has the following effects:
  a) The circuit diagram for the circuit will be displayed 'under' the menu.
  b) The default values of circuit parameters will be displayed in the parameters window.
  c) The circuit output identifiers will be displayed in the outputs window.

Figure 2-2 is a screen dump of the display after circuit no.1 had been selected.

2.5 Circuit Tutor display

The display contains the following functional entities:

The menu

The menu situated in the top right-hand corner is used for accessing virtually all functions and facilities of Circuit Tutor. The menu contains three basic types of items:

(a) Those which cause another menu to be displayed. These items are referred to as *routers* as they route one to the next menu. To distinguish these items they are capitalised.
(b) Those which cause some task to be executed without requiring any input from the user.
(c) Those which require the user to provide some information in the field on the right-hand side of the item.
(d) Toggles. These are similar to those in (c) except that the field value can only assume a fixed number of predetermined values. When these items are selected the next field value is automatically entered.

A menu item is selected by first highlighting it, using the up and down arrow keys, and then pressing *Enter*. The previous menu is displayed by pressing *Esc*. All menus, except the main menu, have been numbered to give their position in the menu hierarchical structure which is four levels deep.

The circuit diagram

A circuit diagram is displayed 'under' the menu after selecting a circuit. The circuit diagram is loaded from the circuit diagram file for the selected circuit, e.g. *cct1.dgm* for circuit number 1.

The circuit parameters window

The Circuit Parameter values are displayed in the *CIRCUIT PARAMETERS* window after the circuit has been selected. It is possible to modify the parameter by first underlining it using the *F9* and *F10* keys (alternatively the *Home* and *End* keys can be used) and then stepping the value up and down using the + and - keys.

The circuit outputs window	The *CIRCUIT OUTPUTS* window contains a list of all the circuit outputs of the selected circuit. When the current values of the circuit outputs are valid they are displayed on the right-hand side of the circuit output identifiers. If the circuit output values are illegal i.e. no solution could be found, then no values are displayed next to the circuit output identifiers. Note that the circuit must be solved at least once for the outputs to be valid.
Three meters	Three simulated analogue meters are provided to monitor circuit output values. The meters are manipulated via the menu.
The graph	A graph to plot circuit output values.
Time	The time is displayed in 24-hour format. Seconds are not displayed.
Free memory	Displays the free memory in kilobyte.

2.6 Modifying circuit parameters

After the circuit has been selected you will want to modify the parameters of the circuit and observe the corresponding change in the circuit outputs. There are several ways of selecting and modifying the parameters of the selected circuit. They are:

- From the *PARAMETER <CctPar>* menu (see figure 2-3(a)). This menu has four fields to store the present, minimum, maximum and step value of the parameter. The present value of a parameter must always be between the limits set by minimum and maximum values. To set the present value to a value outside these limits it is necessary to first modify the limits. The default limits should however be modified with caution as they are set to filter out illegal parameter values. The step value is used for incrementing or decrementing the parameter value using the + and - keys (see next paragraph).

- Using the + and - keys. A parameter is first selected using the *F9* and *F10* keys to underline the parameter in the *CIRCUIT PARAMETERS* window. Note that the contents of this window will be scrolled so that all the parameters for the circuit can be accessed. The underlining is also referred to as the parameter window cursor. Alternatively the *Home* or *End* keys can be used to move the underlining. To increment the parameter value with the step value (see preceding paragraph) press the + key. To decrement the value press the - key.

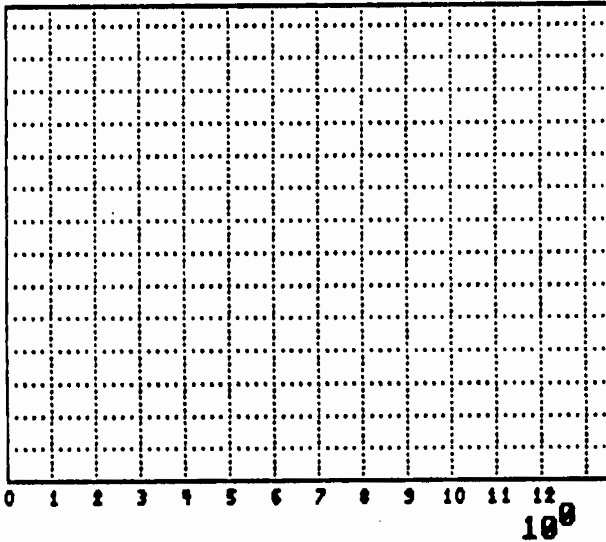
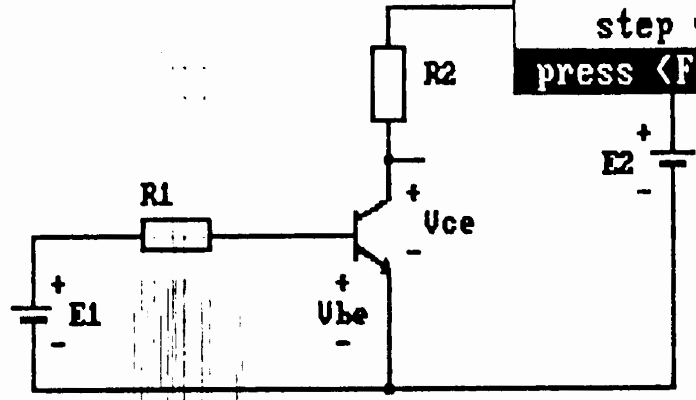
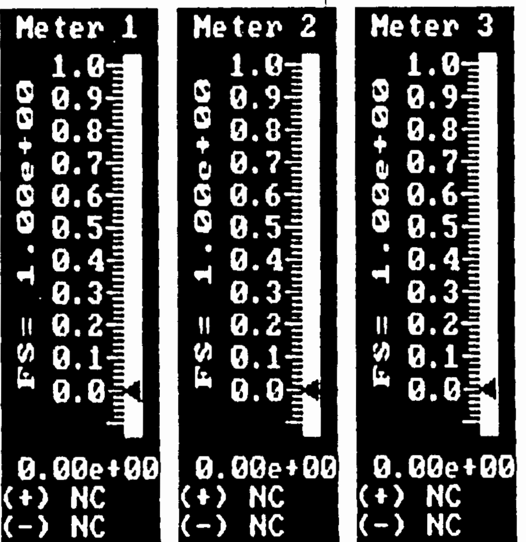
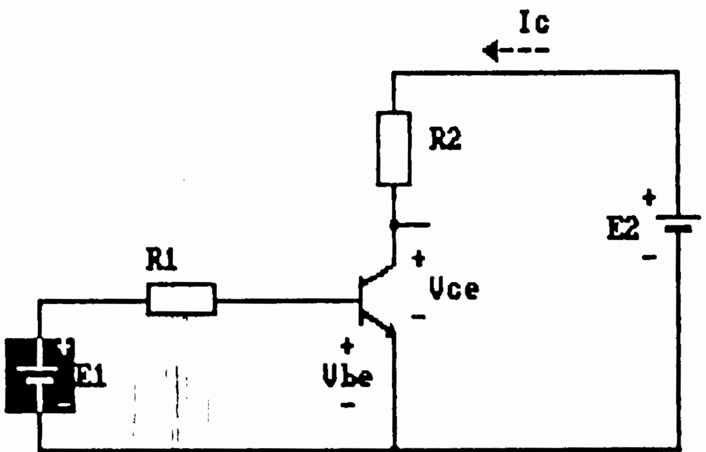




Figure 2-3(a). Modifying circuit parameters using the menu.



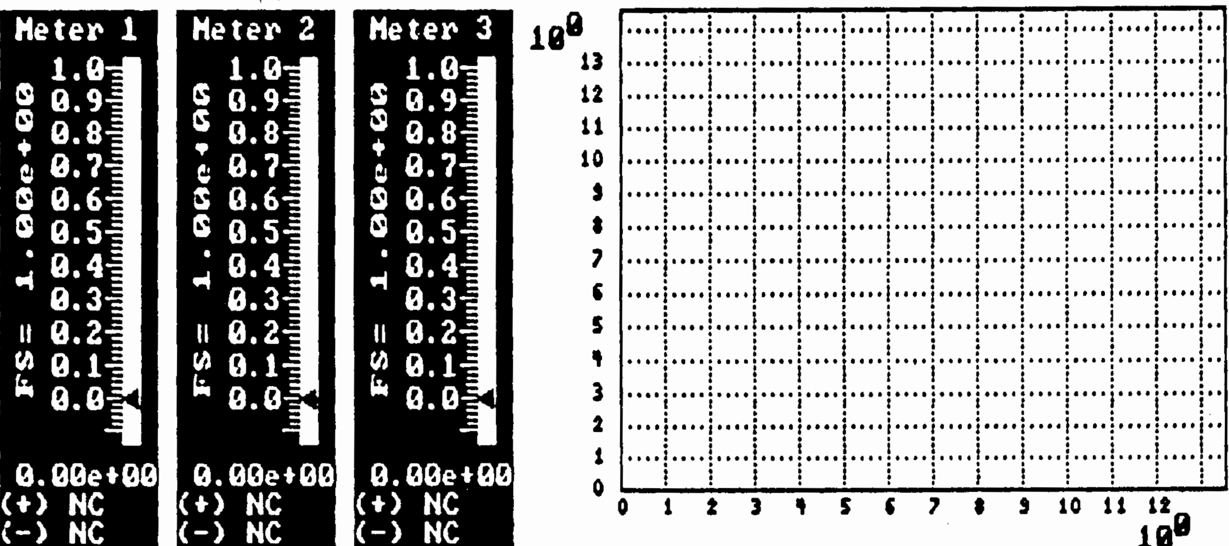


Figure 2-3(b). Modifying circuit parameters using + and - keys (in Quick mode).

Quick mode can be used for selecting the appropriate circuit parameter. Quick mode is selected from the *CIRCUIT* menu or by pressing *F3*. The menu is temporarily removed and a square block - called the circuit cursor - appears on the circuit. The circuit cursor is moved to the parameter to be modified using the arrow keys. This has the affect of moving the *CIRCUIT PARAMETERS* window cursor. Now the selected parameter (in the *CIRCUIT PARAMETERS* window) can be incremented or decremented as explained in the preceding paragraph. If the Parameters window cursor is moved by means of the *F9* and *F10* keys the circuit diagram cursor will also be moved to the parameter selected. It is possible to leave Quick mode in two ways: by pressing *Enter* or *Esc*. *Enter* has the effect of displaying the *PARAMETER <CctPar>* menu (menu numbers 1.2.x) for the parameter selected, whilst *Esc* will display the menu selected be Quick mode was entered. See figure 2-3(b) for a screen dump of Circuit Tutor whilst in Quick mode.

When a parameter is modified a new set of outputs will be calculated automatically and displayed in the *CIRCUIT OUTPUTS* window. It is possible to suppress the automatic solving of the circuit when parameter values are modified by changing the value of the item *auto-solve* in the *CIRCUIT* menu. When *auto-solve* is set *OFF* then the circuit is solved by selecting the *solve circuit* item from the same menu.

2.7 Monitoring the circuit outputs

The circuit outputs can be monitored in the following ways:

- Observing the values in the *CIRCUIT OUTPUTS* window. Page through the contents of this window using *F7* and *F8* or alternatively *PgUp* and *PgDn*.

- Using the simulated analogue meters.
- Using the graph.
- From the *OUTPUTS* menu.

2.7.1' Using the meters

Three simulated analogue meters are available to monitor circuit output values. The meters are set up using the menu. It is possible to assign two circuit outputs to each meter. The value displayed on the meter will be the difference between the two circuit outputs - this is analogous to a analogue meter with two inputs. To deassign a meter input it must be assigned to the default null input *NC*. The maximum deflection of the meter is referred to as the full-scale reading, and can be set to any real value.

The settings of each meter and a digital read-out of the displayed value is indicated on the meter.

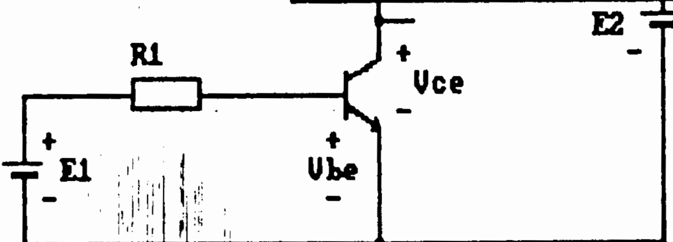
CIRCUIT PARAMETERS	
R1	1.000e+03
R2	8.300e+03
E1	200.0e-03
E2	10.00e+00
T	293.0e+00
Q:Icbo	1.000e-09
Q:Iebo	1.000e-09
Q:hRC	10.00e+00
Q:Eta	2.000e+00
Q:hFE	100.0e+00

F9/F10 to select
+/- to incr/decr

1. Simple transisto

(3.1) GRAPH X-AXIS

assign positive input to : Vbe
assign negative input to : NC
axis minimum value : 0.000e+00
increment per division : 1.000e+00
axis multiplier : 100.0e-03
press <F1> for help



CIRCUIT OUTPUTS	
Vbe	200.0e-03
Vce	9.996e+00
Ib	4.193e-09
Ic	520.3e-09

F7/F8 to page
19:25 404kB free

Meter 1	Meter 2	Meter 3
FS= 1.00e+00	FS= 10.0e+00	FS= 1.00e+00
200.e-03	10.0e+00	0.00e+00
(+) Vbe	(+) Vce	(+) NC
(-) NC	(-) NC	(-) NC

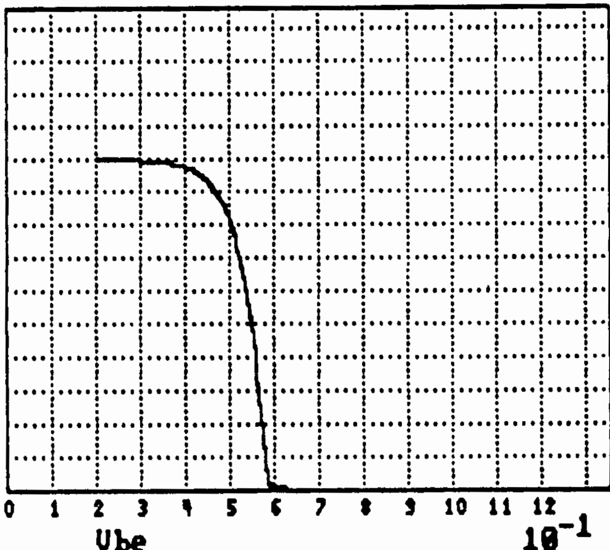


Figure 2-4. Monitoring the circuit outputs using the graph and meters.

illegal values for the circuit outputs and the full-scale value will be rejected. It should be noted that circuit output identifiers are case-sensitive e.g. HFE and hFE identify two different circuit outputs.

2.7.2 Using the graph

The graph is used to plot circuit output values. The graph menu is used for setting up the graph. Circuit outputs can the assigned to the x and y axis of the graph. Similar to the meters, two circuit output values can be assigned to each axis: the difference value will be taken as the input for the axis. The scale of the graph axis is set up by specifying the axis minimum and incremental values in addition to an axis multiplier. The logarithm of the multiplier must be an integer e.g. the following are legal multipliers: 0.1, 10, 1E-09.

The following features are available from the *GRAPH OPTIONS/FUNCTIONS* menu:

- Erase the contents of the graph. The settings for the axis remain unchanged.
- Setting *connect points* option on/off. When this option is selected plotted points are connected with line segments.
- Setting *mark points* option on/off. When this option is selected each plotted point will be marked with a diagonal cross.

2.8 Circuit Tutor menu reference

This section describes the menu in detail. The menu is arranged as a hierarchical structure where the main menu - or the root menu - is at the top of this structure e.g. menu function ***rase graph*** is is invoked by:

selecting *GRAPH* from the *MAIN* menu;
selecting *GRAPH OPTIONS/FUNCTIONS* from menu *(3) GRAPH* menu; and then
selecting *erase graph* from the *GRAPH OPTION/FUNCTIONS* menu.

The list of menus that the user has to go through in order to access the desired function is termed the menu path e.g. for *erase graph* the path is *GRAPH\GRAPH OPTIONS/FUNCTIONS* where the back slash separate menus.

It should be noted that if the menu contains more items than the size allows, then the menu contents can be scrolled up or down. This applies to the following menus/windows:

- SELECT CIRCUIT menu
- PARAMETERS menu
- OUTPUTS menu
- CIRCUIT OUTPUTS window
- CIRCUIT PARAMETERS window

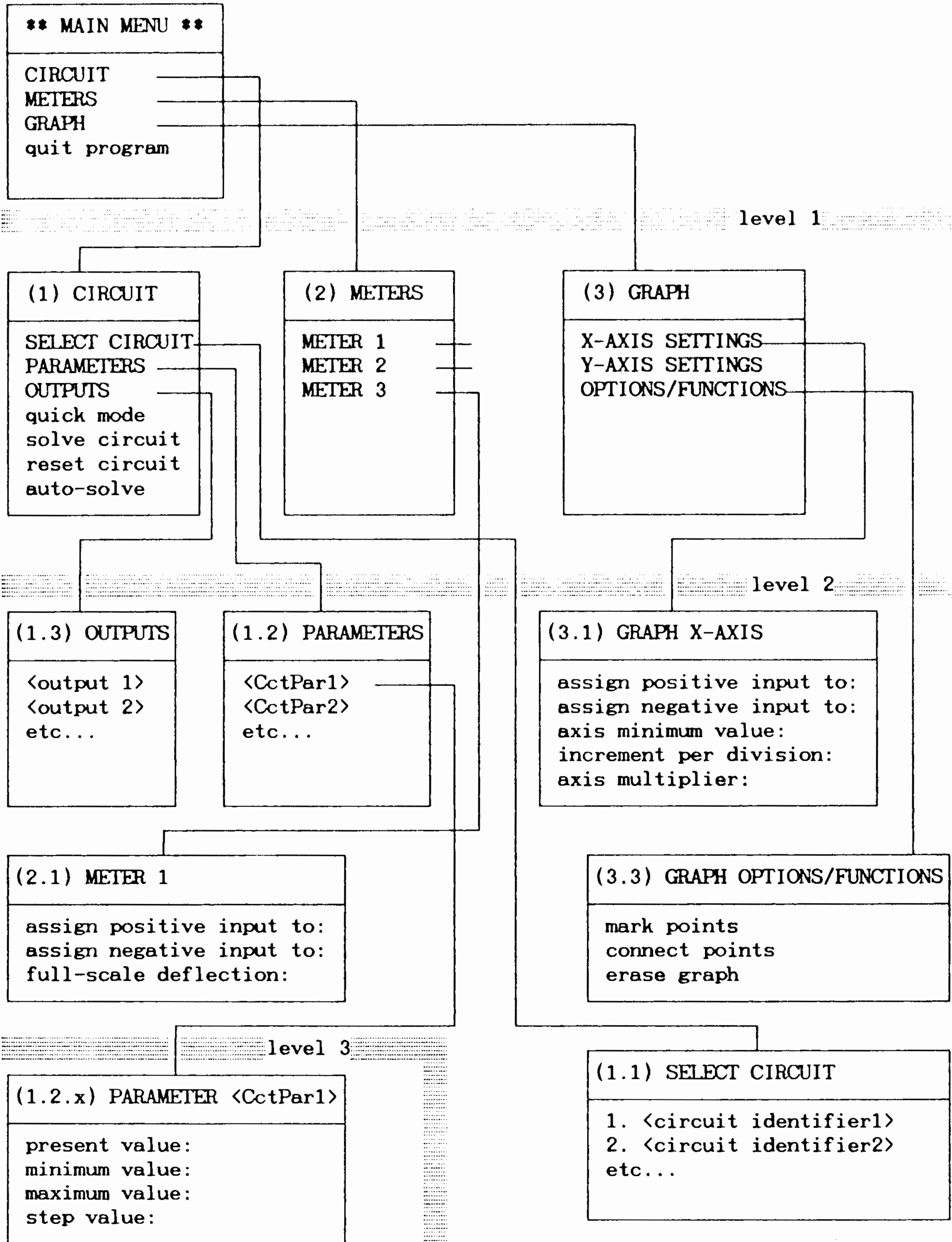


Figure 2-5. Structure of the Circuit Tutor menu

MAIN MENU

quit program	Halt the program and return to DOS.

CIRCUIT MENU — MENU NO. 1

quick mode	Puts the program in Quick mode. In Quick mode the menu is not displayed. Instead a cursor appears on the circuit diagram. By positioning this cursor over a circuit parameter, a parameter is selected. After a parameter is selected it can be incremented/ decremented with the + and - keys. To quit this mode press either *F3*, *Esc* or *Enter*. *Escape* and *F3* will restore the menu that was displayed prior to entering Quick mode, whilst Enter will display the menu for the parameter selected.
solve circuit	Solves the circuit and updates the outputs, the meters and the graph. When Auto-solve mode is on this function is executed automatically after modifying the circuit parameters.
reset circuit	Restore the default values of circuit parameters. The settings of the graph and meters are not affected.
auto solve	Toggles the Auto-solve mode on/off. When this mode is active the Solve Cct function (see above) will be invoked automatically when circuit parameters are modified.

SELECT CIRCUIT — MENU NO. 1.1

<circuit name>	The circuit diagram for the circuit is drawn 'under' the menu. The parameters and outputs are displayed in the *CIRCUIT PARAMETERS* and *CIRCUIT OUTPUTS* windows respectively. Graph and meter inputs are de-assigned, but other settings are not affected.

PARAMETER <Par>	MENU NO. 1.2.x
present value	View/modify the present value of the circuit parameter. The value must be between the minimum and maximum limits. Circuit parameters can also be modified with the + and - keys.
minimum value	View/modify the minimum value of the circuit parameter.
maximum value	View/modify the maximum value of the circuit parameter.
Step	View/modify the step value of the circuit parameter. The present value of the circuit parameter selected in the CIRCUIT PARAMETER window will be incremented or decremented with this value by means of the plus/minus + and - keys.

OUTPUTS	MENU NO. 1.3
<output>	View/modify the value of the circuit output.

METER x	MENU NO. 2.1
assign positive input to	View/modify meter positive input to circuit output assignment. The value assigned to this field must be a legal circuit output identifier. Note that circuit output identifiers are case-sensitive i.e. hfe and hFE identify two different circuit parameters. To de-assign meter input use the predefined circuit output *NC* (Not Connected).
assign negative input to	View/modify meter negative input to circuit output assignment.
full-scale deflection	View/modify the meter full-scale deflection. The value of this field is a real value.

GRAPH X(Y)-AXIS — MENU NO. 3.1, 3.2

assign positive input to	View/modify the graph positive input to circuit output assignment for the axis. The value of this menu field is a valid circuit output identifier.
assign negative input to	View/modify the graph negative input to circuit output assignment for the axis.
axis minimum value	View/modify the graph axis minimum value.
increment per division	View/modify the graph step value per division.
axis multiplier	View/modify the axis multiplier. The logarithm (to the base 10) of the multiplier must be an integer e.g. the following are legal multipliers: 0.01, 0.1 and 1E-09.

GRAPH OPTIONS/FUNCTIONS — MENU NO. 3.3

mark points	Toggles the graph *mark points* option on/off. When this option is selected the points plotted on the graph will be marked with an 'x'.
connect points	Toggles the graph *connect points* option on/off. When this option is selected the points on the graph are connected with line segments.
Erase graph	Erase the contents of the graph. Graph axis input assignments and settings are not affected.

3

USING THE CIRCUIT DRAW UTILITY

To add a diagram to the Circuit Tutor library of diagrams it is necessary to write a Turbo Pascal unit containing procedure to identify, initialise and solve the circuit. In addition a circuit diagram must be created using the Circuit Draw utility.

Circuit Draw offers the following features:

- menu-driven user interface;
- a circuit diagram can be loaded from or stored to a diskette file;
- the user can create any new symbol in the *DEFINE SYMBOL* mode;
- symbol tables (collection of circuit diagram symbols) may be modified; stored and retrieved from disk;
- the circuit diagram may contain both text and symbols;
- symbols can be manipulated in several ways to create new symbols;
- symbols can be rotated.

3.1 System requirements

To run Circuit Draw the following hardware and software is required:

- An IBM PC/XT compatible computer.
- A Hercules Graphics Card.
- 100 kilobyte free RAM
- MS-DOS ver 3.xx

3.2 Circuit Draw distribution diskette

Circuit Draw is distributed on a 5 ¼ inch floppy diskette which contain the following files (see appendix B):

draw.exe	Circuit Draw executable file
.pas files	Source code files for Circuit Draw.
.dgm files	Circuit diagram files. These files are produced by Circuit Draw. *cct1.dgm* is provided as an example.
error.msg	Graphix toolbox error messages.
.fon files	Graphix toolbox character font files.

.sym files	Symbol table files. The symbols from the file *default.sym* will be loaded when the program is started. The user may modify the default symbols file or create additional symbol files.

3.3 Starting Circuit Draw

Before starting Circuit Draw ensure that all the files on the distribution diskette are in the active directory. Then type

draw <Enter>

The following initialisation message should be displayed:

Cct Draw ver. 1.04
Developed by Leon Potgieter, 1988/89
Initialising - please wait ...

Circuit Draw has these three operating modes:

- *draw diagram symbols* mode (MODE 1);
- *draw diagram text* mode (MODE 2); and
- *define symbols* mode (MODE 3).

These modes are selected by pressing *F3*, *F5* and *F7* respectively. The display contains the following functional components (figure 3-1 gives the location of these components, whilst figures 3-2, 3-3 and 3-4 were obtained by dumping the screen contents during a typical Circuit Draw session).

Symbol table	The symbol table on the extreme left-hand side of the display contains 48 symbols. The user can add, delete or modify the symbol table symbols; store the symbol table to disk; or load a new symbol table from disk.
Menu	The contents of the menu depends on the mode selected. In *DRAW DIAGRAM SYMBOLS* mode symbols are transferred from the symbol table to the circuit diagram. In *DRAW DIAGRAM TEXT* mode text can be added to the circuit diagram. *DEFINE SYMBOLS* mode allows the user to define or modify symbols and to manage the symbol table.
Circuit diagram	A circuit diagram can be created, modified, stored and retrieved from disk.
Status line	The current state (mode) of the program and the status of the overwrite mode is displayed. This line is also used for dialogue with the user.
Symbol grid	In *DRAW DIAGRAM SYMBOLS* mode a grid is displayed in the place of the circuit diagram. Symbols can be transferred between the grid and the symbol. The grid is used to create a new symbol or modify an existing symbol.

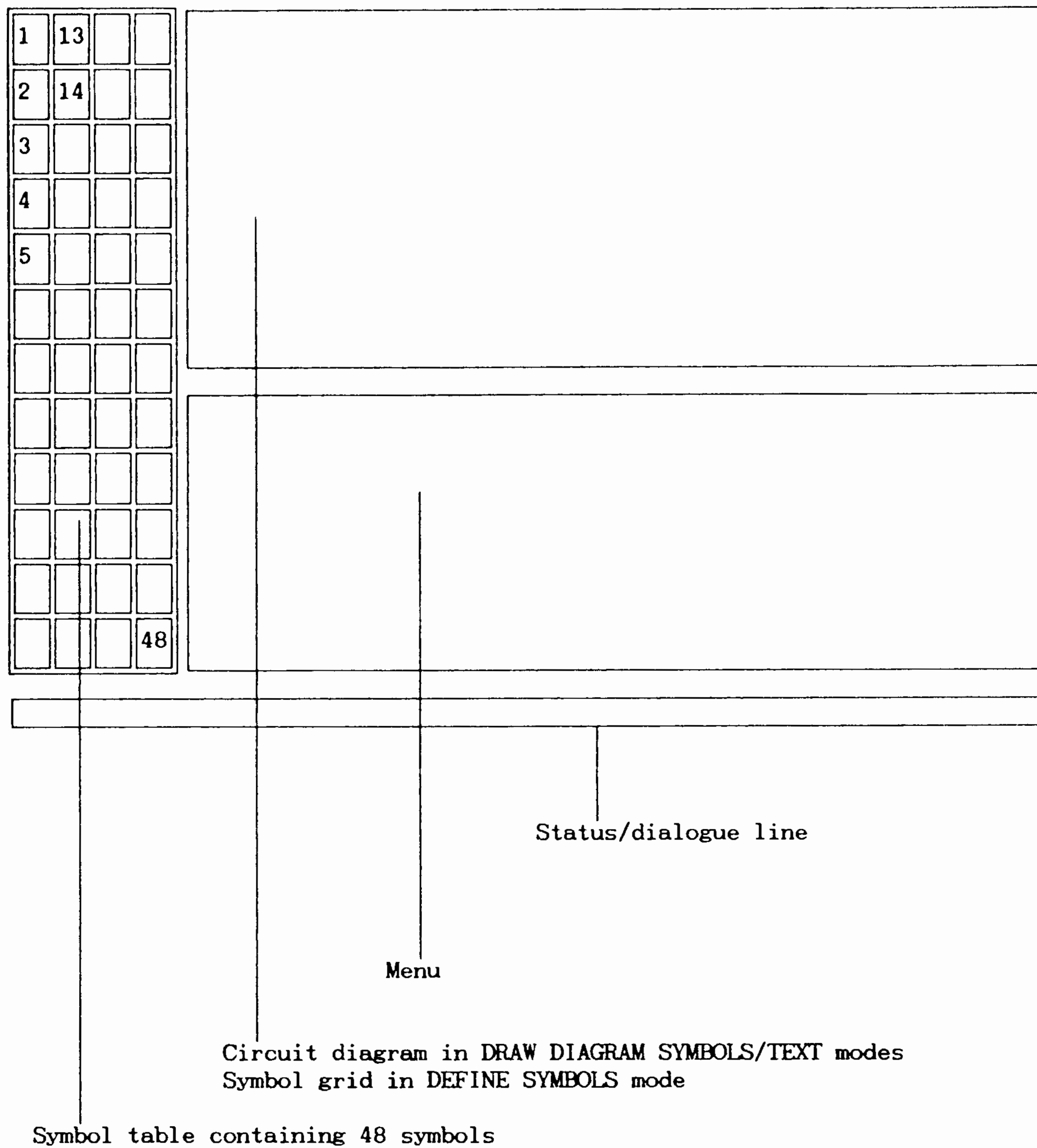


Figure 3-1. Lay-out of the Circuit Draw display.

3.4 Drawing diagram symbols

To draw symbols on the circuit diagram Circuit Draw must be in the *DRAW DIAGRAM SYMBOLS* (selected by pressing *F3*) mode. In this state a highlighted block the size of a circuit symbol appears on the circuit diagram (figure 3-2). This block will also be called the circuit symbol cursor. The up, down, left and right arrow keys are used to move this cursor around the circuit diagram. The circuit symbol cursor position is indicated in the bottom right corner of the menu.

A symbol is selected from the symbol table by highlighting the symbol. The *Home* and *End* keys are used to move the highlighted symbol table symbol - also referred to as the symbol table cursor.

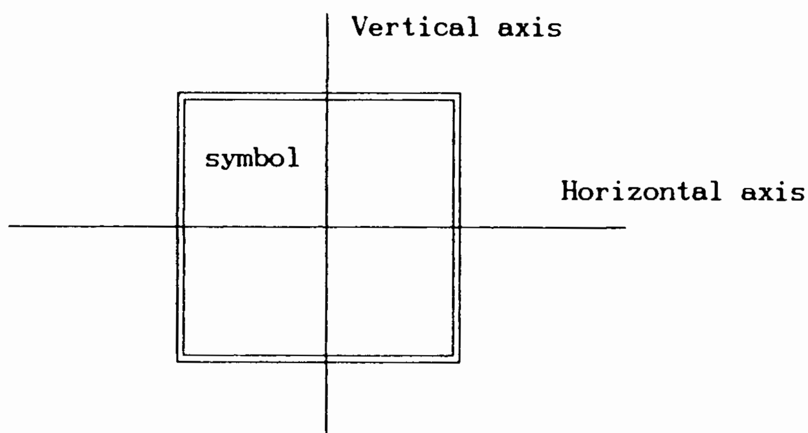
After selecting a symbol from the symbol table, it can be transferred to the circuit diagram by pressing *Enter*. If *OVERWRITE* (toggled with the *F4* key) is off, the symbol from the symbol table will be combined with the symbol 'under' the circuit symbol cursor.

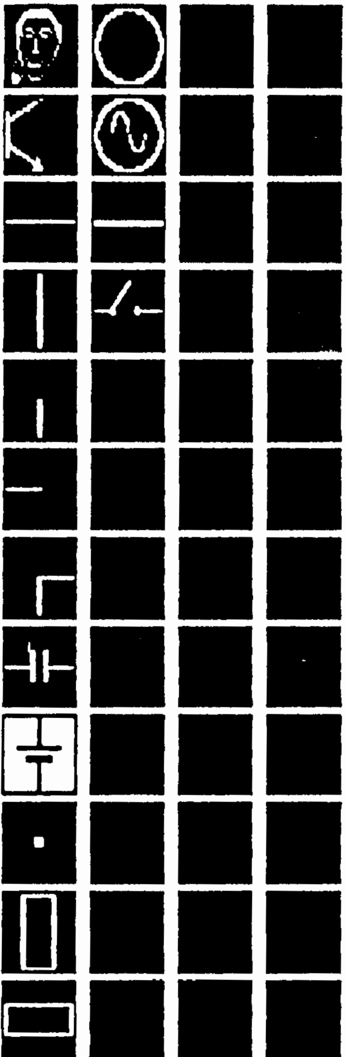
After a diagram is drawn by repeating the above steps, text can added using *DRAW DIAGRAM TEXT* mode. To save the diagram to a disk(ette) file simply press *F1*. The program will prompt the user for a file name. It is possible to append the file name with a path name if desired e.g. *c:\diagrams\cct3*. The file name extension must be omitted as Circuit Draw will add the extension *.dgm*.

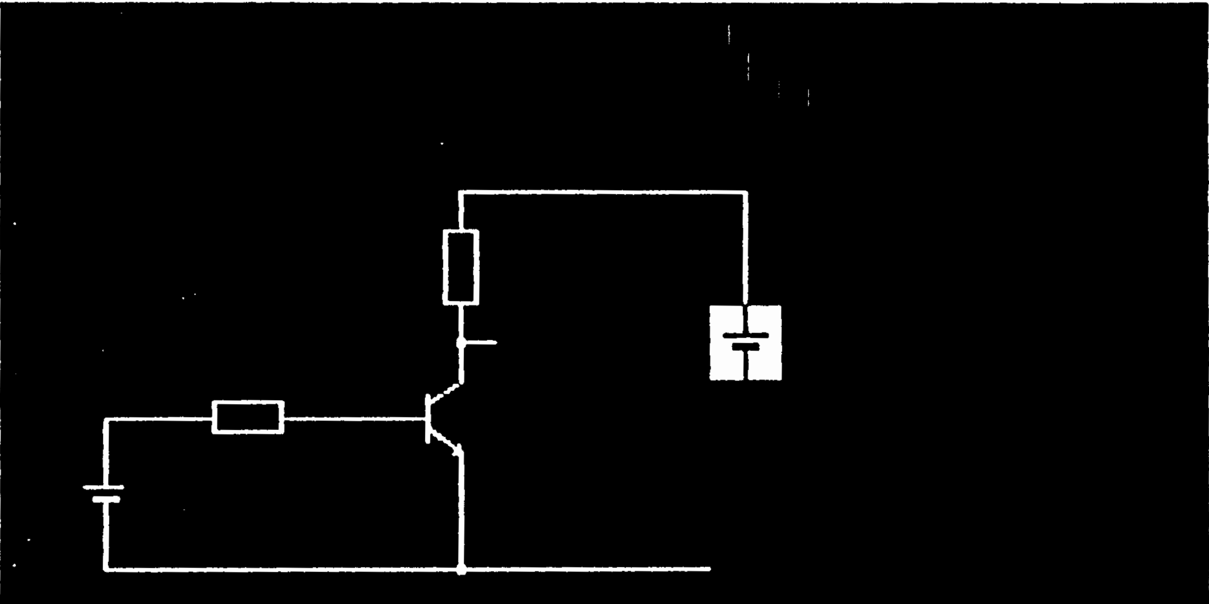
To load a circuit diagram file from disk type *F2*, and then the file name (excluding the extension *.dgm*).

A hard copy of the circuit diagram can be obtained on any Epson FX-80 compatible printer by typing *F6*.

The symbols in the symbol table can be rotated around a horizontal and/or vertical axis (*F8* and *F9* keys), which is analogous to 'flipping' the symbol.







F1	Save diagram	F2	Load diagram
F3		F4	Overwrite on/off
F5	DRAW DIAGRAM TEXT mode	F6	Print circuit diagram
F7	DEFINE SYMBOLS mode	F8	Flip symbol (Hor. axis)
F9	Flip symbol (Vert. axis)	F10	Erase circuit diagram
Home, End	Select circuit symbol		
↑↓→←	Move diagram cursor		CURSOR POSITION
Enter	Transfer a circuit symbol		x : 11
Esc	Quit program		y : 5

DRAW DIAGRAM SYMBOLS

OVERWRITE

Figure 3-2. Circuit Draw display in *DRAW DIAGRAM SYMBOLS* mode.

The contents of the circuit diagram can be erased with the f10 key. Circuit Draw will however ask for confirmation before erasing the diagram.

To quit the program simply press *Esc* and answer *Y* to the confirmation message.

Refer to the next section for a description on how to add text to the circuit diagram.

3.5 Drawing diagram text

To draw text on the circuit diagram, Circuit Draw must be in the *DRAW DIAGRAM TEXT* mode (*F5* key). When this mode is selected the circuit symbol cursor will disappear and be replaced by a diagram text cursor. The diagram text cursor is moved around the screen using the arrow keys. Text is inserted by simply typing the character to be 'drawn' on the diagram. To delete a character type a space over an existing character after setting *OVERWRITE* on (*F4* key).

The other options available in the menu are identical to those in *DRAW DIAGRAM SYMBOLS* mode.

3.6 Creating/editing symbols

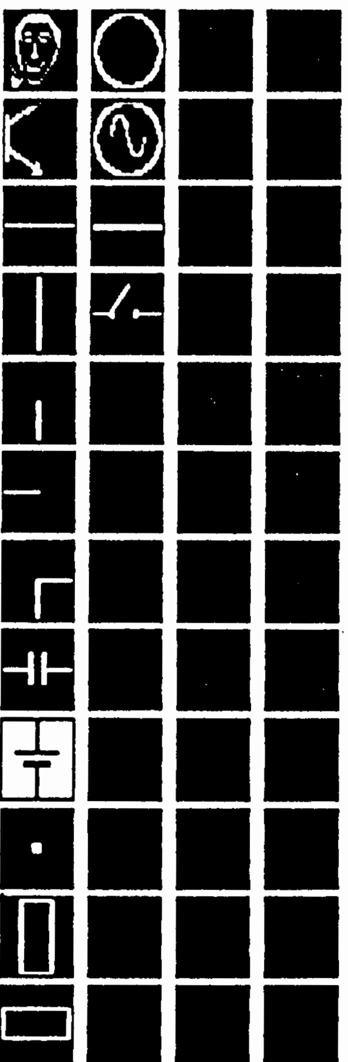
Symbol table management is possible when Circuit Draw is put in the *DEFINE SYMBOLS* mode (figure 3-4). In this mode it is possible to:

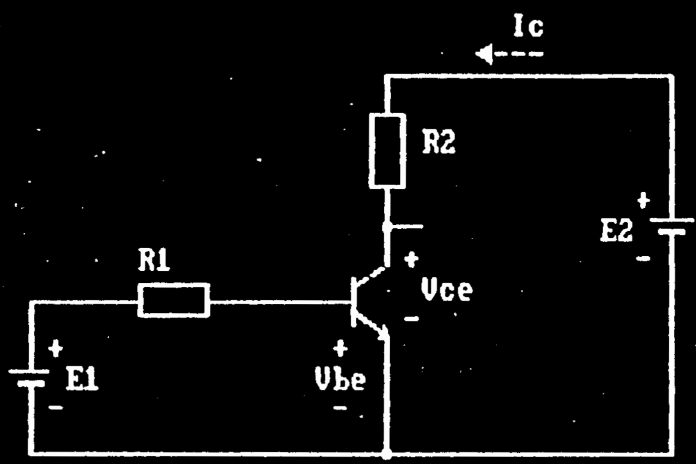
- Save/load a symbol table to/from disk (*F1*, *F2* keys).
- Move a symbol between the symbol table and the grid (*F4*, *F6* keys).
- Modify/create a symbol using the symbol grid.
- Flip symbols around horizontal/vertical axis (*F8*, *F9* keys).

During initialisation Circuit Draw will load the symbol file *DEFAULT.SYM*. It is however possible to load another symbol table by pressing *F2*, and then typing the file name (excluding the extension *.SYM*). After a symbol table has been modified it can be saved to disk (*F1* key).

The grid consists of a 32 by 23 array of blocks, where each block represent a circuit symbol pixel. The flashing grid block is termed the grid cursor. The grid cursor is moved with the arrow keys. To change the colour of the grid block press *Enter*.

A symbol can be loaded from the symbol table by first selecting the symbol (*Home* and *End* keys) and pressing *F4*. After a symbol has been defined on the symbol grid it can be added to any symbol table with spare symbol blocks, by loading the symbol table, selecting a spare symbol table block and then pressing *F6*.





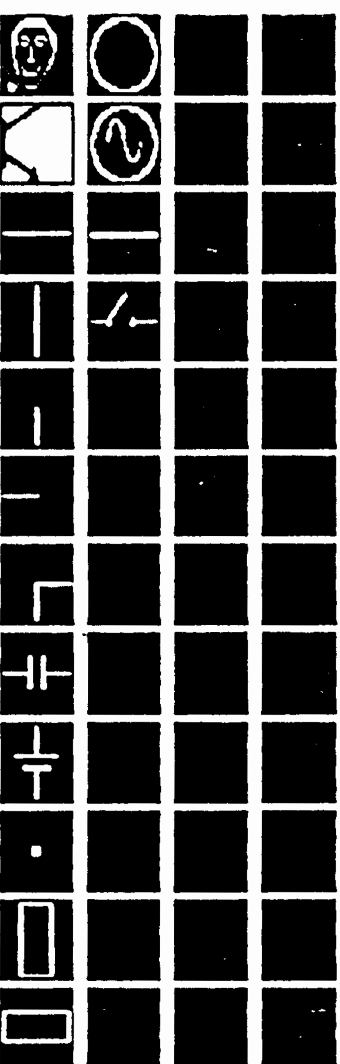
F1	Save diagram	F2	Load diagram
F3	DRAW DIAGRAM SYMBOLS mode	F4	Overwrite on/off
F5		F6	Print circuit diagram
F7	DEFINE SYMBOLS mode	F8	
F9		F10	Erase circuit diagram

↑↓→← Move diagram cursor

Esc Quit program

DRAW DIAGRAM TEXT   OVERWRITE

Figure 3-3. Circuit Draw Display in *DRAW DIAGRAM TEXT* mode



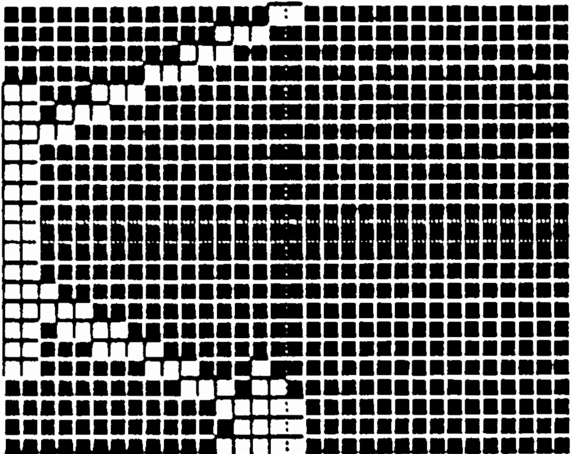


Figure 3-4. Circuit Draw display in *DEFINE SYMBOLS* mode.

Menu options are selected by pressing the appropriate function keys indicated in the menu. Any command may be aborted by pressing *Escape*.

FUNCTION	KEY(S)	MODE(S)	DESCRIPTION
Save diagram	F1	1,2	Circuit Draw prompts the user for a file name which may be appended by a DOS path. An extension must not be specified as Circuit Draw adds extension *.dgm* to the file name.
Load diagram	F2	1,2	A circuit diagram that has been previously saved with above function, can be loaded from disk.
DRAW DIAGRAM SYMBOLS mode	F3	2,3	Puts Circuit Draw in *DRAW DIAGRAM TEXT* mode (MODE 1) to facilitate the drawing of circuit diagram symbols.
Overwrite on/off	F4	1,2	When *OVERWRITE* is off symbols are drawn over the existing symbol on the diagram.
DRAW DIAGRAM TEXT mode	F5	1,3	Puts Circuit Draw into *DRAW DIAGRAM TEXT* mode (MODE 2).
Print circuit diagram	F6	1,2	Print the diagram to a Epson FX-80 compatible printer. Circuit Draw will ask for confirmation. If Circuit Draw is to proceed type *Y* else press *Esc* to abort.
DEFINE SYMBOLS mode	F7	1,2	Put Circuit Tutor in *DEFINE SYMBOLS* mode (MODE 3).
Flip symbol (hor. axis)	F8	1,3	The selected symbol table symbol is rotated 180° around it's horizontal axis.
Flip symbol (vert. axis)	F9	1,3	The selected symbol table symbol is rotated 180° around it's vertical axis.
Erase circuit diagram	F10	1,2	Erase the diagram if user confirms.
Select circuit symbol	Home, End	1,3	A symbol is selected from one of the 48 displayed graphics symbols on the left-hand side of the screen.

FUNCTION	KEY(S)	MODE(S)	DESCRIPTION
Move diagram cursor	arrow keys	1	The diagram cursor (unhighlighted square block on the circuit diagram) is moved using the arrow keys.
Transfer a circuit symbol	Enter	1	The symbol selected in the symbol table is drawn at the circuit diagram cursor position.
Save symbol table	F1	3	Save the symbol table to a *.SYM* file. The default symbol table is loaded from disk file *DEFAULT.SYM*. Circuit Draw asks for a file name. The file name may be appended by a DOS path name, but should not include the *.SYM* file extension.
Load symbol table	F2	3	Load the symbol table from a *.SYM* file previously saved with the above function.
Transfer symbol to grid	F4	3	Transfer the symbol selected for the symbol table to the symbol (font editor) grid.
Transfer symbol from grid	F6	3	Transfer the symbol from the grid to the selected symbol table symbol.
Erase Grid	F10	3	Erase the font editor grid.
Move grid cursor	arrow keys	3	Move the flashing cursor of the font editor grid.
Change pixel colour	Enter	3	Change the colour (black or white) of the pixel selected by the font editor grid cursor.
Quit program	Esc	1,2,3	Quit program and return to DOS.

ADDING A CIRCUIT TO CIRCUIT TUTOR

This chapter is a guide to adding or removing a circuit from Circuit Tutor. To add a circuit the following steps are necessary:

- Instal Circuit Tutor (see Appendix B).
- Draw the circuit diagram using Circuit Draw. For more information refer to chapter 3.
- Create a Turbo Pascal (version 4) unit to solve the circuit.
- Modify Circuit Tutor and recompile (using the Turbo MAKE facility).

4.1 Creating the unit

The unit provided by the user must have the following structure.

```
unit UnitName;

interface

uses MmiGlobals, Misc, UserMsgs, CctDgm, Outputs, Pars;

procedure ProcName(Option: CctSolveOption);

implementation

var <circuit parameter/outputs declarations>

procedure ProcName(Option: CctSolveOption);
begin
  case Option of
    IdentifyCct:
      DefCct(CctName, CctDiagramFile);
    InitCct:
    begin
      <one or more DefParameter statements>
      <one or more DefOutput statements>
      <any other initialisation statements>
    end;
    SolveCct:
    begin
      <assign new value to circuit outputs>
      <assign a value to CctResult>
    end
  end{case}
end{ProcName};

end.
```

Italicised text or text enclosed in < > brackets must be substituted with appropriate text by the programmer.

UnitName is the name of the unit. It is recommended that units be named *Cct1*, *Cct2*, etc. to correspond with the circuit number. The interface section must have a *USES* statement that contains all the units listed in addition to units used by the programmer e.g. mathematical toolboxes.

ProcName must be exported i.e. it must be declared in the implementation part. It is recommended that *ProcName* be named *SolveCct1*, *SolveCct2*, etc..

All circuit parameters and outputs of the circuit must be static variables, this implies that they must be declared at the highest level inside the implementation part of the *UnitName* unit, or alternatively be declared as typed constants in the variable declaration part of *ProcName*. All circuit parameters and outputs must be real numbers.

The case statement has three parts to coincide with the *CctSolveOption* type, which has been declared

```
CctSolveOption = (IdentifyCct, InitCct, SolveCct);
```

4.1.1 IdentifyCct statement

This part of the procedure is invoked during the initialisation of Circuit Tutor. *CctName* is the name of the circuit that will be displayed in the *SELECT CIRCUIT* menu. *CctDiagramFile* is the file name (e.g. *Cct1.dgm*, *Cct2.dgm*, etc.) where the circuit diagram can be found. The circuit diagram is created using Circuit Draw. *IdentifyCct* is declared

```
Procedure IdentifyCct(CctName, FileName: string);
```

4.1.2 InitCct statement

When the user selects the circuit from the *SELECT CIRCUIT* menu, the circuit diagram is loaded from the filename *CctDiagramFile* and displayed on the screen. Thereafter Circuit Tutor executes the *InitCct* part of *ProcName* section. All circuit parameters and outputs to be accessed by the user must be identified. Parameters are identified with the *DefParameter* procedure, whilst outputs are defined with *DefOutput*. Circuit outputs and parameters not accessible by the user need not be defined, but can be initialised here. *DefParameter* is declared in unit Pars as

```
Procedure DefParameter(var UserVar   : real;
                           ParName : string;
                           ValInit,
                           ValMin,
                           ValMax,
                           ValStep   : real;
                           CctX,
                           CctY      : byte);
```

UserVar is the identifier used to store the value of a parameter (e.g. *Beta, Q12hFE*). *ParName* is the string displayed to the user to identify the parameter. Normally this is the string equivalent of *UserVar* (e.g. 'Beta', 'Q12hFE').

ValInit is the default value of the parameter. *ValMin* and *ValMax* are the default minimum and maximum values of the parameter i.e.

ValMin < UserVar < ValMax.

ValStep is the default step value used to decrement/increment the value of a parameter using the + and - keys.

CctX and *CctY* defines the position of the parameter on the circuit diagram. The Circuit Draw cursor position for the parameter symbol must be used. Circuit Tutor uses these co-ordinates to select a parameter in ***Quick*** (or *OnCct*) mode.

DefOutput is declared (in the *Outputs* unit) as

```
Procedure DefOutput (var UserVar: real;
                         Name   : string;
                         ValInit: real);
```

UserVar is the circuit output identifier (e.g. *Vout*) and *Name* is it's string equivalent (e.g. 'Vout'). *ValInit* can normally be set to any value.

4.1.3 SolveCct statement

This part is executed every time a parameter is modified by the user to obtain the new circuit output value. If no solution could be found the value of the *CctResult* global variable should be set to an integer value 1 to 10. This will cause Circuit Tutor to discard the outputs and display message number *CctResult* from the file *TUTOR.MSG*. A list of available messages, and the procedure for adding more messages, is contained in Appendix 1.

The following example of a unit to solve a simple circuit (figure 4-1) demonstrates the above principles.

```
unit Cct2;

interface

uses MmiGlobals, Misc, UserMsgs, CctDgm, Outputs, Pars;

procedure SolveCct2(option: CctSolveOption);

implementation

var E,R,C,t,i,Vc :real;

procedure SolveCct2(option: CctSolveOption);
begin
  case Option of
    IdentifyCct:
      DefCct('2  Capacitor charging circuit','cct2.dgm');
    InitCct:
    begin
      { define circuit parameters }
      DefParameter(E,'E', 1.0, 1.0, 10.0, 1.0, 3,6);
      DefParameter(R,'R', 1.0e+6, 0.1e+6, 10e+6, 0.1e+6, 7,5);
      DefParameter(C,'C', 1e-6, 1e-6, 100e-6, 1e-6, 9,6);
      DefParameter(t,'t', 0.0, 0.0, 100.0, 1.0, 5,5);
      { define circuit outputs }
      DefOutput(t,'t',0.0);
      DefOutput(i,'i',0.0);
      DefOutput(Vc,'Vc',0.0);
    end;
    SolveCct:
      if C<>0 then
      begin
        i := (E/R) * exp(-t/(R*C));
        Vc:= E - i*R;
        CctResult:=0    { indicates that circuit outputs are legal }
      end
      else
        CctResult:=1
  end{case}
end{SolveCct2};

end{Cct2}.
```

CIRCUIT PARAMETERS

E	1.000e+00
R	1.000e+06
C	1.000e-06
t	12.00e+00

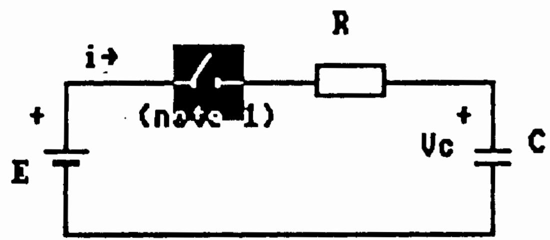
F9/F10 to select
+/- to incr/decr

CIRCUIT OUTPUTS

t	12.00e+00
i	6.144e-12
Vc	1.000e+00

F7/F8 to page
19:45 407kB free

2 Capacitor charging circuit



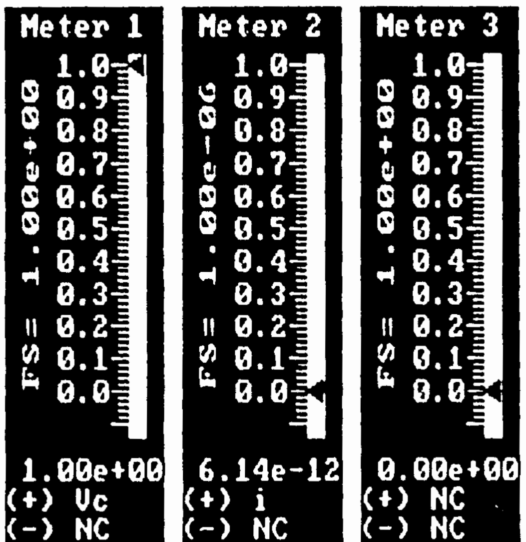
notes.

1. Switch closed for t≥0 and open for t<0.

2. The current i is a funtion of time

   i = i(t)

   $i = i(0)e^{-t/RC}$



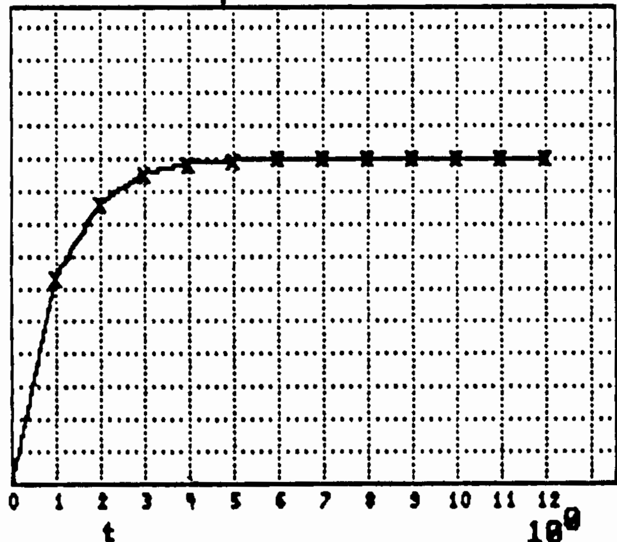


Figure 4-1. Circuit Tutor display (in *QUICK* mode) for circuit no.2

The following modifications are required to Circuit Tutor when adding a circuit.

- Modifications to *Tutor* program (*TUTOR.PAS* file):

  Expand the USES statement to include *UnitName*.

  Expand the CASE statement of procedure *SolveCircuit* to include *ProcName*. With three circuit solve units installed for Circuit Tutor procedure *SolveCircuit* should resemble the following:

```
procedure SolveCircuit(Option: CctSolveOption);
begin
  CctResult:=-1;     { Circuit outputs are assumed to be invalid !! }
  case CctSelected of
    1: SolveCct1 (option) ;
    2: SolveCct2 (option) ;
    3: SolveCct3 (option) ;
  end{case};
  NewCctParsToBeSolved:=False
end{SolveCircuit};
```

- Modifications to *MmiGlobals* unit (*MMIGLOBA.PAS* file):

  Increment the value of the *MaxNoOfCcts* constant to correspond to the number of circuit solver units.

CIRCUIT TUTOR MESSAGES

File *TUTOR.MSG* contains all Circuit Tutor messages. It is possible to edit this file using a text editor that produces ASCII text files. The following table lists all messages used by Circuit Tutor. Messages 1 to 10 can be modified, but should not be deleted. The other messages should be modified with care.

message number	description
1..10	These messages will be displayed when the *CctResult* variable is set to a value in the range [1..10] e.g. if *CctResult = 3*, then message number 3 will be displayed.
11..14	Reserved for future versions of Circuit Tutor.
15..17	Circuit Tutor system error messages.
18	Welcome message.
19	General help message (*F1* key)
20..49	Context-sensitive help messages (*Alt-F1* key).

Messages must have the following format:

```
.mn<2-digit message number>
.he<message heading>
<message text>
.me
```

For example message number 1 may look something like this:

```
.mn01
.heError whilst solving circuit
PROBLEM:
The routine to solve the circuit
outputs can not find a solution.

SOLUTION:
Change the parameter values and
re-attempt. Alternatively reset all
parameters to their default values
using 'reset circuit' on menu no.1
(Circuit menu).
.me
```

As the message window is 40 characters wide, message lines should not exceed this width i.e. the right margin of the editor must be set at column 40.

When Circuit Tutor is initialised the messages are read from file *TUTOR.MSG* into a fixed array variable whose size is determined by the constant *MaxBufLines*.

If the total number of message lines exceeds *MaxBufLines* Circuit Tutor will display an error message during initialisation. When this happens the value of *MaxBufLines*, which is declared in the implementation part of unit *UserMsgs*, must be increased.

CIRCUIT TUTOR INSTALLATION GUIDE

Circuit Tutor has been packaged to facilitate two types of users:

- those only interested in using the program with the circuits already installed; and
- those wanting to add circuits, or modify Circuit Tutor.

To run Circuit Tutor you need

- an IBM PC/XT/AT or compatible micro-computer with at least 384 kByte RAM and one floppy-disk drive;
- a Hercules Graphics Card;
- MS-DOS version 2.0 or later; and
- the Circuit Tutor main diskette.

To add a circuit or to modify Circuit Tutor you need, in addition to the above,

- a hard disk drive;
- Turbo Pascal version 4.0;
- Turbo Graphix Toolbox version 4.0;
- Circuit Tutor library diskette;
- Circuit Tutor source code diskette; and
- the Circuit Draw diskette.

To instal Circuit Tutor, create the following sub-directories for the hard disk:

```
\CctTutor\Tutor
\CctTutor\Library
\CctTutor\Draw
```

Copy all the files from the Circuit Tutor main diskette and Circuit Tutor source code diskette to the \CctTutor\Tutor sub-directory. Copy the files from the Circuit Tutor library diskette to the \CctTutor\Library sub-directory. Finally copy the Circuit Draw diskette files to \CctTutor\Draw.

It is recommended that all Turbo Pascal's files be kept in a directory \TP. This will ensure that the Turbo Pascal configuration files (TURBO.TP) provided with Circuit Tutor and Circuit Draw need not be modified. The Turbo Pascal directory must be included in the parameter list of the PATH command in the AUTOEXEC.BAT file, as follows:

```
PATH = \TP[;<other directories>]
```

To modify Circuit Tutor type the underlined text

C:>cd \CctTutor\Tutor

C:\CCTTUTOR\TUTOR>Turbo

at the DOS prompt.

A list of all the files contained on the four distribution diskettes for Circuit Tutor follows.

Circuit Tutor main diskette

CCT??	DGM	Circuit diagram files for Cct1, Cct2, etc.
TUTOR	EXE	Circuit Tutor program.
14X9	FON	Turbo Graphix Toolbox font file.
4X6	FON	Turbo Graphix Toolbox font file.
8X8	FON	Turbo Graphix Toolbox font file.
8X8UP	FON	Font for vertically orientated text.
ERROR	MSG	Turbo Graphix Toolbox error messages.
TUTOR	MSG	Circuit Tutor message file.

Circuit Tutor source code diskette

CCT??	PAS	Pascal source code for Cct1, Cct2, etc.
CCTDGM	PAS	Pascal source code for CctDgm unit.
CHR8X8	PAS	Pascal source code for Chr8x8 unit.
MENUS	PAS	Pascal source code for Menus unit.
GRAPH	PAS	Pascal source code for Graph unit.
METERS	PAS	Pascal source code for Meters unit.
MISC	PAS	Pascal source code for Misc unit.
MMIGLOBA	PAS	Pascal source code for MMIGlobals unit.
OUTPUTS	PAS	Pascal source code for Outputs unit.
PARS	PAS	Pascal source code for Pars unit.
TUTOR	PAS	Pascal source code for Circuit Tutor main unit (program).
USERMSGS	PAS	Pascal source code for UserMsgs unit.
TURBO	TP	Turbo Pascal configuration file for Circuit Tutor.

Circuit Tutor library diskette

HGC	PAS	Pascal source code for Turbo Graphix Toolbox.
GDRIVER	INC	Pascal source code for Turbo Graphix Toolbox.
KERNEL	INC	Pascal source code for Turbo Graphix Toolbox.
GRAFHGC	ASM	Assembler source code for Turbo Graphix Toolbox.
GRAFHGC	OBJ	Turbo Graphix Toolbox object code file.
MATHS	PAS	Real maths routines.
CMATHS	PAS	Complex maths routines.
ASCII	PAS	Ascii constant identifiers.

Circuit Draw diskette

DRAW	EXE	Circuit Draw program.
*	FON	All the font files as for the Circuit Tutor Main disk.
ERROR	MSG	Turbo Graphix Toolbox error messages.
DEFAULT	SYM	Default symbol table.
DRAW	PAS	Pascal source code for Circuit Draw main unit (program).
DRAWMISC	PAS	Pascal source code for DrawMisc unit.
DRAWTEXT	PAS	Pascal source code for DrawText unit.
SYMTABLE	PAS	Pascal source code for SymTable unit.
DRAWGLOB	PAS	Pascal source code for DrawGlobals unit.
DIAGRAM	PAS	Pascal source code for Diagram unit.
GRID	PAS	Pascal source code for Grid unit.
HGCPLUS	PAS	Pascal source code for HgcPlus unit.
TURBO	TP	Turbo Pascal configuration file for Circuit Draw.

Part 3

Circuit Tutor
Designer's Reference Manual

Version 1.04/hgc
October 1988

1 SPECIFICATION FOR CIRCUIT TUTOR 3-1

1.1 Specification methodology 3-1
1.2 General goals 3-1
1.3 Functional requirements 3-3
1.4 Non-functional requirements for Circuit Tutor 3-4

2 DESIGN OF CIRCUIT TUTOR 3-6

2.1 Design of the VDU lay-out of Circuit Tutor 3-6
2.2 Selection of a VDU format 3-8
2.3 Detailed VDU lay-out 3-11
2.4 Overview of current software design methodologies 3-11
2.5 Criteria for choosing a programming language 3-11
2.6 Choosing a computer language 3-14
2.7 Choosing a design methodology 3-17
2.8 Identifying the main objects 3-18
2.9 Preparing a modular design chart 3-22

3 CIRCUIT TUTOR UNITS IN DETAIL 3-24

3.1 The main (program) unit 3-25
3.1.1 Description of routines 3-26
3.2 The Graph unit 3-29
3.2.1 Description of routines 3-30
3.3 The Meters unit 3-33
3.3.1 Description of routines 3-34
3.4 The Pars unit 3-36
3.4.1 Description of internal data 3-37
3.4.2 Description of routines 3-38
3.5 The CctDgm unit 3-45
3.5.1 Co-ordinate systems used by CctDgm unit 3-46
3.5.2 Description of routines 3-48
3.6 The UserMsgs unit 3-51
3.6.1 Description of internal data 3-52
3.6.2 Description of routines 3-53
3.7 The Chr8x8 unit 3-57
3.7.1 Exported data 3-57
3.7.2 Description of routines 3-58
3.8 The Misc unit 3-60
3.8.1 Description of routines 3-60
3.9 The Outputs unit 3-64
3.6.1 Description of internal data 3-65
3.6.2 Description of routines 3-65
3.10 The Menus unit 3-69
3.10.1 Exported data 3-71
3.10.2 Description of routines 3-76

3.11 The MmiGloblals unit 3-83
3.11.1 Miscellaneous constants 3-83
3.11.2 Graphics window constants 3-83
3.11.3 Graphics co-ordinate system constants 3-84
3.11.4 Screen lay-out constants 3-84
3.11.5 Variables 3-84
3.11.6 Type identifiers 3-85
3.11.7 Exported function 3-86
3.12 The Ascii unit 3-87
3.13 The Maths and CMaths units 3-88
3.13 Description of exported data and routines 3-88
3.14 Other units 3-90

4 DESIGN OF THE CIRCUIT DRAW UTILITY PROGRAM 3-91

4.1 General goals 3-91
4.2 Specifications for Circuit Draw 3-91
4.3 Design of the VDU lay-out 3-92
4.4 Design of the program 3-92
4.4.1 Preparing a modular design chart 3-98

5 THE CIRCUIT DRAW UNITS IN DETAIL 3-99

5.1 The main (program) unit 3-99
5.1.1 Description of routines 3-100
5.2 The Grid unit 3-103
5.2.1 Description of routines 3-103
5.3 SymTable unit 3-107
5.3.1 Description of routines 3-108
5.4 Diagram unit 3-109
5.4.1 Description of exported variable 3-110
5.4.2 Description of routines 3-110
5.5 DrawText unit 3-113
5.5.1 Description of exported constant 3-114
5.5.2 Description of routines 3-114
5.6 DrawMisc unit 3-116
5.6.1 Description of routines 3-117
5.7 HgcPlus unit 3-118
5.8 DrawGlobals unit 3-118
5.8.1 Miscellaneous constants 3-119
5.8.2 Graphics window constants 3-119
5.8.3 Types and variables 3-120

BIBLIOGRAPHY 3-121

1

SPECIFICATION FOR CIRCUIT TUTOR

1.1 Specification methodology

Normally the specifications for a program are prepared from the formal requirements provided by the client. At the same time a preliminary user's manual is prepared and presented to the client. The design of the program is then based on these two documents. In the case of Circuit Tutor and Circuit Draw the specifications for these programs should therefore be associated with the User's Reference Manual.

1.2 General goals

The main function of Circuit Tutor is to provide a man-machine interface (MMI) between

1. a library of circuits whose parameter-output relationship is defined using a high level programming language; and

2. the user who wants to test the circuit by modifying circuit parameters and monitor the corresponding change in the circuit outputs.

To be an effective interface means that Circuit Tutor must be interactive and user-friendly. It should be extremely easy to operate, but not cumbersome to students familiar with it.

On-line help must be available at all times. The on-line help must be extensive which implies that a separate user's manual should under normal operating conditions not be necessary.

It should be simple to add or delete a circuit. In fact, this is one of the most important aspects if Circuit Tutor is going to be of any use. A circuit should be added in a matter of minutes, not days! The programmer wanting to add a circuit to Circuit Tutor should have a library of common mathematical functions (matrix operations, Gauss-Jordan elimination, Complex algebra, etc.) at his disposal.

Each circuit to be added should have a circuit diagram associated with it. To aid programmers to draw the circuit diagram a utility program has to be provided. This program has been called CIRCUIT DRAW.

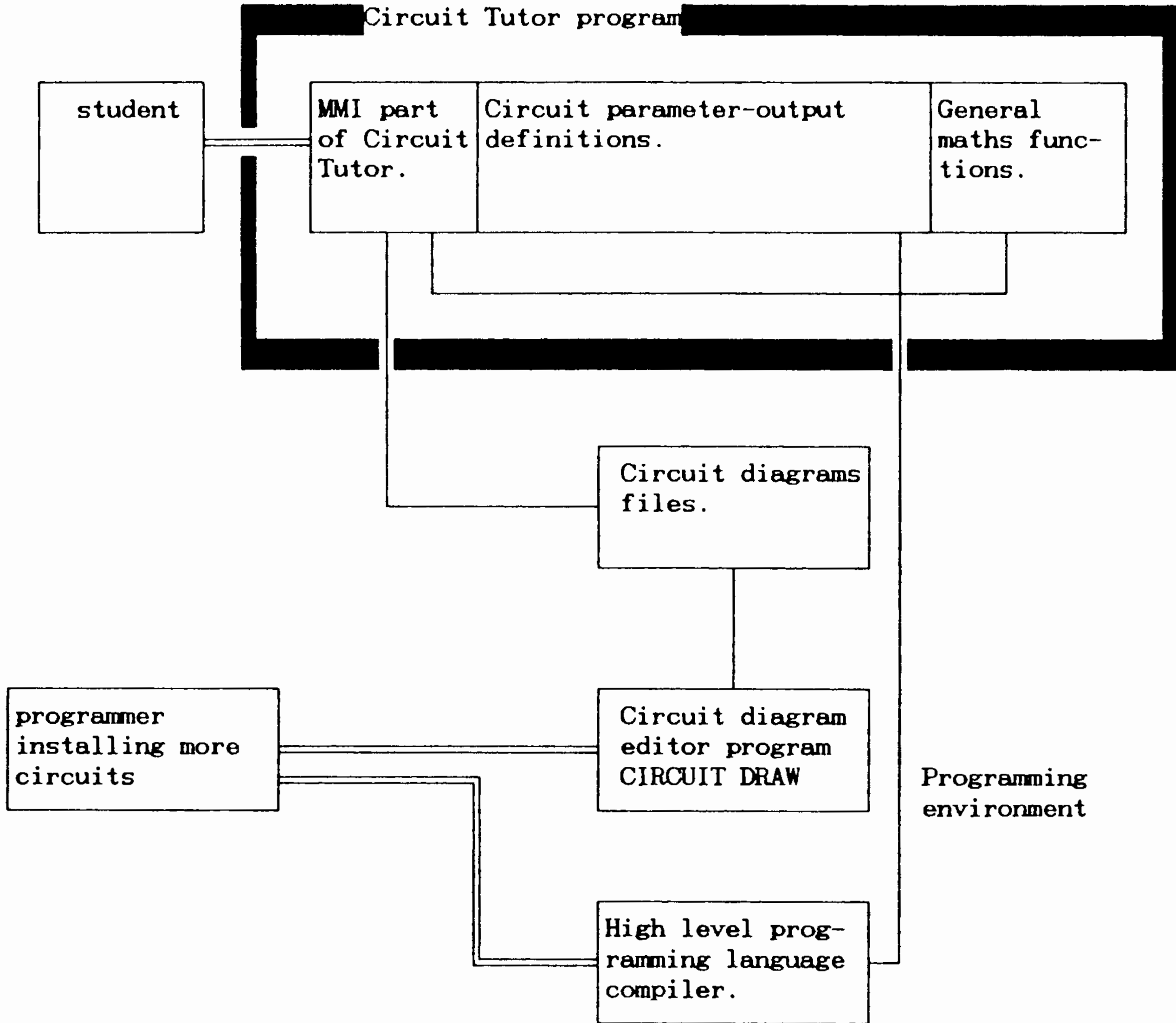


figure 1-1. Detailed functional lay-out of Circuit Tutor system.

1.3 Functional requirements

As stated in chapter 1 the requirements already mentioned the following requirements should be read in conjunction with the user's reference manual for completeness.

1. Circuit Tutor must be menu-driven with clear, concise menus prompting the user for a response. To facilitate experienced users, a quick select facility based on easily remembered keys, must be implemented e.g. to select the main menu the user presses the Alt-M key.

2. Circuits already installed must be selected from a list. After a circuit is selected a circuit diagram must be displayed in addition to all parameter identifiers and their default values.

3. Each circuit parameter to be modified by the user must have the following values associated with it (a circuit parameter is defined as any parameter of the circuit that can be modified by the user e.g. component values, voltage and current supply values, even the time if it is defined as a circuit parameter):

   - Minimum and maximum values. These values define the range over which the parameter can be modified.
   - Step (or incremental) value. The user should be able to make incremental changes to circuit parameters by pressing the + - keys. The step value specifies the increment to be used.
   - Default value. This value is defined by the programmer who adds a circuit to Circuit Tutor. The user should however, have the option at run-time to restore this value.
   - Present value.

   All the above values should be modifiable at run-time. The user should have fast access to these parameters. Circuit parameters and their present values must be displayed in a window hereafter referred to as the circuit parameters window. The present value must be indicated in engineering floating point format e.g. -1.236E-06.

4. Circuit parameters must be selected in the circuit parameters window and the present value incremented or decremented with the step value by pressing the + - keys. The present value and also other values (the minimum/maximum value etc.) must be modifiable from a circuit parameters menu. Both the circuit parameters window and circuit parameters menu must be scrollable.

5. When a circuit parameter is modified by the user a new set of circuit outputs must be calculated and displayed in a second window, the so-called circuit outputs or outputs window. The user must be able to disable this facility and solve the circuits outputs when required. The circuit outputs window must be scrollable.

6. Circuit outputs, identifiers and their values which must also be in engineering format. When circuit outputs are not valid, e.g. when no solution is found, no values must be displayed in the outputs window.

7. Three simulated analogue/digital meters must be provided for monitoring the circuit outputs. Each meter should be able to accommodate two circuit outputs. The value display on the meter must be the difference of the two circuit output values. The meter scale should be numbered -0.1 to 1.0, where 1.0 is the full-scale deflection. The user should be able to associate a full-scale value with the full-scale deflection. The user must be able to modify the circuit output assignment and full-scal value during run-time. In addition these values must be indicated on the meter. The meter should give the user a feeling that he is using a real meter i.e. the needle should simulate the the mechanical damping property of real meters.

8. A simulated graph (xy-plotter) must be provided. The user must be able to configure this graph via the menu. It must be possible to associate two circuit outputs with each axis of the graph. The axis must be set-up by specifying: (a) minimum value; (b) increment per division; and (c) an axis multiplier. In addition the following functions must be provided: (a) erase graph; (b) set connecting of plotted points on/off; and (c) set marking of plotted points on/off.

9. The time and available memory must be displayed. When circuit outputs are displayed a 'Solving outputs' message must be displayed.

1.4 Non-functional requirements for Circuit Tutor

---

1. Circuit Tutor must be implemented on microcomputers in use at the Faculty of Electrical Engineering, UCT. This implies that it should be able to run on a personal computer compatible with the IBM PC/XT/AT.

2. Help messages must be stored in an ASCII file to facilitate easy modification/addition. A general purpose help facility must be available at all times by pressing <F1>. A context-sensitive help facility, which is selected by pressing <Alt F1>, must be available for all menu items.

3. The system must be implemented in a computer language that can easily be maintained by University students and lecturers. Specialised/experimental languages must be avoided.

4. Circuit parameter and output identifiers must be at least 6 characters long. Parameter and output values must be displayed in engineering floating point format with at least four significant digits e.g. -1.234E-12.

5. When a fatal error occurs (e.g. insufficient memory for initialisation) a message must be displayed indicating the reason before termination.

6. When a non-fatal error occurs (e.g. unable to solve circuit outputs) an error message must be displayed to the user. This message must describe the problem and suggest possible solutions.

7. The number of circuit parameters and outputs per circuit should only be limited by the available memory of the personal computer.

DESIGN OF CIRCUIT TUTOR

2.1 Design of the VDU lay-out of Circuit Tutor

The main function of Circuit Tutor is to provide a man-machine interface (MMI) therefore the design of the VDU lay-out deserves special attention. In the design of a MMI the following points must be borne in mind [Huckle, 1981]:

- Information must be properly spaced.
- Make use of highlighting features to draw attention to events: underlining, reverse video, etc. This has the effect of making the interface easier to use and also more exciting.
- Highlight the non-variable part of an input field, and display the non-variable part normally.
- Use symbols to help identify objects, e.g. associate circuit parameters with circuit diagram symbols.
- Do not display information that has no meaning to the user.
- Menu items must be carefully chosen to avoid ambiguity.
- Menu items must not be selected by a numbering scheme.

As a first step the screen lay-out will now be presented (figure 2-1). The VDU area is divided into eight windows, namely:

- The circuit parameters window. This window displays ten circuit parameter identifiers and their present values.
- The circuit outputs window. This window displays nine circuit output identifiers and their calculated values (if valid).
- The miscellaneous window displays the time, available memory or the 'Solving outputs' message.
- Three meter windows. The simulated analogue meters are displayed in these windows. The meter "needles" deflect upwards.
- Graph window. The X-Y graph occupies this screen area. Part of this window is utilised for the axis labels.
- Diagram/menu window. This window contains both the circuit diagram and the menu. The menu will be displayed over the circuit diagram.

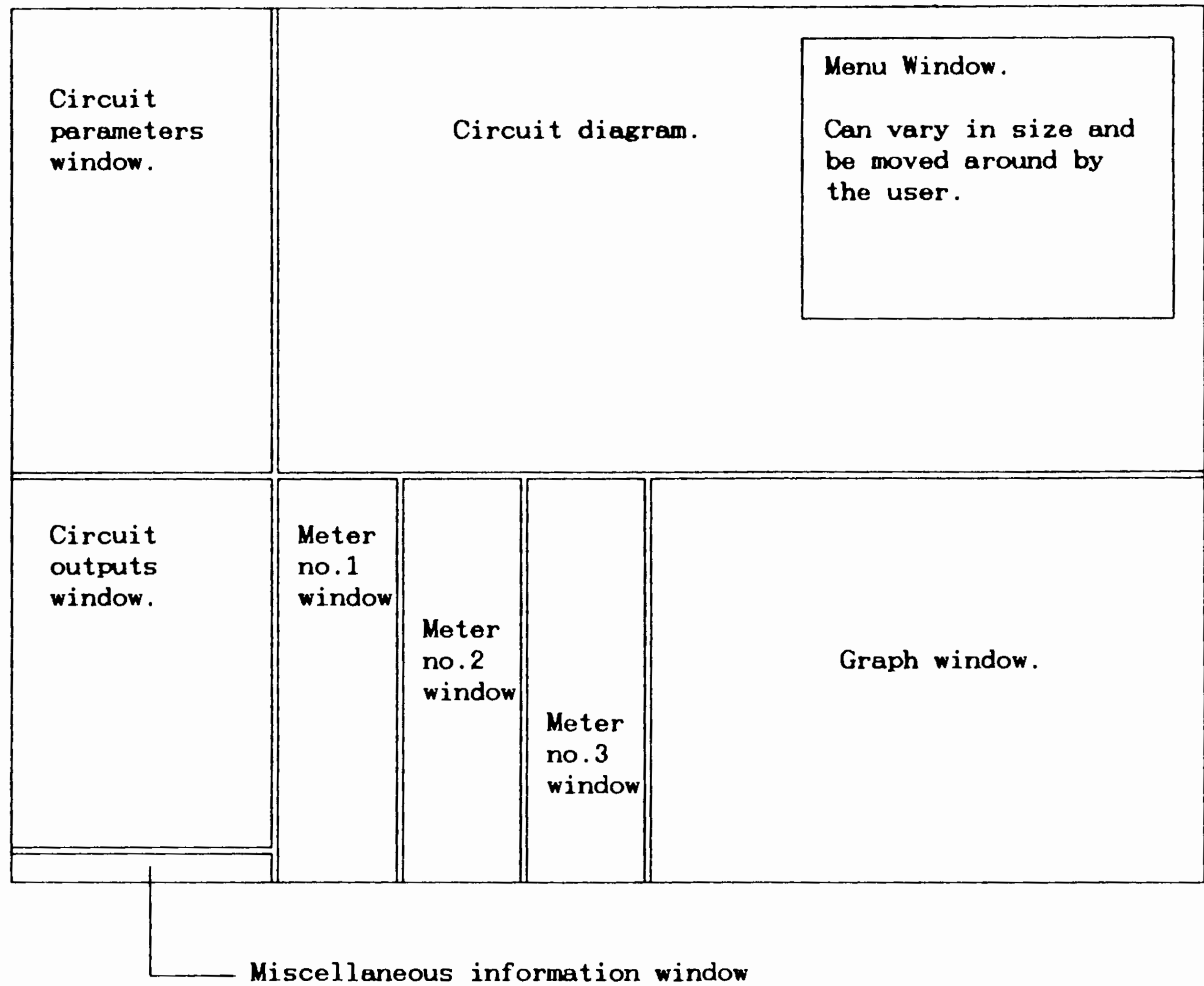


Figure 2-1. Basic lay-out of the VDU for Circuit Tutor.

As the circuit diagram window is the only window not updated on a continuous basis, i.e. every time circuit outputs are solved, the menu must be displayed over this window. It would off course, be better to display the menu in its own window but unfortunately the size of a normal VDU screen does not permit this.

As the user may want to view the entire circuit diagram a facility for clearing the menu temporarily or sliding it over the circuit diagram must be provided.

When the user modifies a circuit parameter the circuit parameter window will be updated with the new circuit parameter value. The circuit outputs will then be calculated and the circuit outputs window updated. If the outputs are monitored by means of the meters and/or graph these windows are also updated.

2.2 Selection of a VDU format

Several different graphics adaptors are available for the IBM PC, including the following:

- IBM CGA
- IBM EGA
- IBM MCGA
- AT&T 400 (or Olivetti M24)
- IBM VGA
- Hercules

It would be ideal to design Circuit Tutor to run on all of the above Graphics adaptors. To design a program that will adapt automatically to all of the above graphics adaptors is however, not a trivial task. As this approach will significantly complicate the design, implementation and especially the testing phase, an alternate approach will be followed. It is considered more practical to implement different version of Circuit Tutor for the above adaptors. By implementing four versions of Circuit Tutor all of the above adaptors can be used.

- 720 (vertical) by 348 (horizontal) pixel version for the Hercules card;
- 640 by 350 pixel version for the EGA and VGA adaptors;
- 640 by 200 pixel version for the CGA and MCGA adaptors;
- 640 by 400 pixel version for the Olivetti adaptor.

The Hercules Monochrome Graphics card is in widespread use in South Africa as this adaptor provides good resolution at a very reasonable price. The EGA and VGA adaptors are very pricy and not in widespread use. Most IBM PC compatible computers are nowadays sold with a Hercules card fitted as standard equipment. The first version of Circuit Tutor, the product of this thesis, will therefore be implemented for the Hercules card.

The version number of Circuit Tutor will be appended with the abbreviation of the graphics adaptor e.g. version 1.04/hgc will only run on a PC with a Hercules card.

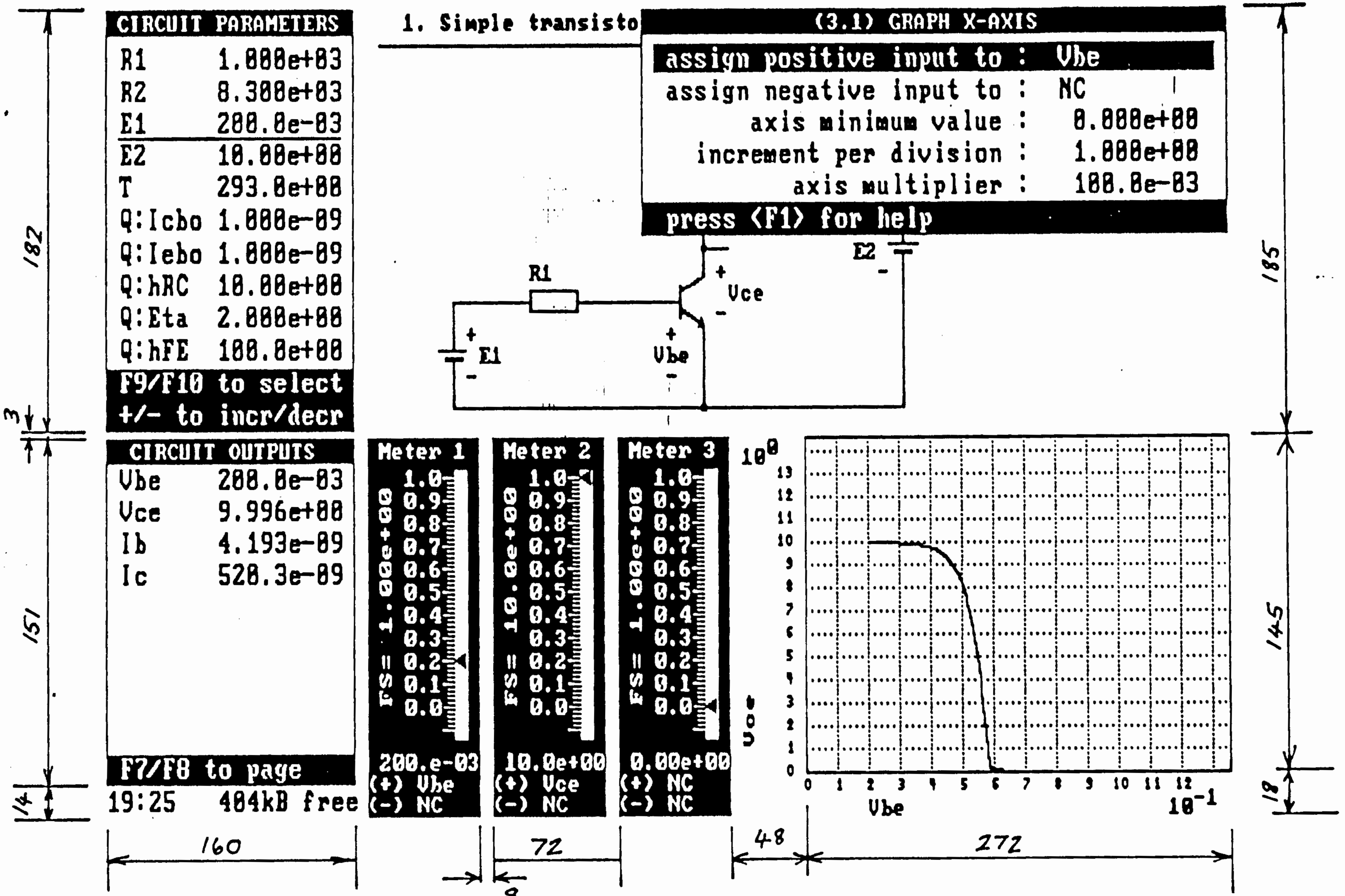


Figure 2-2. Detailed lay-out of Circuit Tutor display.

CIRCUIT PARAMETERS

R1	1.000e+03
R2	8.300e+03
E1	200.0e-03
E2	10.00e+00
T	293.0e+00
Q:Icbo	1.000e-09
Q:Iebo	1.000e-09
Q:hRC	10.00e+00
Q:Eta	2.000e+00
Q:hFE	100.0e+00

F9/F10 to select
+/- to incr/decr

CIRCUIT OUTPUTS

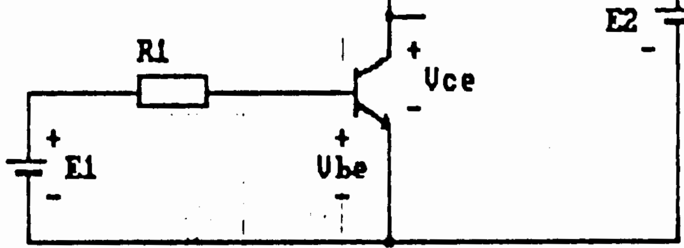
Vbe	200.0e-03
Vce	9.996e+00
Ib	4.193e-09
Ic	520.3e-09

F7/F8 to page
19:25 404kB free

1. Simple transisto

(3.1) GRAPH X-AXIS

assign positive input to : Vbe
assign negative input to : NC
axis minimum value : 0.000e+00
increment per division : 1.000e+00
axis multiplier : 100.0e-03
press <F1> for help



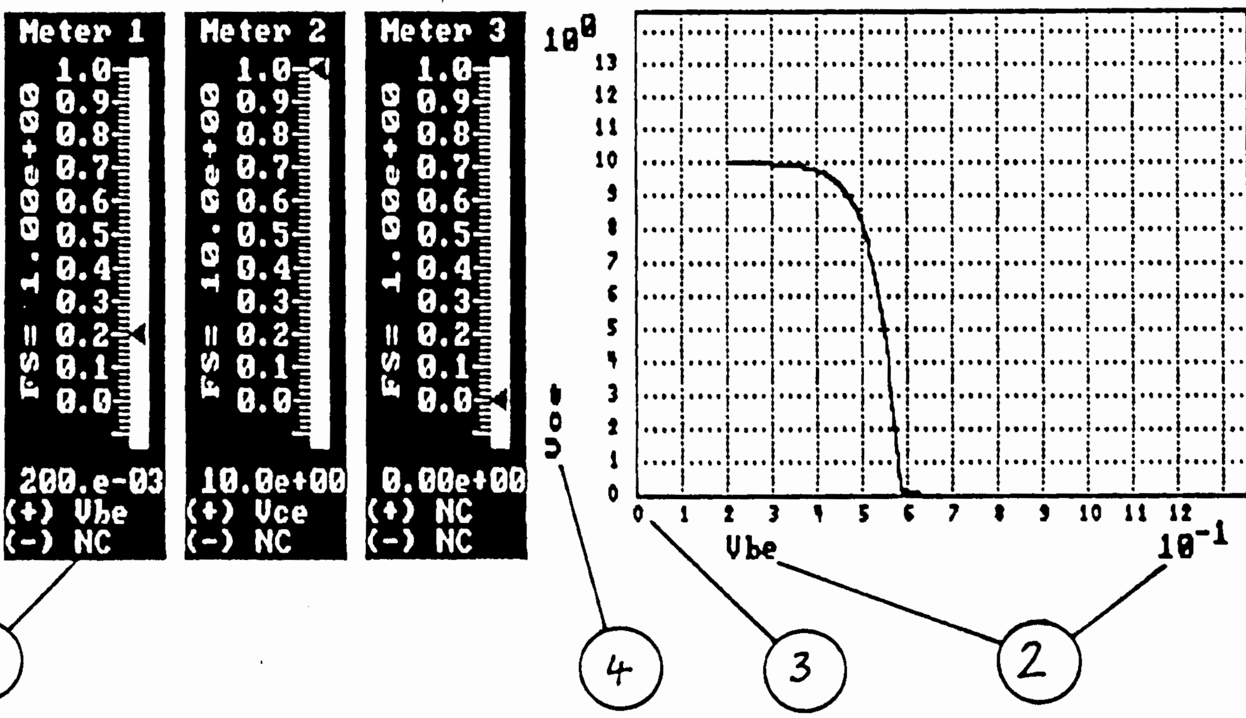


Figure 2-3. Various text fonts used for the Circuit Tutor display.

2.3 Detailed VDU lay-out (for Hercules card)

The VDU lay-out for the Hercules graphics adaptor is given in figure 2-2. Dimensions of the windows are indicated in pixels. Four text fonts have been used:

1. Normal Hercules font 9x14 (horizontal, vertical) pixel matrix.
2. 8x8 pixel matrix font.
3. 4x6 pixel matrix font.
4. 8x8 pixel matrix font, but characters orientated upwards.

Figure 2-3 indicates where these character fonts are used.

2.4 Overview of current software design methodologies

The first step of the design process is to determine the design method to be used. Most existing software design methodologies entail decomposing the program into manageable parts. For example a problem can first be decomposed into three functions F1, F2 and F3 (see figure 2-4). Next F2 can itself be decomposed to comprise of functions F2.1 and F2.2. The design is complete when none of the functions can be further decomposed.

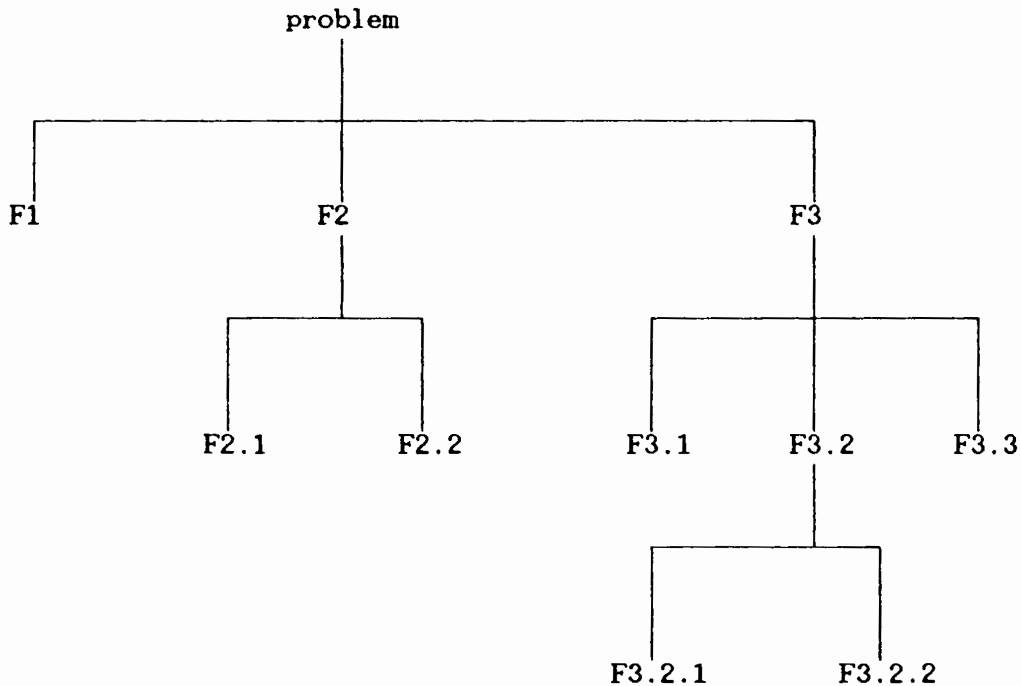


Figure 2-4. Top-down refinement of problem.

By breaking up the program into separate parts the software development and maintenance effort is simplified; for example it is much easier to develop and maintain for example ten 500 line modules than it is to maintain one 5 000 line monolithic program.

In practice, the decomposition of a problem is often a fairly complex task. There are a number of techniques available to aid the designer. As a detailed discussion of all these techniques are beyond the scope of this dissertation the interested reader can refer to chapter 4 of Wiener (1984), which gives a good review of these methods. Two methods include:

- Top-down-bottom-up design. Most problems can normally be decomposed in a top-down manner. Sometimes however, this is accompanied with low-level design; for example the available graphics support will normally dictate the functions to be used at the bottom level of the design. Pseudo-code can be used to express some or all of the functions. It is often easier to decompose a function after writing the pseudo-code for a function.

- Object-orientated design. This is a fairly modern approach which is gaining increasing popularity with the advent of modern programming languages such as Ada and Modula-2. This design approach is described in detail by R. Wiener. In short it entails mapping an entity from the problem domain to the module of the programming language.

  The basic principles of object orientated design can be demonstrated by considering the simple problem of implementing a last-in-first-out (LIFO) stack. Here the object is clearly the LIFO stack and can be implemented by a module called STACKS. The STACKS module will export the data types and functions necessary for other modules to define and manipulate their own stacks. For example the STACKS module can export an abstract type STACK, and the functions necessary for initialising and popping/pushing elements on/off STACK type variables. One of the most important aspects of object-orientated design is data hiding. Data hiding is the ability of modules to export a type without specifying its internal representation details. If data abstraction is applied for the STACKS module other modules would not be able to directly modify the internal structure of STACK type variables.

  True data abstraction is only possible if the programming language supports private (or opaque) types. If the STACKS module is implemented using the Modula-2 programming language [Wirth (1983), Gleaves (1984)], which supports data hiding, the definition module may resemble the following:

```
DEFINITION MODULE stacks;
EXPORT QUALIFIED stack, empty, pop, push, initialise, remove;
TYPE stack;          (* Note that the representational details are hidden! *)
PROCEDURE empty(s:stack):BOOLEAN;       (* True if stack 's' empty.        *)
PROCEDURE pop  (VAR s:stack):CHAR;      (* Pop character off stack 's'.    *)
PROCEDURE push (VAR s:stack;ch:CHAR);(* Push character onto stack 's'.  *)
PROCEDURE initialise (VAR s:stack);     (* Initialise stack variable 's'.  *)
PROCEDURE remove (VAR s:stack);         (* Release memory occupied by 's'. *)
```

The module that implements the stack functions will contain the full declaration of the STACK type, for example the Modula-2 implementation module will contain the following type declaration:

```
TYPE stack = POINTER TO RECORD
  item: ARRAY [1..100] OF CHAR;          (* 100 byte stack.      *)
  top : CARDINAL;                        (* Index of top of stack. *)
END;
```

If the internal representation details of the STACK type were not hidden other modules would be able to manipulate STACK type variables directly and could corrupt the stack. But the biggest disadvantage of modifying variables directly is apparent when the internal representation of the exported abstract type is modified. Now all modules directly modifying variables of the abstract type would have to be identified, modified and finally recompiled. Modules not directly modifying the abstract type need not be modified. For example if the STACK type is modified to comprise of 200 characters, then none of the modules would have to be modified since they could not modify an opaque type variable.

Of course to enforce data hiding the programming language implementation must support data hiding (by means of private/opaque types). If the language implementation does not support data hiding then the onus is on the programmer not to modify the internal representational structure directly.

As the design methodology is dependant on the choice of the programming language (and its implementation) this issue will now be considered.

2.5 Criteria for choosing a programming language

A computer programming language can be evaluated in terms of the following criteria [Smith,1987]:

- Simplicity. A language should be simple to learn to ensure its wide usage. For example Pascal, which was originally designed by Wirth to teach programming, has established itself as an important language at colleges and universities as a result of its simplicity.

- Readability. It is important that a program be as readable as possible since poor readability will significantly increase the software maintenance effort for a program. Assembler languages, whilst efficient, are the least readable. Certain high level languages, for example BASIC and Fortran, whilst being more readable than assembler, still do not provide the same level of readability as that of most block structured languages, for example Pascal and Modula-2.

- Security. This relates to the ability of the language to minimise errors during programming or at run-time. During programming, strong typing, separate modules, and data hiding contribute to the security of a program. During run-time it should be easy to handle errors e.g. out-of-range input values. Ada's exception handling facility for example, provides an efficient means of handling run-time errors.

- Efficiency. Although efficiency is sometimes regarded as a characteristic of the compiler, certain languages were designed with efficiency in mind. One such language, C, is often used as an alternative to assembler.

- Portability. Older programming languages did not allow system dependant functions to be added to the language. Each language implementor therefore added these functions for their particular system. For example, there are probably hundreds of different implementations for the BASIC language. Most modern languages however allow implementors to include system dependant functions, without having to modify the language definition. Modula-2 for example does not have any standard READ or WRITE functions. These functions are considered to be system dependant and must therefore be provided by the implementor. Normally these functions are contained in a separate module, for example Modula-2 normally contains these functions in the SYSTEM module.

In addition to the above-mentioned criteria the following requirements should also be considered when selecting a programming language:

- An efficient compiler must be available: it should be fast; produce efficient code; include features for developing and debugging large modular programs; and have a good user interface.

- Software support should be available for the particular hardware to be used. For example graphics support should be available for the VDU format that has been selected.

No single computer language implementation for the IBM PC provides all of the above features and functions. It is therefore necessary to select a language implementation that meets with most of the stipulated criteria.

2.6 Choosing a computer language

Several language implementations are available for the IBM PC. These languages will now be evaluated:

- Fortran 77. This is a semi-structured language refined from the original language that was developed some 30 years ago. It is still used fairly extensively for engineering and scientific applications. However Fortran 77 does not encourage readability and is not very secure.

- BASIC. This language was developed from Fortran with simplicity as the main criteria. Apart from its simplicity this language has little else to offer: it's neither readable, efficient, portable or secure. In addition a limited number of variable types are available with no facilities for creating additional types. Although BASIC can be considered for small programs, its use should be avoided for larger programs.

- C. The C language contains numerous low-level facilities and allows the programmer significant freedom. This freedom is however gained at the cost of security. The language definition is quite simple thereby making it easy to implement efficient compilers. As the language definition omits system specific functions, implementors do not have to develop their own dialects of the language. Many excellent C compilers are available for the IBM PC. Although C is more readable than the unstructured languages, for example BASIC and Fortran, it is not as readable as Pascal, Modula-2 or Ada.

- Pascal. Pascal was originally developed as a language to teach structured programming, and therefore purposely omitted many important features required in a practical computer language. For example implementors have have had to include functions for modular program development, string handling, and low-level functions. Borland's implementation of Pascal, namely Turbo Pascal, meets virtually all the stipulated criteria for a good computer language. In addition extensive graphics support is available for Turbo Pascal. The Turbo Pascal compiler version 4 is excellent in virtually all respects, except that no source code debugger is included. Turbo Pascal supports modular programming: data structures and associated routines may be grouped together in 'units' that may be compiled separately. Units have two parts: the interface part contains constants, types, variables, procedures and functions automatically exported to all units 'using' the particular unit; whilst the implementation part contains the actual source code for implementing the routine.

  Turbo Pascal does not however, include the features necessary for the implementation of a true object-orientated design: data hiding is not supported; and units must be arranged in a hierarchical structure. For example, if unit A uses unit B then unit B may not use unit A. This is demonstrated in the following figure:

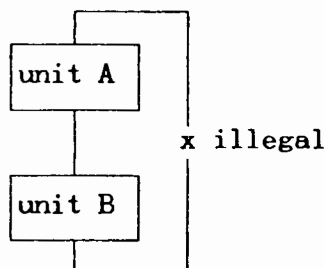


Figure 2-5. Pascal units may not be mutually dependant.

Turbo Pascal does not allow functions to be passed as parameters, an important feature for the writing of general purpose mathematical functions. For example a procedure to solve f(x)=0 can be implemented in Modula-2 by a procedure with the declaration

```
PROCEDURE Zero(f: fx);
```

where 'fx' has been declared as

```
TYPE fx = PROCEDURE (REAL): REAL;
```

Pascal is not very suitable for developing routines that manipulate complex numbers; programmers often forfeit the advantages of a structured language by resorting to Fortran. With the advent of Ada, which supports operator overloading, this will fortunately no longer be necessary. Operator overloading allows one to define (or load) any operator, for example the '*' operator can be defined to indicate complex multiplication.

In conclusion it can be stated that Turbo Pascal is an excellent implementation of an adequate language. Turbo Pascal's power, simplicity and low cost has made it the preferred language of many engineering faculties.

- Modula-2. The Modula-2 language was designed to rectify many, but not all, of Pascal's shortcomings. The Modula-2 language conforms to all the requirements of a computer language, with the exception that it does not provide operator overloading. As previously mentioned Modula-2 supports object-orientated design. Data hiding is supported and implementation modules may import each other. Modula-2 is still however a fairly new language and has yet to establish itself. One of the main reason for its relatively slow uptake (even at universities) has been the absence of a good compiler for the IBM PC. I was unable to obtain extensive graphics support for the Hercules graphics card.

- Ada. This language was designed mainly for the development of embedded systems. It is a fairly large and complex language which offers many sophisticated features such as concurrency, operator overloading, generic functions and exception handling. With the exclusion of simplicity, it conforms to all of the stipulated criteria for a computer language. As Ada compilers require significant computing resources it is unlikely that compilers will ever be available for a standard IBM PC.

In terms of the stipulated criteria Modula-2 appears to be the 'best' language, with Turbo Pascal and C second and third choice respectively. As I could not obtain extensive graphics support or an efficient compiler for Modula-2, both of which are available for Turbo Pascal, it was decided to use Turbo Pascal.

I managed to obtain a copy of Logitech's implementation of Modula-2 (a fairly popular implementation), which I compared with Turbo Pascal version 4. Table 2-1 compare important features of these two implementations. It should be noted that the comments in table 2-1 reflect my personal opinion.

Table 2-1. Comparison of Logitech Modula-2 and Turbo Pascal version 4 features.

Feature	Logitech Modula-2	Turbo Pascal version 4
Support for the Hercules card.	Not obtainable.	Yes, Borland's Turbo Graphix toolbox (version 4).
User interface	Poor.	Excellent.
Compilation speed.	Very slow.	Very fast.
Separately compilable interface and implementation modules.	Supported (standard Modula-2 feature).	No, Turbo Pascal units have an interface and an implementation part. Units may therefore not use each other mutually.
Opaque types.	Supported (standard Modula-2 feature).	Not supported.
Overlays.	Supported.	Not supported.
Procedures passed as parameters.	Supported (standard Modula-2 feature).	Not supported.
Source code debugger.	Available as part of the system.	Produce MAP files for most standard source code debuggers (e.g. Periscope).

2.7 Choosing a design methodology

Section 2.4 introduced two software design methodologies: top-down-bottom-up, and object-orientated design. Turbo Pascal version 4 supports breaking up code into units; it does not support object-orientated design directly. It is however possible to make use of some of the principles of object-orientated design. Circuit Tutor will be designed using the following approach which will be termed **top-down modular design**:

- Firstly, the main objects will be identified. This will be accomplished by writing down the pseudo-code for the main part of the program. This is similar to the first step to be followed in a top-down design. Objects will be mapped to Turbo Pascal units. A modified modular design chart, similar to that proposed by Wiener (1984, pp.130-134), will be used to represent the overall design.

- Secondly, each unit will be refined in a top-down/bottom-up manner. The structure of each unit will be represented by means of hierarchy charts.

2.8 Identifying the main objects

The main objects can be identified by developing an informal strategy for Circuit Tutor. It should be noted that unnecessary detail has been omitted from the strategy.

The syntax of the pseudo-code is similar to that of Modula-2. Control statements are all ended in appropriate END statements (ENDIF, ENDWHILE, ENDCASE etc.). To distinguish control statements they have been capitalised. Underlined functions are further expanded.

```
ALGORITHM tutor;
  initialise program and set up menu structure;
  clear the StopProgram and SolveCctOnce flags;
  LOOP
    IF (the user has typed a keyboard character) THEN
      ProcessKbdInput;  (* Subroutine *)
    ENDIF;
    IF (a message is not displayed) and (circuit has been loaded) and (circuit
    parameters have changed or SolveCctOnce flag has been set) THEN
      display "solving circuits" message to the user;
      solve the circuit outputs and clear SolveCctOnce flag;
      Update the values of the circuit outputs;
      IF (circuit outputs are valid) THEN
        update the meters;
        update the graph;
      ELSE
        write an error message to the user;
      ENDIF;
    ENDIF;
    update the time display;
    update the memory available display;
    IF (StopProgram flag has been set) THEN
      halt the program;
    ENDIF;
  ENDLOOP;
END tutor.
```

```
ALGORITHM ProcessKbdInput;
  CASE (keyboard character) OF
    AltF1: display context-sensitive help message
  ¦ F1:    display general help message
  ¦ F2:    solve circuit outputs
  ¦ F3:    Toggle between default and quick mode
  ¦ F4:    display the menu for the parameter selected in the parameter window
  ¦ F7:    page the contents of the outputs window up
  ¦ F8:    page the contents of the outputs window down
  ¦ F9:    move the circuit parameters cursor up
  ¦ F10:   move the circuit parameters cursor down
  ¦ AltR:  draw the main menu
  ¦ AltC:  draw the circuit menu
  ¦ AltM:  draw the main menu
  ¦ Alt1:  draw the meter 1 menu
  ¦ Alt2:  draw the meter 2 menu
  ¦ Alt3:  draw the meter 3 menu
  ¦ AltG:  draw the graph menu
  ¦ AltX:  draw the graph x-axis menu
  ¦ AltY:  draw the graph y-axis menu
  ¦ AltP:  draw the circuit parameters menu
  ¦ AltO:  draw the circuit outputs menu
  ¦ AltS:  draw the select circuit menu
  ¦ AltQ:  Set the StopProgram flag
  ENDCASE;
  IF (a menu was selected for the above case statement) THEN
    exit from ProcessKbdInput;
  ELSIF (keyboard character is a '+'/'-' character) and (no message displayed)
  and (the program is not in text input mode) THEN
    increment/decrement circuit parameter;
  ELSIF (a message is currently displayed) THEN
    CASE (keyboard character) OF
      DownArr:   scroll message contents down
    ¦ UpArr:     scroll message contents up
    ¦ PgUp:      page message contents up
    ¦ PgDn:      page message contents down
    ¦ Home:      goto top of message
    ¦ EndKey:    goto end of message
    ¦ Esc, CR:   clear the displayed message
    ENDCASE;
  ELSIF (scroll lock is on) and (the program is in the default or text input
  state) THEN
    CASE (keyboard character) OF
      DownArr:   move the menu down
    ¦ UpArr:     move the menu up
    ¦ RightArr:  move the menu to the right
    ¦ LeftArr:   move the menu to the left
    ENDCASE;
```

```
  ELSIF (the program is in the default state) THEN
    CASE (keyboard character) OF
      DownArr:  move the selected menu cursor down
    | UpArr:    move the selected menu cursor up
    | CR:
        get the menu item selected by user and associated task numbers;
        CASE (menu item category) OF
          Router: update the contents of the user fields for the next menu and
                  then select the next menu
        | Field:  put the program in text input state
        | Toggle: ExecuteTask to toggle the value of the menu item
        | Task:   ExecuteTask
        ENDCASE;
    | Esc:      update the user fields for the menu up and select it
    ENDCASE;
  ELSIF (the program is in the text input state) THEN
    CASE (keyboard character) OF
      CR, Esc:
        get the menu item selected and associated task numbers;
        ExecuteTask;
        set the new value of the user field;
        put the program in default state
      ELSE send character to then menu module
    ENDCASE;
  ELSIF (the program is in quick (On-circuit) mode) THEN
    CASE (keyboard character) OF
      DownArr:  move circuit diagram cursor down and select circuit parameter
    | UpArr:    move circuit diagram cursor up and select circuit parameter
    | LeftArr:  move circuit diagram cursor left and select circuit parameter
    | RightArr: move circuit diagram cursor right and select circuit parameter
    | CR:
        put the program in default state;
        update the user fields for the menu associated with the circuit
        parameter selected in the circuit parameters window and display it
    | Esc:
        put the program in default state and restore the menu displayed before
        quick mode was entered;
    ENDCASE;
  ENDIF;
END ProcessKbdInput;
```

```
ALGORITH ExecuteTask(TASKNUMBER; TASKPARAMETERS);
  CASE TASKNUMBER OF
    TaskSetPar:
      assign new value to circuit parameter
  | TaskSetOutput:
      assign new value to circuit output
  | TaskSetMeterInput:
      associate new circuit output with meter input
  | TaskSetMeterScale:
      assign new value to meter scale
  | TaskSetGraphInput:
      associate new circuit output with graph input
  | TaskSetGraphScale:
      assign new value to graph axis scale
  | TaskSelectCircuit:
      draw circuit diagram;
      re-initialise program;
      display circuit parameters in circuit parameters window;
      display circuit output identifiers in circuit outputs window;
      display CIRCUIT menu
  | TaskQuitProgram:
      set StopProgram flag
  | TaskSelOnCctMode:
      put program in on-circuit (quick) mode
  | TaskEraseGraph:
      erase graph
  | TaskDispInfo:
      display message
  | TaskResetCctPars:
      restore the default values of circuit parameters
  | TaskSolveCctPars:
      set the SolveCctOnce flag
  | TaskToggleMarkPoints:
      toggle graph mark plotted points option on/off
  | TaskToggleConnPoints:
      toggle graph connect plotted points option on/off
  | TaskToggleAutoSolve:
      toggle automatic solving of circuit outputs on/off
  | TaskGetPar:
      get present, minimum, maximum or step value of cct parameter
  | TaskGetOutput:
      get the present value of a circuit output
  | TaskGetMeterInput:
      get the circuit output associated with the meter inputs
  | TaskGetMeterScale:
      get the value of the meter scale
  | TaskGetGraphInput:
      get the circuit output associated with the graph inputs
  | TaskGetGraphScale:
      get the graph axis scale values
  ENDCASE;
END ExecuteTask;
```

2.9 Preparing a modular design chart

From the preceding section it is possible to identify the main objects and the operations on these objects. A modular design chart will now be drawn up to indicate all the modules and their interdependency. A list of objects and related data and functional abstractions follow:

Object	Functions
Graph	• Set/get circuit outputs associated with graph inputs. • Set/get the graph axis settings. • Draw the graph. • Toggle 'connect plotted points' option. • Toggle 'mark plotted points' option. • Plot a point.
Meters	• Set/get circuit outputs associated with meter inputs. • Set/get the meter full-scale value. • Deflect the meter needle. • Update the meter display.
Menus	• Initialise a variable of type MENU. • Add a menu to the menu structure. • Add a menu item to a menu. • Get menu item category. • Get task numbers associated with menu item. • Get help message number for menu item. • Get the user field for menu item. • Dispose of all menu items for a menu. • Restore the active menu. • Move a menu. • Move the cursor for displayed menu. • Set the inverse video for user field on or off. • Echo a character to the user field of the menu. • Display the help message for the selected menu item. • Set the value of the user field. • Display a specific menu. • Get active menu. • Get the menu one level up. • Get the menu one level down. • Get a list of all task numbers for the menu. • Clear the displayed menu.
Circuit Parameters (Pars unit)	• Define a circuit parameter and extend the menu structure to include parameter. • Select a parameter. • Increase/decrease selected parameter. • Select the circuit parameter 'under' the circuit diagram cursor. • Set/get the value of a parameter. • Get the menu for selected parameter.

Circuit Outputs (Outputs unit)	▪ Define a circuit output and extend the menu structure to include output. ▪ Set/get the circuit output value. ▪ Update the circuit outputs window.
Circuit Diagram (CctDgm unit)	▪ Identify circuit title and circuit diagram file. ▪ Load the circuit diagram from disk and draw it. ▪ Move the circuit diagram cursor. ▪ Get the position of the circuit diagram cursor.
User messages (UserMsgs unit)	▪ Load the message file into RAM. ▪ Display a message. ▪ Clear a displayed message. ▪ Get the number of the displayed message. ▪ Display an error message. ▪ Scroll/page the displayed message.

In addition to units required to represent the above-mentioned objects the following units are also required:

- Main unit. In Turbo Pascal this is the program unit. The pseudo-code for the main unit has already been presented. Basically this 'unit' is responsible for initialising the program and scheduling all tasks.

- Cct<circuit number> units. These units export the procedure for solving the circuit outputs.

- Misc unit. Contains miscellaneous functions.

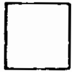
- MMIGlobals unit. Global constants, types and variables.

- HGC unit. Contains all the Turbo Graphix Toolbox routines.

- Chr8x8 unit. Graphics support for 8x8 pixel characters.

CIRCUIT TUTOR UNITS IN DETAIL

This chapter describes each of the Circuit Tutor units in detail. A hierarchy chart is presented for each unit. Hierarchy charts indicates the internal structure of the unit. When a routine (Pascal procedure or function) is exported, the routine is indicated in a double line box.



routine not exported

routine exported

Routines and data declared at the outermost unit level are all individually described. Where necessary routines are described by means of pseudo-code.

3.1 The main (program) unit

The main unit is responsible for initialising/re-initialising the program and thereafter scheduling all tasks.

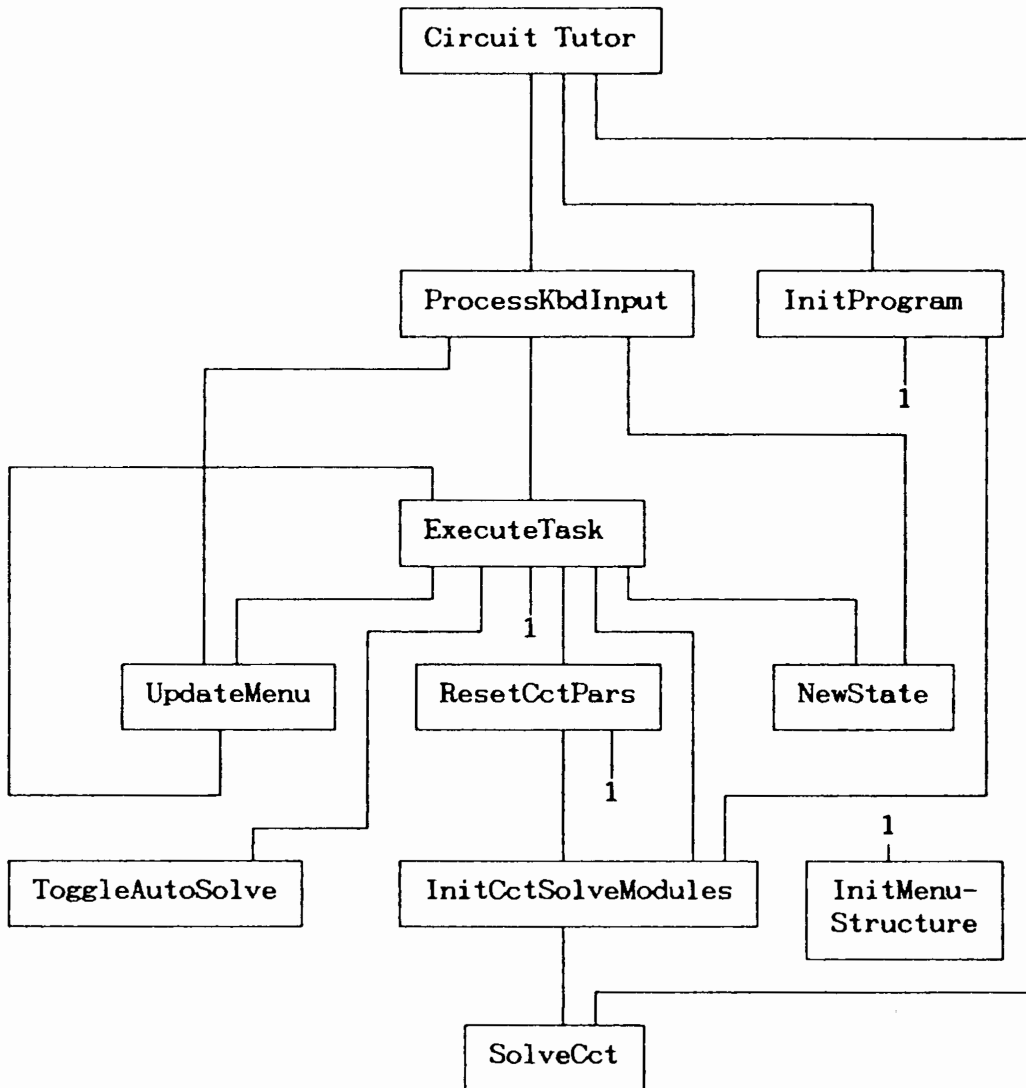


Figure 3-1. Structure of the Main unit.

3.1.1 Description of routines

ProcessKbdInput procedure

Declaration	Procedure ProcessKbdInput;
Function	Process all keyboard input. See chapter 2 for the pseudo-code of this procedure.

InitProgram procedure

Declaration	Procedure InitProgram
Function	Performs a complete initialisation of Circuit Tutor.

ExecuteTask procedure

Declaration

```
Procedure ExecuteTask (Tsk, TskOpt: integer;
                       ReadOnly: boolean;
                       VAR IOStr: string);
```

Function

Executes task number *Tsk* option *TskOpt*. The MMIGlobals unit contain the constant identifiers for all tasks. *TskOpt* is used to pass additional information to the procedure to be executed. *ReadOnly* must be set if the task to be executed is to examine the current state of a toggle without toggling it. *IOStr* passes information to/from the procedure to be executed. *IOStr* is normally used to pass the new value of the menu item field to the destination procedure. The destination procedure will process the value of *IOStr* and then return the new value e.g. when the new value is illegal the returned value of *IOStr* will contain the previous legal value. See chapter 2 for the pseudo-code of this procedure.

UpdateMenu procedure

Declaration	UpdateMenu (M: Menu);
Function	Update all the user input fields of menu ***M***.

ProcessKbdInput procedure

Declaration	Procedure ProcessKbdInput;
Function	Process all keyboard input. See chapter 2 for the pseudo-code of this procedure.

InitProgram procedure

Declaration	Procedure InitProgram
Function	Performs a complete initialisation of Circuit Tutor.

ExecuteTask procedure

Declaration

```
Procedure ExecuteTask (Tsk, TskOpt: integer;
                       ReadOnly: boolean;
                       VAR IOStr: string);
```

Function

Executes task number *Tsk* option *TskOpt*. The MMIGlobals unit contain the constant identifiers for all tasks. *TskOpt* is used to pass additional information to the procedure to be executed. *ReadOnly* must be set if the task to be executed is to examine the current state of a toggle without toggling it. *IOStr* passes information to/from the procedure to be executed. *IOStr* is normally used to pass the new value of the menu item field to the destination procedure. The destination procedure will process the value of *IOStr* and then return the new value e.g. when the new value is illegal the returned value of *IOStr* will contain the previous legal value. See chapter 2 for the pseudo-code of this procedure.

UpdateMenu procedure

Declaration	UpdateMenu (M: Menu);
Function	Update all the user input fields of menu ***M***.

SolveCircuit procedure

Declaration	Procedure SolveCircuit (Option: CctSolveOption);
Function	Executes SolveCct<cct no.>(*Option*) for the selected circuit.
Example	If circuit number 4 has been selected then SolveCct4(*Option*) is executed.

3.2 The Graph unit

The Graph unit provides routines to create and manipulate the graph displayed in the bottom left corner of the screen. The structure of this unit is represented in figure 3-2.

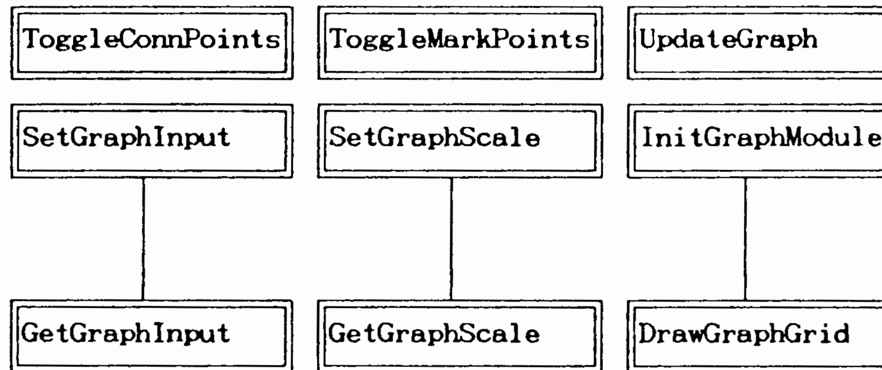


figure 3-2. Structure of the Graph unit.

ToggleConnPoints procedure

Declaration

```
Procedure ToggleConnPoints (ChangeState: boolean;
                            VAR state  : boolean);
```

Function

If the value returned by *state* is true then the plotted graph points will be connected by line segments. To toggle this feature on/off *ChangeState* must be true.

ToggleMarkPoints procedure

Declaration

```
Procedure ToggleMarkPoints (ChangeState: boolean;
                            VAR state  : boolean);
```

Function

If the value returned by *state* is true the plotted graph points will be marked with an *x*. Toggle this feature on/off *ChangeState* must be true.

SetGraphInput procedure

Declaration

```
Procedure SetGraphInput (InputNo      : byte;
                         VAR CctOutput: string);
```

Function

Assign a circuit output identifier *CctOutput* to a graph x-axis/y-axis input. *InputNo* is used to select the graph input:

InputNo	Graph input
1	x-axis positive
2	x-axis negative
3	y-axis positive
4	y-axis negative

If a graph axis input is not to be connected to any circuit output then *CctOutput* must be *NC* (i.e. Not Connected). The Circuit output identifier (except *NC*) is displayed next to the graph axis. The value returned by *CctOutput* will be value actually assigned - if an illegal circuit output was passed to this routine, then the value returned will be the last legal circuit output.

GetGraphInput function

Declaration Function GetGraphInput (InputNo: byte): string;

Function Returns the circuit output identifier assigned to the graph axis input *InputNo* by SetGraphInput. If no circuit output has been assigned then a value of *NC* (Not Connected) will be returned.

SetGraphScale procedure

Declaration Procedure SetGraphScale (option: byte; VAR GScale: real);

Function Configures the x and y axis of the graph. The exact function is dependant on the value of *option*:

option	function
0	initialise axis
1	set x-axis minimum value
2	set x-axis increment per division
3	set x-axis multiplier
4..6	as for 1..3, but for y-axis

For each of the above functions the value is passed to this procedure by *GScale*. If the value is within range then the associated function is performed and the value returned by *GScale* will be the same. If the value is illegal, then the last legal value will be returned. The axis scale values are displayed next to the graph axis.

GetGraphScale function

Declaration Function GetGraphScale (option: byte): real;

Function Gets the current value of the graph axis scale. *Option* is used to select the scale parameter as for SetGraphScale.

InitGraphModule procedure

Declaration Procedure InitGraphModule (FullInit: boolean);

Function When *FullInit* is *true* all the graph variables are initialised and the graph is drawn. When *FullInit* is *false* then the graph inputs are de-assigned (i.e. assigned to *NC*).

DrawGraphGrid procedure

Declaration Procedure DrawGraphGrid

Function Displays the grid of the graph. The axis labels and scale values are not affected. The first time this routine is invoked the grid will drawn and then stored using the graphix toolbox routine StoreWindow. Subsequent calls will just restore the grid from memory.

UpdateGraph procedure

Declaration Procedure UpdateGraph;

Function If the graph has been set up correctly with the *SetGraphInput* and *SetGraphScale* routines, then the value of the assigned circuit outputs will be plotted and stored. The first time this routine is called the plotted point will be marked (if mark points option has been selected) and stored. Subsequent calls will draw a line segment between the previous plotted point and the current point if the connect points option has been selected. If the mark points option has been selected, plotted points will be marked with a diagonal cross *X*, if not, plotted points are invisible.

The Meters unit contains routines to create and to manipulate the three simulated analogue/digital meters displayed in the lower centre of the screen. The lay-out of this unit is given in figure 3-3.

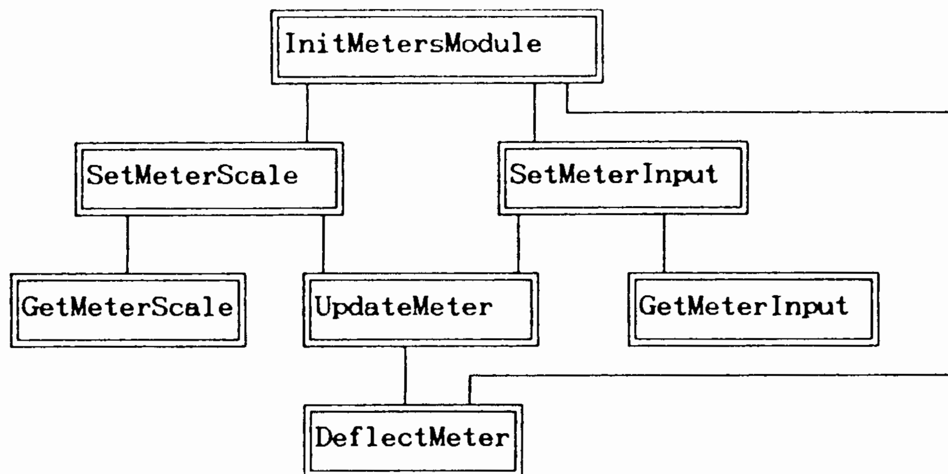


figure 3-3. Structure of the Meters unit.

InitMetersModule procedure

Declaration — Procedure InitMetersModule (FullInit: boolean);

Function — When *FullInit* is *true* then the meters are drawn and all meter variables initialised. When *FullInit* is false then only the meter input assignment (to circuit outputs) are initialised.

SetMeterScale procedure

Declaration — Procedure SetMeterScale (MeterNo: byte; VAR Scale: real);

Function — Sets the full-scale deflection *scale* for meter number *MeterNo*. The absolute value of *scale* must lie in the range 10^{-12} to 10^{6}. If *scale* is valid then the new value is displayed on the meter, and the meter deflection adjusted.

GetMeterScale function

Declaration — Function GetMeterScale (MeterNo): real;

Function — Gets the current full-scale deflection for meter number *MeterNo*.

SetMeterInput procedure

Declaration

```
Procedure SetMeterInput (MeterNo       : byte;
                         InputNo       : byte;
                         VAR OutputName: string);
```

Function — Assigns a circuit output identifier to a meter input. *MeterNo* is the meter number (1..3). The positive input of the meter is selected when InputNo=1, and the negative input when InputNo=2. *OutputName* must be a circuit output identifier that has been defined with the DefParameter procedure.

GetMeterInput function

Declaration — Function GetMeterInput (MeterNo, InputNo: byte): string;

Function — Returns the current circuit output identifier assigned to the meter input.

UpdateMeter procedure

Declaration — Procedure UpdateMeter (option: byte);

Function — If the meter inputs have been assigned to circuit outputs then meter number *option* is updated. If option=0 then all the meters are updated. Updating the meter entails:

1. Deflecting the meter *needle* e.g. if the full-scale deflection=2 and the difference between the two circuit outputs assigned to the meter inputs is 1, then the needle will be moved to the middle of the meter.

2. Updating the digital display i.e. for above example *1.00e+00* will be displayed.

DeflectMeter procedure

Declaration — Procedure DeflectMeter (Meter: byte; InputVal: real);

Function — Deflect the meter *needle* of meter number *meter*. *InputVal* is the unscaled input value i.e. the difference between the two circuit outputs assigned to the meter. This value will be scaled with the full-scale deflection.

This unit provides the routines to define and manipulate circuit parameters and the circuit parameters status window *CCT PARAMETERS*. The structure of this unit is given in figure 3-4.

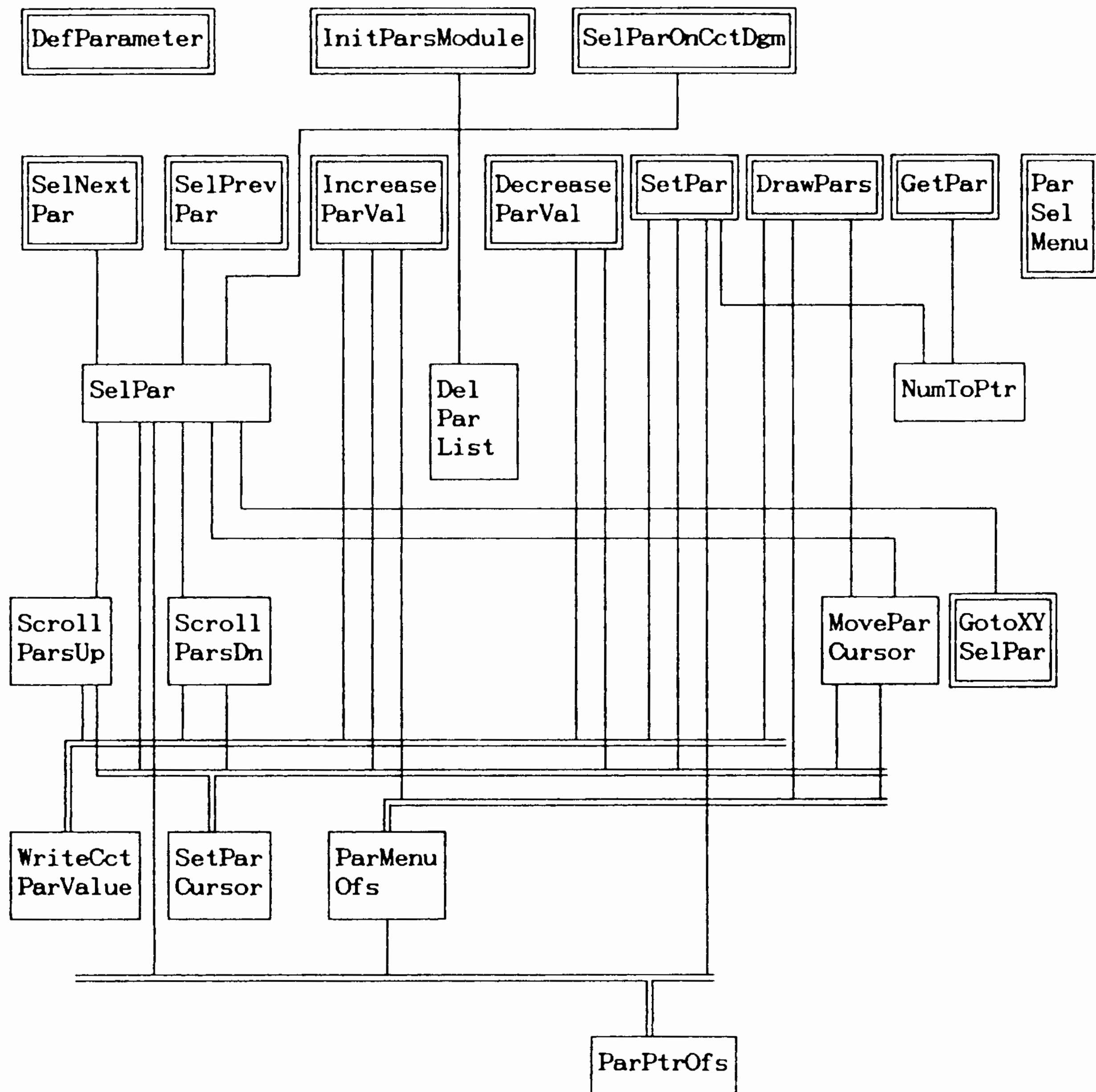


Figure 3-4. Structure of the Pars unit.

Circuit parameters are stored as a linked list in the heap memory i.e. these records are accessed by means of pointer variables.

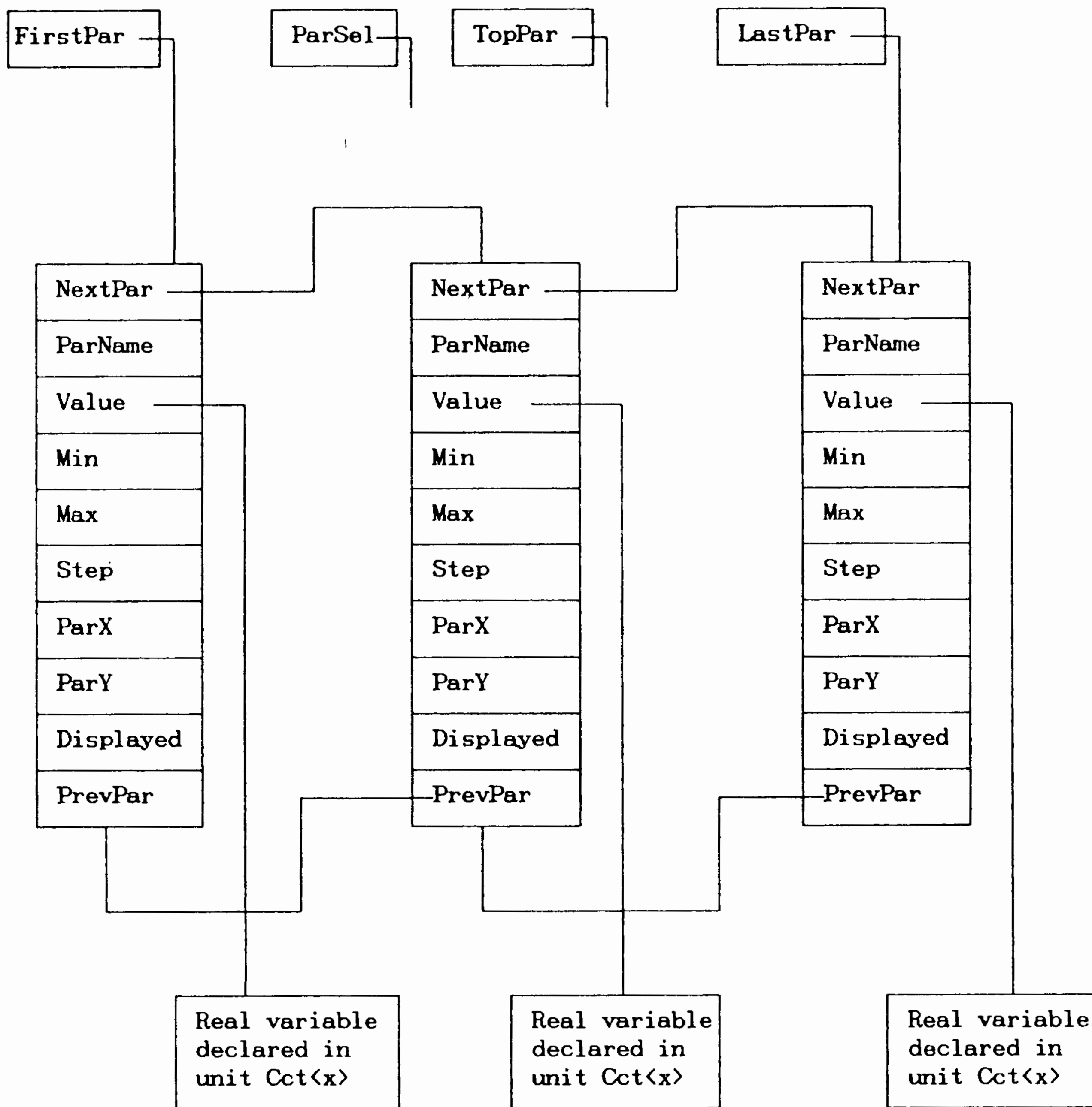


Figure 3-5. Linked list for storing the circuit parameter data.

DefParameter procedure

Declaration

```
Procedure DefParameter (VAR UserVar : real;
                            par_name : string;
                            ValInit  : real;
                            ValMin   : real;
                            ValMax   : real;
                            ValStep  : real;
                            CctX     : byte;
                            CctY     : byte);
```

Function

- Creates a record for storing circuit parameter information. The record is added to the linked list of circuit parameter records (see figure 3-5). The variable *UserVar* must be declared in the unit Cct<x> and must be used to compute the circuit outputs. This routine gets the address of *UserVar* and sets up a pointer to it, thereby giving Circuit Tutor access to it at run-time. For example, when the user increments the parameter at run-time using the + and - keys, procedures *IncreaseParVal* and *DecreaseParVal* will increase/decrease the value pointed to by *ParSel^.Value^* which is the same variable used by the circuit solver routine Cct<x> to calculate the circuit outputs.

- Define a menu for each parameter to give the user access to the circuit parameter via the menu at run-time.

- Set up the value of the pointer array variable (*DgmParPtr*) which stores the circuit diagram co-ordinate / parameter record reference e.g. *DgmParPtr[4,6]* will point to the record of the parameter whose symbol is drawn on circuit diagram co-ordinates [4,6].

InitParsModule procedure

Declaration Procedure InitParsModule (FullInit: boolean);

Function Initialises the Pars unit. When *FullInit* is *false* then the circuit parameters status window *CCT PARAMETERS* on the left-hand side of the screen is drawn.

When *FullInit* is *true* then

- delete the contents of the circuit parameters status window;
- release heap memory used to store the circuit parameters linked list;
- initialise the pointer array (*DgmParPtr*) for storing the circuit diagram co-ordinate / parameter record reference.

SelParOnCctDgm procedure

Declaration Procedure SelParOnCctDgm;

Function Selects the circuit parameter according to the current position of the circuit diagram cursor.

SelNextPar procedure

Declaration Procedure SelNextPar;

Function Select the next circuit parameter in the linked list (see figure 3-5).

SelPrevPar procedure

Declaration Procedure SelPrevPar;

Function Select the previous circuit parameter in the linked list (see figure 3-5).

IncreaseParVal procedure

Declaration Procedure IncreaseParVal;

Function

- Add the circuit parameter step value to the present value.
- Update the value of the circuit parameter for the circuit parameter status window.

DecreaseParVal procedure

Declaration	Procedure DecreaseParVal;
Function	As for Procedure IncreaseParVal, but the step value is deducted.

SetPar procedure

Declaration
: Procedure SetPar (option: byte; VAR NewVal);

Function
: Modifies the present, minimum, maximum or step value (depending on the value of *option*) of a circuit parameter currently accessed via it's circuit parameter menu i.e. the active menu.

OPTION	CCT PARAMETER	LEGAL VALUES
1	Present value	Min $\leq$ *NewVal* $\leq$ Max
2	Minimum value	*NewVal* $\leq$ PresVal
3	Maximum value	*NewVal* $\geq$ PresVal
4	Step value	*NewVal* $\leq$ (Max - Min)

If *NewVal* is an illegal value it will be rejected, a long beep will be issued, and the last legal value will be assigned to *NewVal*. When the present value of the circuit parameter is modified, the circuit parameter status window contents will be updated.

DrawPars procedure

Declaration
: Procedure DrawPars;

Function
: Draw the contents of the circuit parameters status window. Parameter identifiers and their values are written for all the parameters that can fit in the window, starting with the first circuit parameter of the linked list.

GetPar function

Declaration
: Function GetPar (option: byte): real;

Function
: Returns the present, minimum, maximum or step value of the circuit parameter selected via the active menu. Refer to procedure SetPar for a description of *option*.

SelPar procedure

Declaration   Procedure SelPar (Ptr: ParPtrType);

Function

- The parameter record pointed to by *Ptr* will become the current selected parameter i.e. the grey + - keys will increment decrement this parameter.
- The newly selected parameter will be underlined in the circuit parameters window. If necessary the window contents will be scrolled to display the selected parameter.

DelParList procedure

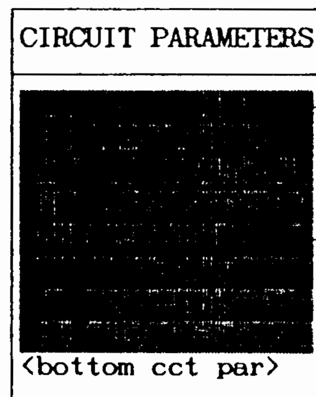
Declaration   Procedure DelParList

Function   Release the heap memory used for storing the linked list of parameter records.

ScrollParsUp procedure

Declaration   Procedure ScrollParsUp;

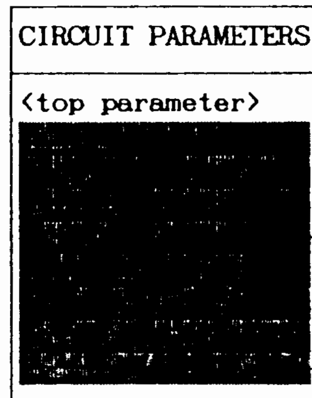
Function   The contents of the circuit parameters status window, *CCT PARAMETERS*, will be moved one line down, a new top parameter will be displayed. This is accomplished by defining a window within the circuit parameters window and then moving the window one line down.



ScrollParsDn procedure

Declaration Procedure ScrollParsDn;

Function As for procedure *ScrollParsUp*, but the contents of the circuit parameters status window, *CCT PARAMETERS*, will be scrolled one line up and a new bottom parameter will be displayed.



This window is moved up thereby over-writing the existing top parameter.

One new circuit parameter is displayed here after moving the window.

MoveParCursor procedure

Declaration Procedure MoveParCursor;

Function Moves the circuit parameter underlining to the newly selected circuit parameter in the *CCT PARAMETERS* window.

GotoXYSelPar procedure

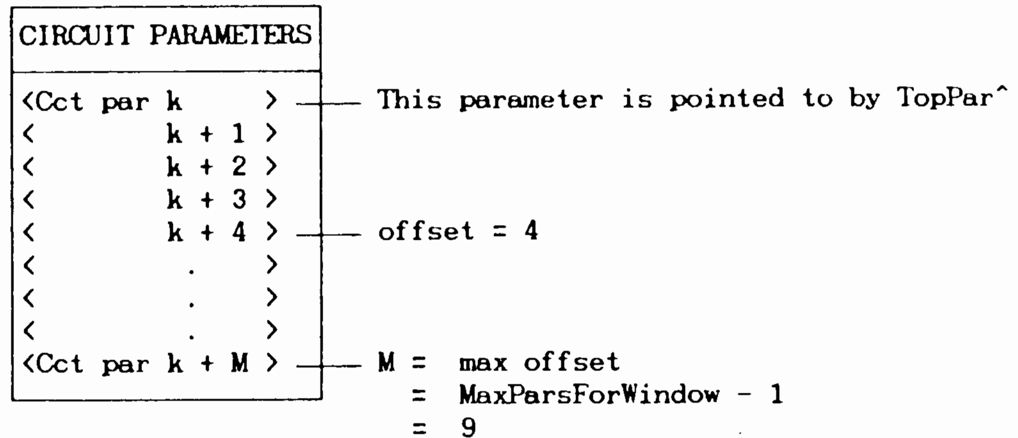
Declaration Procedure GotoXYSelPar;

Function Position the circuit diagram cursor on the circuit diagram co-ordinates [*ParX*,*ParY*] of the selected circuit parameter record (see figure 3-5).

WriteCctParValue procedure

Declaration Procedure WriteCctParValue (offset: byte);

Function Writes the circuit parameter identifier and it's value at *offset* lines from the top of the *CCT PARAMETERS* window.



SetParCursor procedure

Declaration Procedure SetParCursor (state: boolean);

Function When *state=true* then the selected circuit parameter is underlined in the *CCT PARAMETERS* window. When *state=false* then the selected circuit parameter underlining is removed.

ParMenuOfs function

Declaration Function ParMenuOfs: integer;

Function Returns the offset of the selected circuit parameter in the *CCT PARAMETERS* window.

See also *WriteCctParValue* procedure for a description of the window offset.

ParPtrOfs function

Declaration Function ParPtrOfs (ptr: ParPtrType): integer;

Function Returns the order of the circuit parameter record pointed to by *ptr*. The first parameter in the linked list will be of order 0 i.e. for *ptr=FirstPar*, the second parameter will have an order of 1, etc..

ParSelMenu function

Declaration Function ParSelMenu: Menu;

Function Returns the circuit parameter menu for the currently selected circuit parameter.

NumToPtr function

Declaration Function NumToPtr(ParNum): ParPtrType;

Function Returns a pointer that points to circuit parameter record number *ParNum*.

This unit provides routines to create and manipulate the circuit diagram.

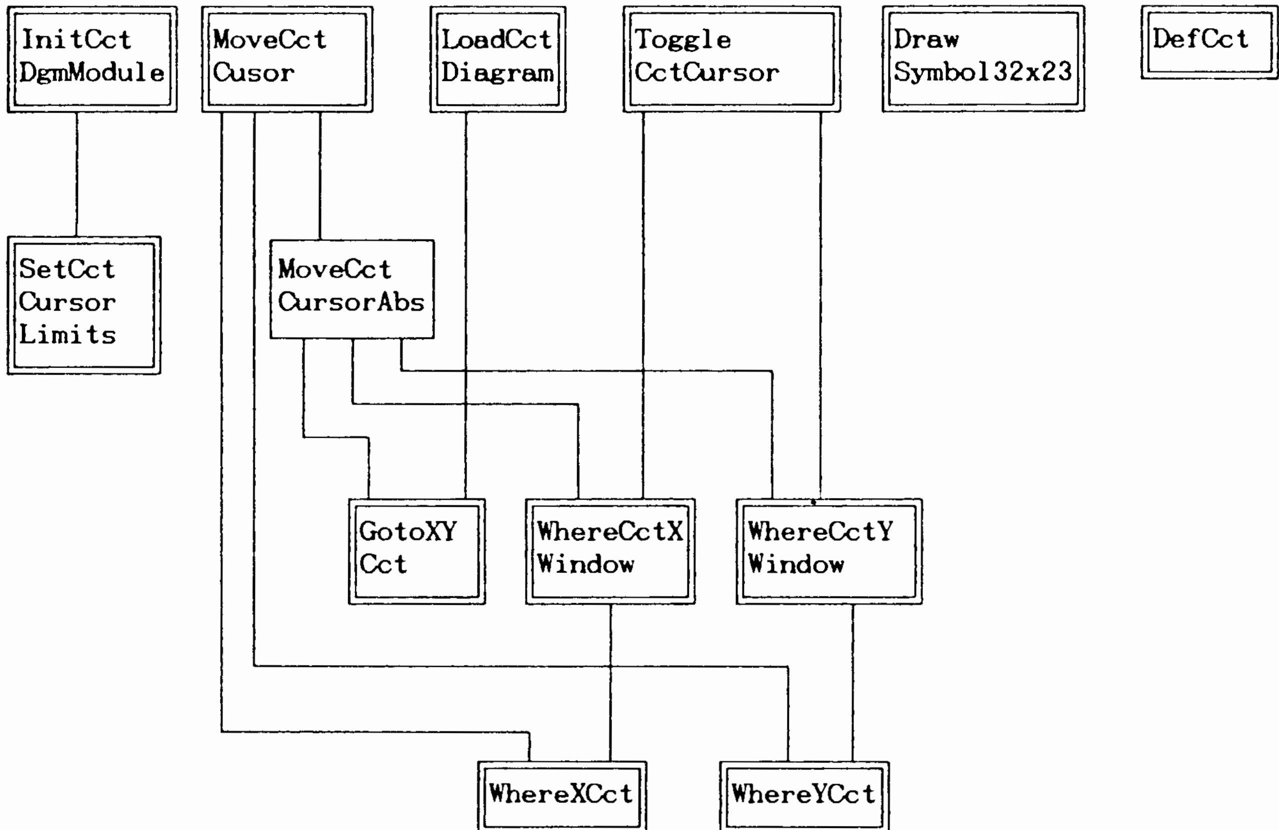


Figure 3-6. Structure of the CctDgm unit.

3.5.1 Co-ordinate systems used by the CctDgm unit

An understanding of the various co-ordinate systems is necessary to understand this unit. These are:

- **Screen co-ordinates.** Each pixel position is given a co-ordinate position. The pixel in the top-left corner has co-ordinates [0,0] and the pixel in the bottom-right corner has co-ordinates [(max x-resolution)-1, (max y-resolution)-1] i.e. [719,349] for the Hercules graphics card.

- **Window co-ordinates.** These are similar to screen co-ordinates except that the x-axis of the screen is divided in 8-bit increments i.e. byte-sized. The co-ordinate range will be from [0,0] to [89,349].

- **Circuit diagram co-ordinates.** This co-ordinate system ranges from [1,1] to [17,8] and is demonstrated in figure 3-7. Each block is 32H x 23V pixels and is the size of a circuit diagram symbol (e.g. a resistor, transistor, etc.). In quick mode (on-circuit mode) a circuit parameter can be selected by highlighting it's circuit diagram symbol. This highlighted circuit diagram symbol is defined as the circuit diagram cursor.

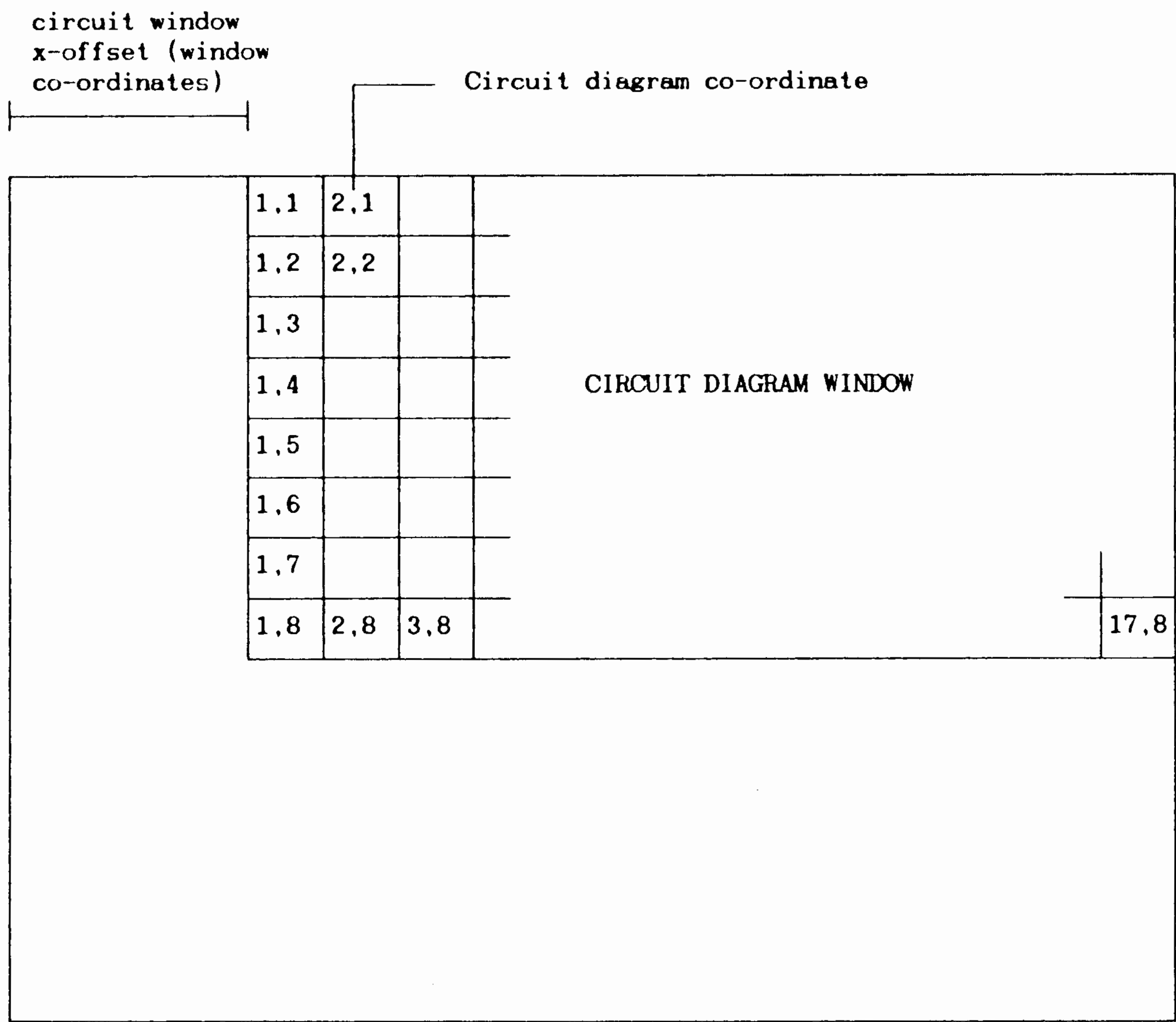


Figure 3-7. The circuit diagram co-ordinate system.

InitCctDgmModule procedure

Declaration Procedure InitCctDgmModule;

Function Initialise the CctDgm unit, which includes:
- initialising unit variables;
- defining the circuit diagram window dimensions; and
- defining the circuit cursor domain.

MoveCctCursor procedure

Declaration Procedure MoveCctCursor (dx, dy: integer);

Function Move the circuit cursor relative to it's present position. [*dx,dy*] is the cursor offset in circuit diagram co-ordinates.

LoadCctDiagram procedure

Declaration Procedure LoadCctDiagram (CctNo: CctNumber);

Function The circuit diagram for circuit number *CctNo* will be loaded from the disk(ette) file associated with it and displayed in the top right-hand corner of the screen. The circuit diagram file (*.dgm* file extension) is produced with the CctDraw utility program.

ToggleCctCursor procedure

Declaration Procedure ToggleCctCursor;

Function Toggle the circuit cursor on/off.

DefCct procedure

Declaration — Procedure DefCct (CctName, FileName: string);

Function — This circuit has two funtions, namely:
- Declaring the name of the circuit, *CctName* which will be displayed in the CIRCUIT SELECT menu.
- Identifying the file name *FileName* where the circuit diagram for the circuit has been saved. This file must have been created with Circuit Draw.

Remarks — This routine must be called from the *IdentifyCct* section of the case statement of the routines to solve the circuit.

SetCctCursorLimits procedure

Declaration — Procedure SetCctCursorLimits (xmin,xmax,ymin,ymax,xofs:integer);

Function — *xmin, xmax, ymin, ymax* define the circuit diagram cursor limits (circuit diagram co-ordinates). *xofs* is the x-offset of the circuit diagram window in 8-bit divisions (window co-ordinates) from the left of the screen. Note that the circuit diagram window will always be defined with a y-offset of zero i.e. at the top of the screen.

MoveCctCursorAbs procedure

Declaration — Procedure MoveCctCursorAbs (xpos, ypos: integer);

Function — Move the circuit diagram cursor i.e. turn off the inverse video for the symbol presently selected and turn on the inverse video for the circuit symbol at circuit diagram co-ordinates [*xpos, ypos*].

GotoXYCct procedure

Declaration — Procedure GotoXYCct (X, Y: byte);

Function — Specify the next position of the circuit diagram cursor [*X,Y*] (circuit diagram co-ordinates). When the co-ordinates are outside the range set with the *SetCctCursorLimits* procedure then the out-of-range parameter is ignored.

See also — *WhereXCct*, *WhereYCct* functions.

WhereCctXWindow function

Declaration Function WhereCctXWindow: integer;

Function Returns the x-position in window co-ordinates of the circuit diagram cursor.

WhereCctYWindow function

Declaration Function WhereCctYWindow: integer;

Function Returns the y-position in window co-ordinates of the circuit diagram cursor.

WhereXCct function

Declaration Function WhereXCct: byte;

Function Returns the x-position in circuit diagram co-ordinates of the circuit diagram cursor.

WhereYCct function

Declaration Function WhereYCct: byte;

Function Returns the y-position in circuit diagram co-ordinates of the circuit diagram cursor.

The UserMsgs unit provides routines to display messages from the messages file TUTOR.MSG.

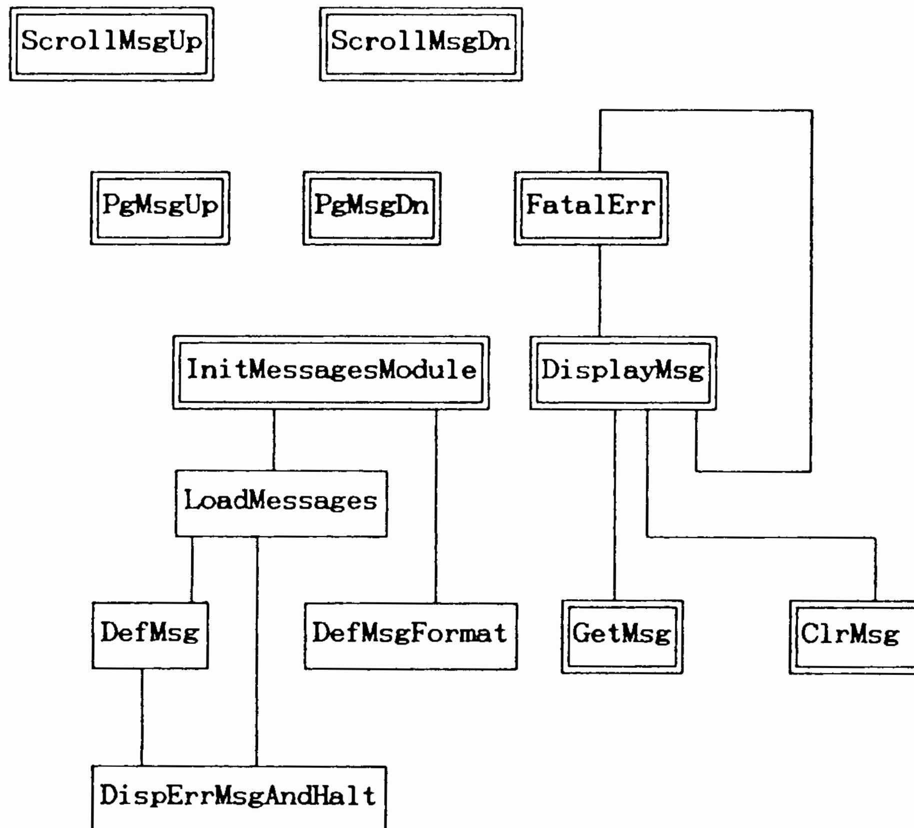


Figure 3-8. Structure of the UserMsgs unit.

The following data objects are declared at the outermost level of the implementation section of UserMsgs.

```
const
  MaxFormats =    1;                 { Maximum number of message formats. }
  MaxMsgs    =   50;                 { Maximum number of messages. }
  MaxBufLines=  650;                 { Lines of text in file TUTOR.MSG. }
type
  String40    =  String[40];
  MsgNo       =  0..MaxMsgs;
  FormatNo    =  1..MaxFormats;
  MsgRec =                           { A record is kept for each message. }
  record
    MsgExist : boolean;              { True if message in TUTOR.MSG. }
    FormNo   : FormatNo;             { Format number for message. }
    MsgHdng  : string40;             { Message heading. }
    Ln1      : integer;              { Index of first line of message. }
    TopLn    : integer;              { Index of first line of message to be
                                       displayed. }
    LastLine : word                  { Index of last line of message. }
  end;

  MsgFormatRec =
  record
    { Text co-ordinates of top left corner of message window. }
    MsgX      : XTxt;
    MsgY      : Ytxt;
    { Message width and height in text co-ordinates. }
    MsgWidth : XTxt
    MsgHeight: Ytxt;
  end;
var
  x1,x2,y1,y2: byte;     { Message window text co-ordinates. }
  BottomLn   : integer;
  MsgFileVar : text;     { Message file buffer. }
  MsgOn      : MsgNo;    { Number of displayed message; 0 if no message
                           displayed. }
  PrevMmiState: MmiStateType;
  FormatArr   : array[FormatNo] of MsgFormatRec;
  MsgArr      : array[MsgNo] of MsgRec;
  MsgBuf      : array[1..MaxBufLines] of string40;
```

InitMessagesModule procedure

Declaration Procedure InitMessagesModule;

Function Initialises UserMsgs unit variables and read the contents of the TUTOR.MSG file into RAM memory.

FatalErr procedure

Declaration Procedure FatalErr(error: byte);

Function Display message number *error* and halts the program when the user presses a key.

LoadMessages procedure

Declaration Procedure LoadMessages;

Function Opens the TUTOR.MSG file and reads it's contents into the variable *MsgBuf*. When the number of text lines exceeds *MaxBufLines*, an error message is displayed and Circuit Tutor is halted. Messages are defined in TUTOR.MSG using the following format for each message:

```
.mn<two-digit message number>
.he<message heading>
<message contents>
.me
```

In addition a format number directive *.fn<format number>* can be entered on any line of TUTOR.MSG to change the format of the displayed message. Circuit Tutor will display an error message and halt if arguments for the *.mn* or *.fn* are out of range or do not comprise of two digits.

Values are assigned to the variable *MsgArr* by invoking the DefMsg procedure.

DefMsg procedure

Declaration	Procedure Defmsg (Msg : integer; Format : integer; Heading : string; FirstLn : integer; LastLn : integer);
Function	Assigns values to the array variable *MsgArr*. *Msg* is the message number. *Format* is the format number of the message. *Heading* is the message window header. *FirstLn* is the number of the first line of message text i.e. the first line of text will be stored in *MsgBuf[FirstLn]*. The last line of the message will be stored in *MsgBuf[LastLn]*. Range checking is done for *Msg* and *Format*. Illegal values will cause Circuit Tutor to display an error message and then halt.

DispErrMsgAndHalt procedure

Declaration	Procedure DispErrMsgAndHalt (ErrorMsg: string);
Function	Return to text mode, write "FATAL ERROR: *ErrorMsg*", and then halt the program.

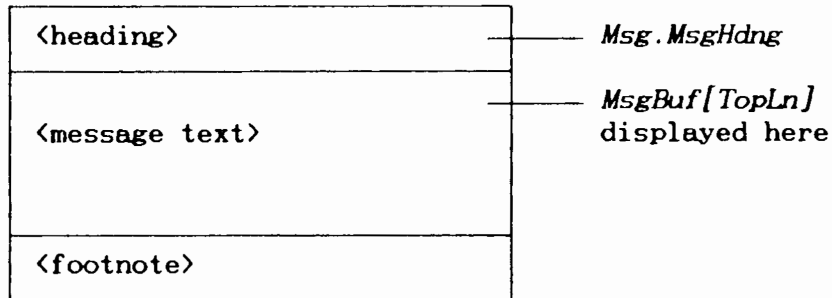
DefMsgFormat procedure

Declaration	Procedure DefMsgFormat(Format : FormatNo; xpos : Xtxt; ypos : Ytxt; WWidth : Xtxt; WHeight : Ytxt);
Function	Define the format of a message i.e. assign values to the array variable *FormatArr[Format]*. *xpos* and *ypos* are the text co-ordinates of the message window top-left corner. *WWidth* and *WHeight* are the width and height respectively, in text co-ordinates, of the message window.

DisplayMsg procedure

Declaration   Procedure DisplayMsg (Msg: MsgNo);

Function   Displays message number *Msg*. The message text displayed will start with line *MsgBuf[TopLn]* (the first time the message is displayed *MsgBuf[TopLn]=MsgBuf[Ln1]*). If the last 'page' of the message is displayed the footnote "ESC/ENTER to clear" is displayed; otherwise "ESC/ENTER to clear. PgUp, PgDn to read" is displayed.



GetMsg function

Declaration   Function GetMsg:byte;

Function   Returns the number of the active message, *MsgOn*. If no message is displayed *GetMsg* = 0.

ClrMsg procedure

Declaration   Procedure ClrMsg;

Function   Clears the displayed message and set *MsgOn* = 0.

PgMsgUp procedure

Declaration   Procedure PgMsgUp (option: byte);

Function

- *Option=0*. Displays the first page of the active message.
- *Option<>0*. Displays the previous page of the active message.

PgMsgDn procedure

Declaration	Procedure PgMsgDn (option: byte);
Function	▪ *Option=0.* Displays the last page of the active message. ▪ *Option<>0.* Displays the next page of the active message.

ScrollMsgUp procedure

Declaration	Procedure ScrollMsgUp;
Function	Scroll the contents of the active message (the displayed message one line up).

ScrollMsgDn procedure

Declaration	Procedure ScrollMsgDn;
Function	Scroll the contents of the active message (the displayed message one line down).

The Chr8x8 unit provides routines the write characters comprising of a 8x8 dot-matrix array to the display.

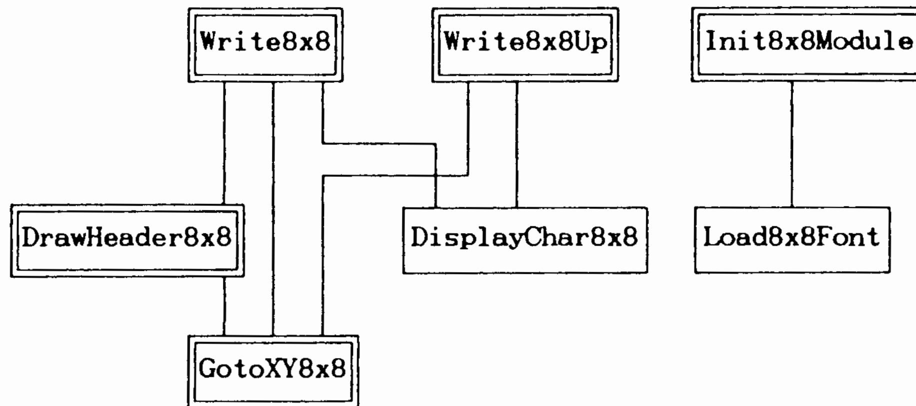


Figure 3-9. Structure of the Chr8x8 unit.

3.7.1 Exported data

Inverse8x8 variable

Declaration Var Inverse8x8 : boolean;

Description If *Inverse8x8* is true then 8x8-font characters are displayed in inverse video.

Overwrite8x8 variable

Declaration Var Overwrite8x8 : boolean;

Description If *Overwrite8x8* is true the existing video memory is overwritten by 8x8 characters. If *Overwrite8x8* is false then 8x8 character pixels and video memory pixels are AND-ed.

3.7.2 Description of routines

Write8x8 procedure

Declaration Procedure Write8x8 (StrOf8x8chars: String);

Function Display the string of characters *StrOf8x8chars*, from left to right, on the screen.

Write8x8up procedure

Declaration Procedure Write8x8Up (StrOf8x8Chars: string);

Function Display the string of characters *StrOf8x8chars*, from bottom to top, on the screen.

Init8x8Module procedure

Declaration Procedure Init8x8Module;

Function Initialise the chr8x8 unit. The 8x8 character fonts are loaded from files 8x8.FON and 8x8UP.FON, and the two exported variables are initialised as follows:

Overwrite8x8:=true; and
Inverse8x8:=false

GotoXY8x8 procedure

Declaration Procedure GotoXY8x8(x:byte; y:integer);

Function Move the (invisible) cursor for 8x8-font characters to window co-ordinate position [*x*,*y*].

DrawHeader8x8 procedure

Declaration Procedure DrawHeader8x8 (TopLeftX: Byte;

```
Procedure DrawHeader8x8 (TopLeftX: Byte;
                         TopLeftY: Integer;
                      WindowWidth: byte;
                         LabelStr: string);
```

Function Draw a label *LabelStr* (which can be used as a window header) in inverse video. *TopLeftX* and *TopLeftY* are the text co-ordinates of the top-left corner of the label. *WindowWidth* is the width of the label.

DisplayChar8x8 procedure

Declaration Procedure DisplayChar8x8(Chr: byte; Up: boolean);

Function A 8x8-font character *Chr* is displayed at the current cursor position. If *Up* is true then the character is orientated on it's side.

Load8x8Font procedure

Declaration Procedure Load8x8Font;

Function Read the contents of the two 8x8-font files, 8x8.FON and 8x8UP.FON, into memory.

3.8 The Misc unit

This unit contains miscellaneous routines not associated with any of the other units. The routines are independent of each other, with the exception of *HaltIfAbsent* which uses *FileExists*. All routines are exported.

3.8.1 Description of routines

Exponent function

Declaration Function Exponent(base,power: real):real;

Function Returns the value $base^{power}$.

SetSwitch procedure

Declaration Procedure SetSwitch (St:string;
VAR flag:boolean;
VAR error:boolean);

Function If *St* is an affirmative string ('Y', 'YES', 'OK') then *flag* is true. If *St* is not an affirmative ('N', 'NO') then *flag* is false. *St* can be lower or uppercase characters. If *St* is not recognised then *error* is true.

StrFormat function

Declaration Function StrFormat(St:String; FldWidth:byte):String;

Function Truncate or pad (with space characters) *St* to fit it into a field *FldWidth* characters wide.

RemoveSpaces function

Declaration Function RemoveSpaces(St:string):string;

Function Remove all spaces from the string *St*.

Log function

Declaration Function log(num:real):Real;

Function Returns $\log_{10}(num)$.

Beep procedure

Declaration Procedure Beep;

Function Emits a 800Hz beep for 50ms.

LongBeep procedure

Declaration Procedure LongBeep;

Function Emits a 400Hz beep for 200ms.

GetTime function

Declaration Function GetTime:string;

Function Returns the present time in the format *hh:mm*.

EngFormat function

Declaration Function EngFormat (R:Real; digits:Integer):String;

Function Convert the real number *R* to a string value that resembles a number in engineering format. The engineering format string will comprise of *digits* ($3 \geq digits \leq 11$) number of significant digits. The following table demonstrates how this function operates:

R	digits	EngFormat (R, digits)
0.001234	3	1.23E-03
0.001234	4	1.234E-03
0.000123	3	123.E-06
123456	4	123.5E+03

ScrollLockOn function

Declaration — Function ScrollLockOn: Boolean;

Function — Checks whether the Scroll lock is on.

ReadStr procedure

Declaration — Procedure ReadStr(Var St:string; Var Abort:boolean);

Function — Read a string from the keyboard into *St*. If the user inputs an Escape character the *Abort* will be true and the value of *St* should be ignored. The backspace key can be used for deleting the last character. Characters can be entered in a field that is defined from the starting position of the text cursor (WhereX function) to the end of the line (column 80).

StrToChr function

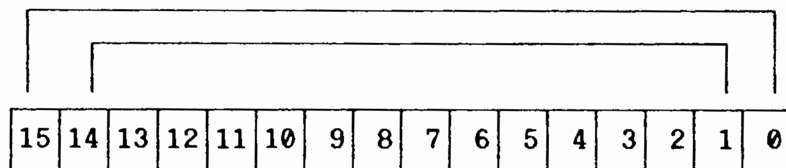
Declaration — Function StrToChr(St:String; StPos:byte):char;

Function — Return the value of the *StPos*'th character of string *St*.

SwapBitsOfWord function

Declaration — Function SwapBitsOfWord(w:integer):integer;

Function — Swap all bits of the word. Swap bits 0 and 15; 1 and 14; etc..



Debug procedure

Declaration	Procedure Debug(DebugMsg:string; DebugNo:LongInt);
Function	Displays a debug message on the screen; wait for the user to type any keyboard character; and then restore the previous display. The debug message number *DebugNo* and debug message *DebugMsg* is displayed inside a debug window.

FileExists procedure

Declaration	Function FileExists(FileName:string):boolean;
Function	Returns TRUE if file *FileName* exists in the default path.

HaltIfAbsent procedure

Declaration	Procedure HaltIfAbsent(filename:string);
Function	If the file *filename* does not exist the program will be halted.

The Outputs unit provides routines to

- access the data structure for storing the circuit output values; and
- manipulate the *CIRCUIT OUTPUTS* window.

The internal structure of this unit is apparent from figure 3-10.

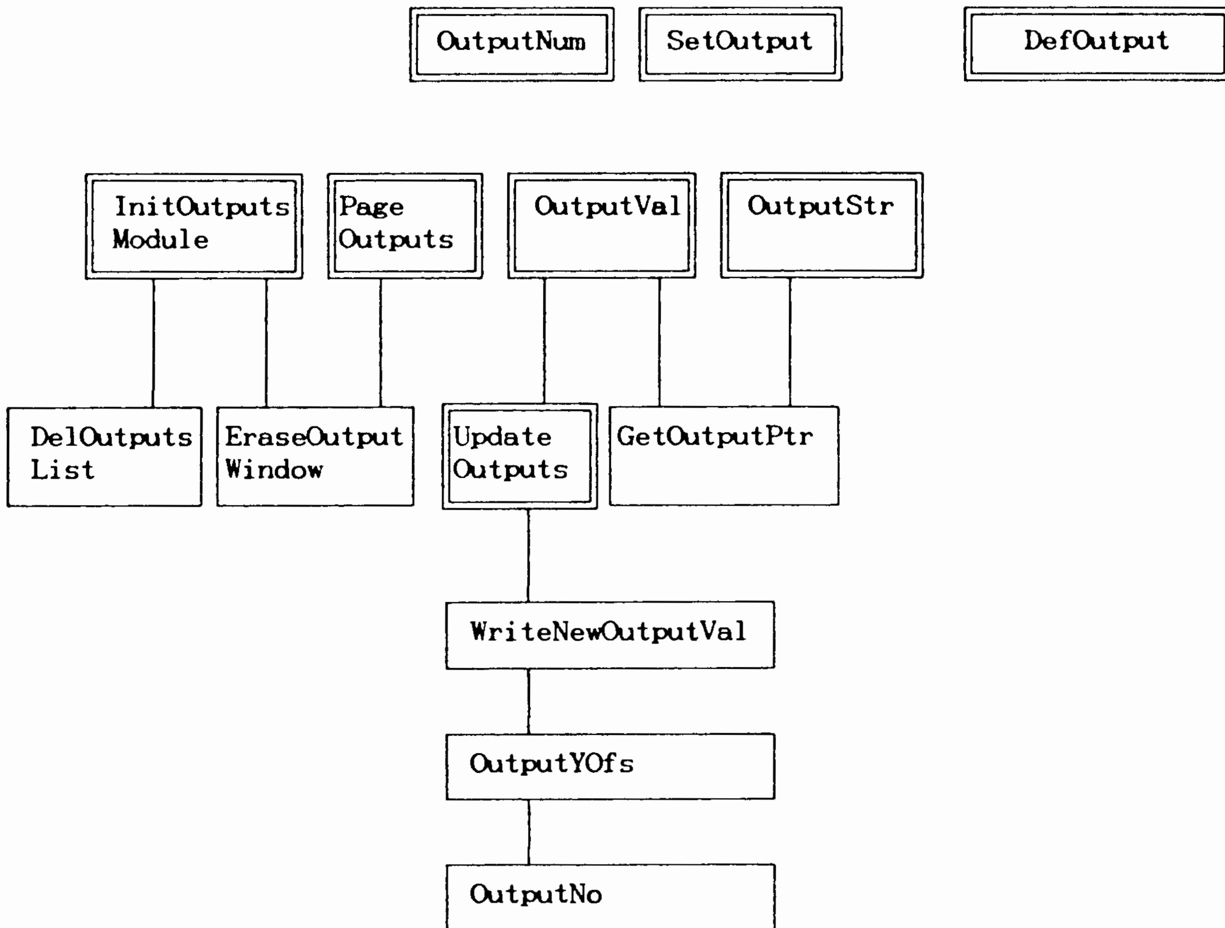


Figure 3-10. Structure of the Outputs unit.

3.9.1 Description of internal data

For each circuit output a record declared as follows is kept:

```
type
  OutputPtrType = ^OutputRec;
  OutputRec=
  record
    PrevOutput,
    NextOutput: OutputPtrType;
    OutputName: String[6];
    OutVal     : ^Real;
  end;
```

Circuit outputs are stored as a linked list in the heap memory. This linked list has the same structure as the linked list for the circuit parameters (refer to the Pars unit). *PrevOutput* and *NextOutput* points to the two records before and after the record considered. *OutputName* is the circuit output identifier string displayed in the *CIRCUIT OUTPUTS* window. *OutVal* points to the value of the real variable declared in unit Cct<x>.

3.9.2 Description of routines

InitOutputsModule procedure

Declaration — Procedure InitOutputsModule (FullInit: boolean);

Function — If *FullInit=true* a full initialisation is performed. The pointers for the linked list of circuit outputs are initialised and *CIRCUIT OUTPUTS* window drawn.

If *FullInit=false* then the unit is re-initialised. The linked list of circuit outputs are deleted and the contents of the *CIRCUIT OUTPUTS* cleared.

PageOutputs procedure

Declaration — Procedure PageOutputs (down: boolean);

Function — When *down=true* then the contents of the *CIRCUIT OUTPUTS* window is moved down, one 'page' at a time, until the last circuit output of the linked list is displayed on the first line of the window. If *down=false* then the contents of the window are moved up.

OutputVal function

Declaration Function OutputVal (NoOfOutput: integer): real;

Function Returns the present value (a real number) of circuit output number *NoOfOutput* (the *NoOfOutput*'th record in the linked linked list).

SetOutput procedure

Declaration Procedure SetOutput (NoOfOutput: integer; VAR NewVal: real);

Function Assign *NewVal* to the value circuit output number *NoOfOutput*.

DefOutput procedure

Declaration

```
Procedure DefOutput (VAR UserVar: real;
                         Name   : string;
                         ValInit: real)
```

Function This routine is similar to the *DefParameter* procedure (declared in the *Pars* unit). *UserVar* is a real variable declared in a Cct<x> unit. *Name* is the circuit output string identifier associated with UserVar and will be displayed on the left-hand side, inside the *CIRCUIT OUTPUTS* window. *ValInit* is the initial value of the circuit output. The basic function of this routine is to create a record for the circuit output and to add it to the end of the linked list of circuit output records. After this routine has been invoked for a circuit output it can be viewed/modified by the user during run-time (by means of the *CIRCUIT OUTPUTS* menu and the *CIRCUIT OUTPUTS* window).

See also DefParameter procedure (Pars unit)

OutputNum function

Declaration Function OutputNum (OutputTitle: string): integer;

Function Searches all records in the linked list of circuit outputs, and returns the order of the record for which *OutputTitle* matches a circuit output identifier string (i.e. the *OutputName* field of the *OutputRec* record). If no match is found a *0* is returned.

DelOutputsList procedure

Declaration	Procedure DelOutputsList;

Function	Release the memory used for storing the linked list of circuit output records.

EraseOutputWindow procedure

Declaration	Procedure EraseOutputWindow;

Function	Clear the contents of the *CIRCUIT OUTPUTS* window.

UpdateOutputs procedure

Declaration	Procedure UpdateOutputs;

Function	Update the value of all circuit outputs displayed in the *CIRCUIT OUTPUTS* window.

GetOutputPtr function

Declaration	Function GetOutputPtr(NoOfOutput: integer): OutputPtrType;

Function	Returns a pointer pointing to the *NoOfOutput*'th record of the linked list of circuit output records.

WriteNewOutputVal procedure

Declaration	Procedure WriteNewOutputVal(PtrToOutput: OutputPtrType);

Function	If the value of the circuit output is legal (*CctResult=0*), then the displayed value (in the *CIRCUIT OUTPUTS* window) of the circuit output record pointed to by *PtrToOutput* is updated. If the value is illegal the value field is deleted for all outputs displayed in the *CIRCUIT OUTPUTS* window.

See also	Unit MmiGlobals for a description of the CctResult variable.

OutputYOfs function

Declaration — Function OutputYOfs(ptr: OutputPtrType): integer;

Function — Calculates the difference in the order of the circuit output pointed to by *ptr* and the output displayed at the top of the *CIRCUIT OUTPUTS* menu.

OutputNo function

Declaration — Function OutputNo(PtrToOutput: OutputPtrType): integer;

Function — Returns the order of the circuit output record pointed to by *PtrToOutput*. Returns a value of 0 for the first record, a value of 1 for the second, etc..

3.10 The Menus unit

This unit implements a menu facility. As the coupling between the main unit and this unit is quite tight this module was the most difficult to implement. In fact, the menu facility was incorporated in the main module in an early version of Circuit Tutor. This unit is not general enough to be used by any other program without prior modification, but could be adapted quite easily. If the menu facility was incorporated into the main unit this would not be possible.

Another reason for splitting the menu facility and the main unit is to facilitate easier software maintenance. If a bug is discovered that relates to the menu or its data structure the first step would be to find the fault in the *Menus* unit.

The data abstraction for the menu is accomplished by the two exported types, *Menu* and *MenuItem*, which should be treated as opaque/private types.

The main unit must first initialise a main (or root) menu using the *InitMenuModule* procedure. Thereafter items can be added to the main menu using the *AddMenuItem* procedure. The *AddMenu* procedure will associate a menu with the last item defined for the parent menu, and must be called immediately after *AddMenuItem* for a *Router* type menu item. The order of *AddMenuItem* and *AddMenu* procedure calls should therefore reflect the menu structure desired. To see how the menu structure is set up for Circuit Tutor please refer to the *InitMenuStructure* procedure in the main (program) unit.

The physical lay-out of the displayed menu is given in the following figure. The menu item identifier and user fields are also referred to as fields 1 and 2 of the menu item.

```
<menu number> <menu title>
-----------------------------------------------------
<menu item no.1 identifier field> [: <user field>]
<menu item no.2 identifier field> [: <user field>]
...
```

The structure of the *Menus* unit is given in figure 3-11.

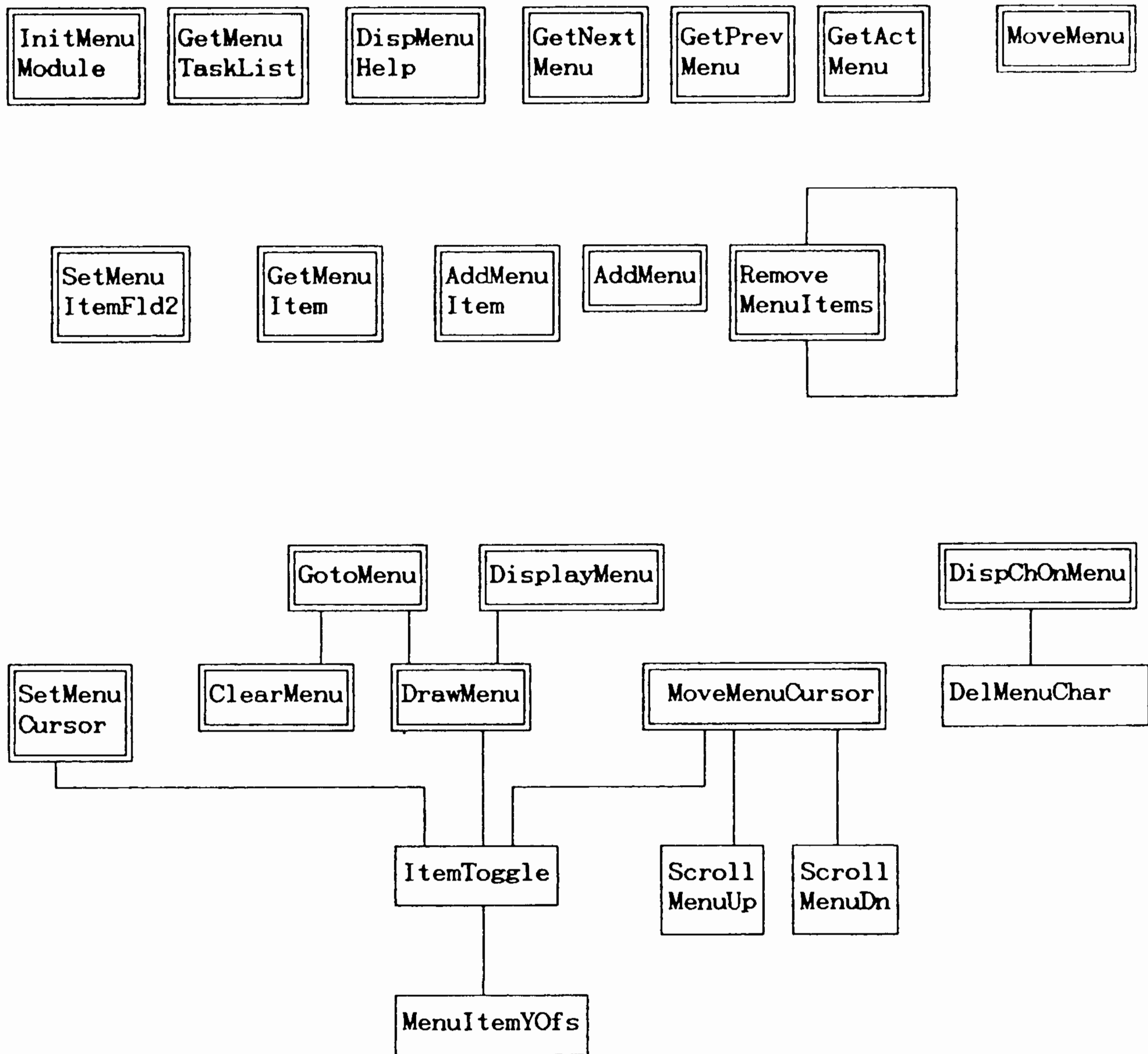


Figure 3-11. Structure of the Menus unit.

MenuItemCat type

Declaration Type MenuItemCat = (Router,Task,Field,Toggle,Info,NullItem);

Purpose When a menu item is selected by the user certain actions are required depending on the category of the menu item selected:

- Router. The next menu (child menu) must be displayed. The user fields for the menu items of the next menu must be updated before displaying the next menu. The *GetMenuTaskList* procedure must be used to obtain the list of task numbers to update the user fields. The *SetMenuItemFld2* procedure must be used for updating all user fields of the child menu. The following procedure will update all the user fields for menu *M*.

```
Procedure UpdateMenu(M:Menu);
Var
  Task:TaskList; i,MaxTasks:byte; UserFld:string;
begin
  GetMenuTaskList(M,Task,MaxTasks);
  For i:=1 to MaxTasks do
  begin
    ExecuteTask(Task[i,2],Task[i,3],true,UserFld);
    SetMenuItemFld2(M,Task[i,1],UserFld)
  end
end{UpdateMenu};
```

- **Task**. The task associated with task number 1 must be executed. The *GetMenuItem* procedure must be used to obtain task number 1.
- **Field**. The first time the user selects this field it signifies that future characters must be sent to the menu by means of the *DispChOnMenu* procedure. The next time the user keys in an *Esc* or *Enter* character the GetMenuItem procedure must be used to obtain Task1 and the user field string. Thereafter Task1 must be executed and the user field updated.
- **Toggle**. Similar to Field. The only difference is that the user does not have to type in a new field value. When the user selects this field the next value of the toggle (displayed in the user field) will be determined (with Task1) and displayed.
- **Info**. The user field for this item can not be modified by the user. The user field must be updated before the menu is displayed.
- **NullItem**. The item is invalid/spare, therefore the data returned by the *GetMenuItem* procedure is invalid.

Declaration

```
Type
  Menu = ^MenuRec;
  MenuRec = Record
    Title     : String;
    ItemSel   : MenuItem;
    FirstItem: MenuItem;
    LastItem  : MenuItem;
    TopItem   : MenuItem;
    PrevMenu  : Menu;
    XPos      : byte;
    YPos      : byte;
    WHeight   : byte;
    WWidth    : byte;
    ChkSt     : string[10];
  End;
```

Purpose

Defines the data structure of menu records. A description of record objects follow:

- **Title**. The menu title i.e. the name displayed in the menu header.
- **ItemSel**. Points to the selected menu item.
- **FirstItem**. Points to the first record in the linked list of menu items.
- **LastItem**. Points to the last record in the linked list of menu items.
- **TopItem**. Points to the menu item that is currently displayed on the first line of the menu. Normally *TopItem* will point to the first item, but when the menu contents are scrolled, this will point to some other menu item record.
- **PrevMenu**. The menu to be displayed when the user requests to go up one menu level (i.e. when the user pressess *Esc*).
- **XPos,YPos**. The co-ordinates (text co-ordinate system) of the top left corner of the menu.
- **WHeight**. The height of the menu (text co-ordinates).
- **WWidth**. The width of the menu (text co-ordinates).
- **ChkSt**. This string is used by some of the routines to check whether the menu has been initialised.

MENU RECORD | MENU ITEM RECORDS | MENU RECORD

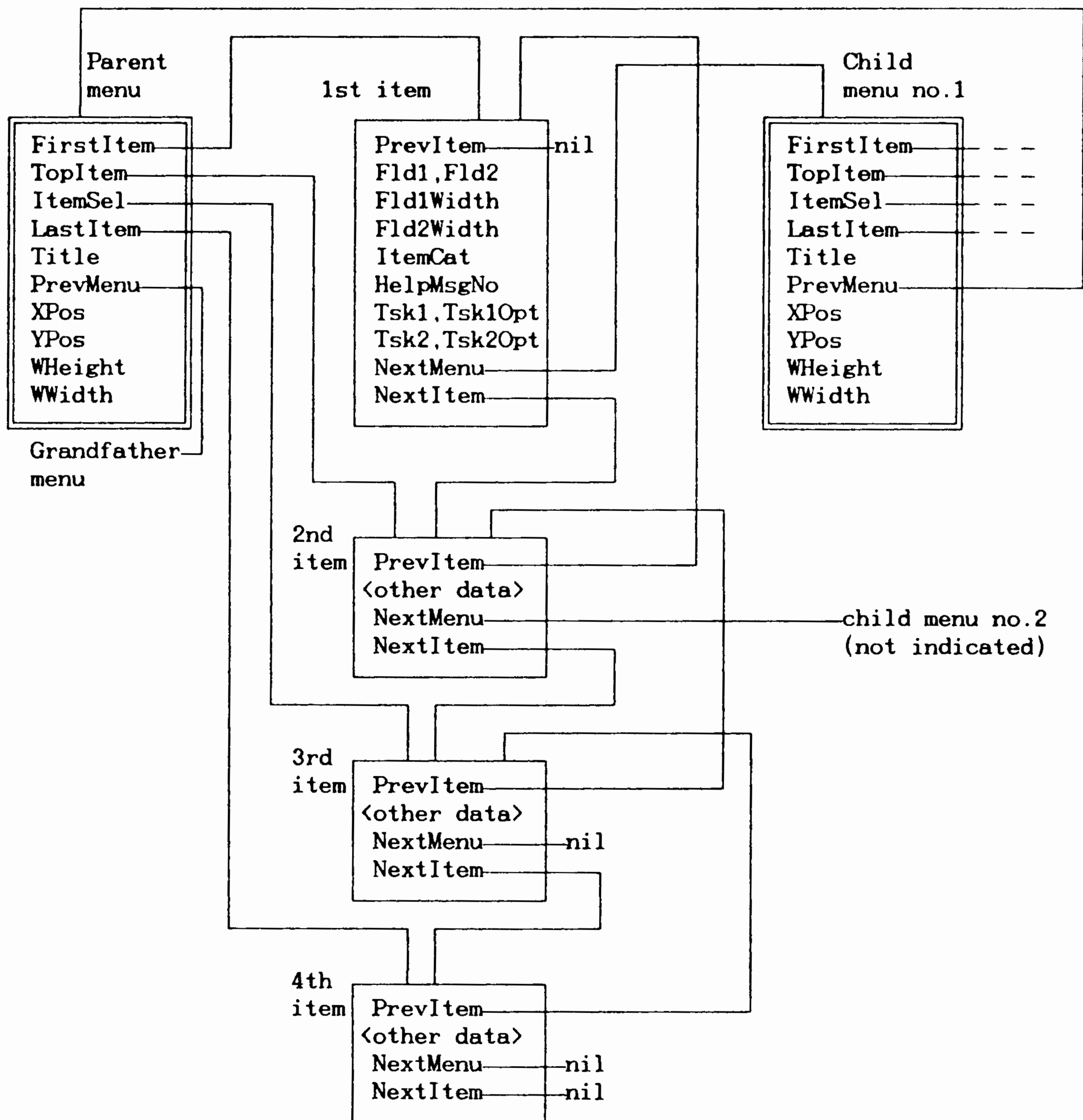


Figure 3-12. Typical menu data structure.

MenuItem type

Declaration

```
Type
  MenuItem    = ^ItemRecord;
  ItemRecord  =
  record
    ItemCat   : MenuItemCat;
    PrevItem  : MenuItem;
    NextItem  : MenuItem;
    Fld1,Fld2 : String;
    Fld1Width,
    Fld2Width : byte;
    Tsk1,Tsk2,
    Tsk1Opt,
    Tsk2Opt  : integer;
    HelpMsgNo: byte;
    NextMenu : Menu;
  end;
```

Purpose

Defines the data structure for menu items. A linked list of menu item records can be associated with each menu. Figure 3-12 gives the data structure for a typical menu with four items. A description of each object follows:

- **ItemCat**. The item category.
- **PrevItem**. Points to the previous item in the linked list.
- **NextItem**. Points to the next item in the list.
- **Fld1**. The item identifier e.g. *solve circuit*.
- **Fld2**. The user field value displayed to the right of the field text. This field can be modified by the user if the item has a category of *Toggle* or *Task*.
- **Fld1Width**. Width of the item identifier field.
- **Fld2Width**. Maximum number of characters that the user may input for *Fld2* i.e. the field value width.
- **Tsk1**. Task to be executed when the user selects the item.
- **Tsk1Opt**. Specifies option for Tsk1.
- **Tsk2**. Task to get the user field value (field 2).

- **Tsk2Opt**. Specifies option for Tsk2.
- **HelpMsgNo**. Number of the help message associated with this item. Messages are contained in file *TUTOR.MSG*.
- **NextMenu**. Points to the next menu to be displayed after an item has been selected.

Direction type

Declaration

```
Type Direction = (Up,Down,Left,Right);
```

Purpose

The directions that the menu can be moved to.

TaskList type

Declaration Type TaskList = Array[1..100,1..3] of integer;

Purpose Defines the data type for invoking the *GetMenuTaskList* procedure.

See also *GetMenuTaskList* procedure.

SelCctMenu, OutputsMenu and ParsMenu variables

Declaration Var SelCctMenu,OutputsMenu,ParsMenu: Menu;

Purpose Points to the *SELECT CIRCUIT*, *OUTPUTS* and *PARAMETERS* menu records respectively.

InitMenuModule procedure

Declaration Procedure InitMenuModule(VAR RootMenu: Menu;
RootMenuTitle: string;
XLeft,YUp,Width,Height: byte);

Function Initialise the menu data structure without drawing it. The linked list of records are initialised with *RootMenu* as the first record. The values of parameters *RootMenuTitle*, *XLeft*, *YUp*, *Width* and *Height* are assigned to the *Title*, *XPos*, *YPos*, *WWidth* and *WHeight* fields of *RootMenu*.

AddMenuItem procedure

Declaration Procedure AddMenuItem(ToMenu: Menu;
Cat: MenuItemCat;
Fld1Str: string;
Task1,Task1Opt,
Task2,Task2Opt: integer;
HelpMsg: byte);

Function Add a menu item to the linked list of menu items associated with menu *ToMenu*. The new item is linked to the existing last menu item and thereafter becomes the last menu item. The values of parameters *Cat*, *Fld1Str*, *Task1*, *Task1Opt*, *Task2*, *Task2Opt* and *HelpMsg* are assigned to the *ItemCat*, *Fld1*, *Tsk1*, *Tsk1Opt*, *Tsk2*, *Tsk2Opt* and *HelpMsgNo* fields of the newly created menu item record. If *Cat=Router* then the *AddMenu* procedure must be called after this procedure.

AddMenu procedure

Declaration Procedure AddMenu (VAR NewMenu: Menu;
ParentMenu: Menu;
MenuTitle: string;
XLeft,YUp,Width,Height: byte);

Function Associate a new menu record *NewMenu* with the last item of the linked list of menu items of menu *ParentMenu*. This procedure must be invoked after adding a *Router* type menu item. The values of parameters *RMenuTitle*, *XLeft*, *YUp*, *Width* and *Height* are assigned to the *Title*, *XPos*, *YPos*, *WWidth* and *WHeight* fields of *NewMenu*.

GetMenuTaskList procedure

Declaration	Procedure GetMenuTaskList(M:Menu; VAR TskLst:TaskList; VAR ListLength: byte);
Function	Returns a list of task numbers to be used for obtaining the new user fields (*Fld2*) of a menu *M*. *ListLength* is the length of the list and is not necessarily identical to the number of items for menu *M* as some menu items have no user fields. *TskLst* contains the following information: TskLst[x,1] returns a menu item number, e.g. *I*; TskLst[x,2] returns Tsk2 field for item number *I*; and TskLst[x,3] returns Tsk2Opt field for item number *I*.

DisplayMenuHelp procedure

Declaration	Procedure DisplayMenuHelp;
Function	Display the help message associated with the selected menu item.

GetNextMenu function

Declaration	Function GetNextMenu: Menu;
Function	Returns the menu associated with the selected menu item of category *Router*.

GetPrevMenu function

Declaration	Function GetPrevMenu: Menu;
Function	Get the menu one level up (with respect to the active menu).

GetActMenu function

Declaration	Function GetActMenu: Menu;
Function	Get the active menu.

SetMenuItemFld2 procedure

Declaration	Procedure SetMenuItemFld2(M:Menu; Item:byte; NewSt: String);
Function	Assign *NewSt* to *Fld2* of menu record ***M***.

GetMenuItem procedure

Declaration

```
Procedure GetMenuItem(VAR Fld2Str : string;
                      VAR Cat     : MenuItemCat;
                      VAR Task1   : integer;
                      VAR Task1Opt: integer;
                      VAR Task2   : integer;
                      VAR Task2Opt: integer;
                      VAR HelpMsg : byte);
```

Function

Get the detail of the selected menu item. The parameters *Fld2Str*, *Cat*, *Task1*, *Task1Opt*, *Task2*, *Task2Opt* and *HelpMsg* return the values of the selected item record fields *Fld2*, *ItemCat*, *Tsk1*, *Tsk1Opt*, *Tsk2*, *Tsk2Opt* and *HelpMsgNo* respectively.

RemoveMenuItems procedure

Declaration	Procedure RemoveMenuItems(M: Menu);
Function	Releases the heap memory used for storing the items of menu ***M***. If an item points to another menu that menu and its items will also be disposed i.e. the process disposes of all dynamic memory associated with the menu items of menu ***M***. It is for example, possible to dispose of the entire menu structure by invoking *RemoveMenuItems(MainMenu)*.

MoveMenu procedure

Declaration	Procedure MoveMenu(dir: direction);
Function	Move the displayed menu in the direction *dir*. The background will be preserved. The horizontal step size is 72 pixels (to line HGC text up properly) and the vertical step size 14 pixels.

GotoMenu procedure

Declaration Procedure GotoMenu(NewMenu: Menu);

Function Clears the displayed menu and draws the new menu *NewMenu*. *NewMenu* becomes the active menu.

ClearMenu procedure

Declaration Procedure ClearMenu;

Function The displayed menu is cleared. The DisplayMenu function can be used to restore the active menu.

DisplayMenu procedure

Declaration Procedure DisplayMenu;

Function Draws the active menu.

DispChOnMenu procedure

Declaration Procedure DispChOnMenu(Ch: Char);

Function Adds *Ch* to the user field (*Fld2* string) of the selected menu item and echoes the character on the right-hand side of the selected menu item. If the number of keyboard characters exceed the maximum length (*Fld2Width*) specified for the menu field value the character is ignored and a beep is emitted.

DelMenuChar procedure

Declaration Procedure DelMenuChar;

Function Deletes the last keyboard character added to the menu user field with the *DispChOnMenu* procedure.

MoveMenuCursor procedure

Declaration — Procedure MoveMenuCursor(dir: direction);

Function — Moves the menu cursor (of the displayed menu) in direction *dir* (up or down only). The contents of the menu are scrolled if necessary.

Pseudo-code

```
ALGORITHM MoveMenuCursor (dir: direction);
  set the item highlighting off;
  IF (dir is up) and (the selected menu item is not the first
  item)
  THEN
    select the menu item one up;
    IF (the menu item one up is the top menu item) THEN
      scroll the contents of the menu upwards;
    ENDIF;
  ENDIF;
  IF (dir is down) and (the selected menu item is not the last
  item)
  THEN
    select the next menu item;
    IF (the menu item is 'below' the menu) THEN
      scroll the contents of the menu downwards;
    ENDIF;
  ENDIF;
  set the item highlighting on;
END MoveMenuCursor;
```

ScrollMenuUp procedure

Declaration — Procedure ScrollMenuUp

Function — Scroll the contents of the active menu one line up.

ScrollMenuDn procedure

Declaration — Procedure ScrollMenuDn;

Function — Scroll the contents of the displayed menu one line down.

SetMenuCursor procedure

Declaration Procedure SetMenuCursor (state : boolean);

Function Set the menu cursor on/off for the user field portion of the item selected.

Pseudocode

```
ALGORITH SetMenuCursor (state: boolean);
  IF (state is true) THEN
    clear the user field (Fld2) of the selected menu item;
    calculate the initial x,y co-ordinates of the menu cursor;
    set menu item highlighting off;
    display the cursor;
    set menu item highlighting for the  menu item identifier
    field on (field 1 of the menu item)
  ELSE
    set menu item highlighting off for field 1;
    display the user field (Fld2) of the selected menu item;
    set menu item highlighting on for whole item
  ENDIF
END SetMenuCursor;
```

DrawMenu procedure

Declaration Procedure DrawMenu(M: menu);

Function Save the menu background and then draw menu *M*. Menu *M* is assigned as the active menu. The item selected will be highlighted.

ItemToggle procedure

Declaration Procedure ItemToggle(WideField: boolean);

Function Highlights the selected menu item for the active menu. If *WideField* is *true* then highlighting for the entire menu item (item identifier and user fields) will be toggled. If not, highlighting will be toggled for the item identifier field only.

MenuItemYOfs function

Declaration Function MenuItemYOfs: byte;

Function MenuItemYOfs =
order(selected menu item) - order(top menu item)

where order() returns the number of the menu item record e.g. the first menu item in the linked list has an order of 1.

3.11 MMIGlobals unit

This unit contains data objects global to all other Circuit Tutor units. In addition a routine to get the state of Circuit Tutor is provided.

3.11.1 Miscellaneous constants

- Version. The current version number of Circuit Tutor. When Circuit Tutor is modified this number must be incremented (adding a circuit is not considered to be a modification). The current version number is *1.04/hgc*, where *hgc* indicates that the program requires a Hercules Graphics Card. A minor modification will result in a new version number *1.05/hgc*. A major modification will result in a new version number *2.00/hgc*. If the current version is adapted for example for the Enhanced Graphics Adaptor, then *Version* will be *1.04/ega*.

- MaxNoOfCcts. The number of circuits that have been installed for Circuit Tutor. When a new circuit is added *MaxNoOfCcts* must be incremented.

- MaxNoOfErrorMsgs. Message number 1 to 10 of Circuit Tutor message file *TUTOR.MSG* are used for outputting error messages when the circuit outputs can not be solved. *MaxNoOfErrorMsgs* must be set to the number of the last message installed e.g. if message number 3 to 8 have not been used then *MaxNoOfErrorMsgs* must be set to 2.

- Task<TaskDescription>. Task number identifiers. For more information refer to documentation for the ExecuteTask procedure in the main (program) unit.

3.11.2 Graphics window constants

The Turbo Graphix toolbox permits the definition of a number of screen windows. To avoid confusion and to permit program readability each defined window has been given a constant identifier. The following window identifiers are contained in MMIGlobals:

- MenuW. Menu window.
- HighlightW. Menu item highlighting.
- MiscWindow. Miscellaneous window.
- CctWindow. Circuit diagram window.
- Meter<m>Window. Window for meter <m>, where m=1,2 or 3.
- GraphWindow. Graph window.
- Meter<m>NeedleW. Window for meter 'needles'.
- CctCursorW. Circuit diagram symbol highlighting.
- ParWindow. CIRCUIT PARAMETERS window.
- ParCursW. CIRCUIT PARAMETERS window item underliner.
- OutputWindow. CIRCUIT OUTPUTS window.
- Header8x8W. Window used by the DrawHeader8x8 procedure.
- MsgInnerWindow, MsgOuterWindow. Message windows.

3.11.3 Graphics co-ordinate system constants

Turbo Graphix screen co-ordinates systems, are similarly grouped together in this unit.

- **CctCoord**. Circuit diagram co-ordinate system, defining a grid of symbols.
- **Meter<m>Coord**. Co-ordinate system for meter<m> scales.
- **GraphCoord**. Co-ordinate system for graph.

3.11.4 Screen lay-out constants

The values of the following constants determine the screen lay-out:

- **MeterHeight**. Height of meters (screen co-ordinantes).
- **MeterWidth**. Width of meters (text co-ordinates).
- **ParWWidth**. CIRCUIT PARAMETER window width (text co-ordinates).
- **CctWLeftX**. X-position of the left-hand side of the circuit diagram (text co-ordinates).
- **CctFontHeight**. Height of circuit symbol (screen co-ordinates).

3.11.5 Variables

- **NewCctPars (boolean)**. If true then the circuit parameters have been modified.
- **CctResult (integer)**. This variable must be modified by procedures Cct1, Cct2, etc. If CctResult=0 then no errors were encountered whilst solving the circuit outputs. If an error occurs CctResult should be assigned the number of the message (in file TUTOR.MSG) that must be displayed.
- **ParsLoaded (boolean)**. Indicates that circuit parameters have been defined.
- **OutputsLoaded (boolean)**. Indicates that circuit outputs have been defined.
- **SolveCctOnce (boolean)**. If true, circuit outputs are not automatically solved after circuit parameters are modified.
- **CctDgmDrawn (boolean)**. If true the circuit diagram has been drawn.
- **StopProgram (boolean)**. If true the program will be halted.
- **CctSelected (CctNumber)**. Indicates the number of the circuit currently selected.

MMIStateType type

Declaration Type MMIstateType= (Default,TxtInput,OnCctMode);

Purpose The various states of the Man-Machine Interface (MMI) part of Circuit Tutor. After initialisation, Circuit Tutor is put in the *default* state. When the user selects a menu item of type *Field* (the user desires to modify the user field portion of the selected menu item), Circuit Tutor is put in the *TxtInput* state. When the user selects QUICK mode via the menu or by pressing *F3* (refer to the User's Reference Manual) Circuit Tutor is put in the *OnCctMode* state.

CctSolveOption type

Declaration Type CctSolveOption=(IdentifyCct, InitCct, SolveCct);

Purpose This type identifier is used by the routines to solve the circuit parameter-output relationship. For its useage please refer to chapter 4 of the User's Reference Manual.

CctNumber type

Declaration Type CctNumber = 1..MaxNoOfCcts;

Purpose The number of the circuit selected must be of this type.

XTxt type

Declaration Type XTxt = 1..80;

Purpose Legal X co-ordinates for HGC text cursor.

YTxt type

Declaration YTxt = 1..25;

Purpose Legal Y co-ordinates for HGC text cursor.

3.11.7 Exported function

MmiState function

Declaration	Function MMIstate: MMIstateType;
Function	Returns the current state of the program.

3.12 The Ascii unit

This small unit contains constant identifiers for most extended ASCII character constants e.g.

```
Const
  PgUp     = #73;
  PgDn     = #81;
  Home     = #71;
  EndKey   = #79;
  Enter    = #13;
  Esc      = #27;
  Null     = #00;
  AltF1    = #104;
  AltF2    = #105;
```

etc.

3.13 The Maths and CMaths units

These two units provide the necessary maths support for the units that solve the circuit outputs. Maths contain routines for real maths, whilst CMaths contain routines for complex maths. All the routines in Maths are also contained in CMaths, the only difference being that complex numbers instead of real numbers are used. For example the CMatrixAdd procedure in CMaths will add two complex matrices, whilst the MatrixAdd procedure in Maths will add two real matrices.

For the above reason only the CMaths unit will be described.

3.13.1 Description of exported data and routines

The interface section of the CMaths unit follows:

```
Const
  CMaxRows = 10;
  CMaxCols = 20;
Type
  Complex =
  record
    Re: real;
    Im: real;
  end;
  ComplexPtr= ^Complex;
  CBigMatrix = Array[1..CMaxRows,1..CMaxCols] of Complex;
  CVector    = Array[1..CMaxRows] of Complex;

Procedure CMatrixAdd
 (Matrix1,Matrix2: CBigMatrix;
  Var Result : CBigMatrix;  { Result=Matrix1+Matrix2 }
  rows, cols : byte);       { dimension of matrix 1,2 }

Procedure CMatrixSubtract
 (Matrix1,
  Matrix2    : CBigMatrix;
  Var Result : CBigMatrix;  { Result=Matrix1-Matrix2 }
  rows, cols : byte);       { dimension of matrix 1,2 }

Procedure CMatrixMultiply
 (Matrix1,
  Matrix2    : CBigMatrix;
  Var Result : CBigMatrix;  { Result=Matrix1*Matrix2 }
  rows1, cols1,             { dimension of matrix 1 }
  cols2      : byte);       { columns for matrix 2 }

Procedure CMatrixInvert
 (Y          : CBigMatrix;
  Var Result : CBigMatrix;  { Result=1/Y }
  rows       : byte;        { rows for Y }
  Var Error  : boolean);    { Result invalid if true }
```

```
Procedure CMatrixTranspose
 (Matrix    : CBigMatrix;
  Var Result: CBigMatrix; { Result=transpose(Matrix) }
  rows, cols: byte);      { dimension of Matrix }

Procedure CGaussJordan
 (Var Y      : CBigMatrix;
  rows, cols : byte;      { cols ≥ rows+1 }
  var Error  : boolean);  { elimination failed if true }

Procedure CAss
 (VAR A: Complex;
  RealPart, ImagPart: Real);                  { A.Re=RealPart; A.Im=ImagPart}
Function  CAbs (A: Complex): real;            { CAbs  = abs(A) }
Function  CAdd (A, B: Complex): ComplexPtr;   { CAdd^ = A+B }
Function  CSub (A, B: Complex): ComplexPtr;   { CSub^ = A-B }
Function  CDiv (A, B: Complex): ComplexPtr;   { CDiv^ = A/B }
Function  CMult(A, B: Complex): ComplexPtr;   { CMult^= A*B }
```

The CGaussJordan procedure will manipulate all the elements of all the rows whilst normalising the submatrix of dimension [*rows,rows*]. Matrix *Y* must be at least of dimension [*rows,rows*+1] for the procedure to return a valid value. If the elimination fails *Error* will be true. For example *CGaussJordan(Y,3,4,err)* will solve a set of linear equations in 3 variables. The fourth column of Y will contain the value of the three variables if no error occurred (err=false).
The usage of the other matrix procedures should be straightforward from the comments provided.

CSub, CAdd, CMult, etc. perform simple maths operations on complex numbers and return a pointer that points to a complex number instead of a complex number, as Pascal functions can not return record values.

3.14 Other units

The units to solve the circuit outputs (Cct1, Cct2, etc.) must be provided by the user. The User's Reference Manual contains detailed information regarding the creation of these units. The following units are included for demonstration purposes:

- **Cct1 unit: simple transistor circuit.** The transistor model is that of Ebers-Moll [Gray,1969]. The non-linear matrix equation for the circuit is solved using the Newton-Raphson method [Fidler, 1978].

- **Cct2 unit: capacitor charging circuit.**

- **Cct3 unit: resistor network.** The matrix equation for the circuit is solved using the GaussJordan procedure from the Maths unit.

- **Cct4 unit: Chebyshev low-pass filter (6th order).** The matrix equation (which contains complex numbers) for the circuit is solved using the CGaussJordan procedure from the CMaths unit.

The following units are used by Circuit Tutor but were not developed as part of the project.

- **HGC unit.** This unit was created by grouping together all the Turbo Graphix Toolbox (version 4) routines contained in the three units GWindow, GKernel and GDriver.

- **Printer, Dos, and Crt units.** These units are distributed with the Turbo Pascal (version 4) compiler.

4

DESIGN OF THE CIRCUIT DRAW UTILITY PROGRAM

4.1 General Goals

Installing a circuit for Circuit Tutor entails:

- writing the Pascal unit to solve the circuit; and
- drawing the circuit diagram for the circuit.

The main goal of Circuit Draw is to provide an efficient means of creating and/or editing circuit diagram files. Using Circuit Draw it should be possible to draw an average circuit diagram in a matter of a few minutes.

4.2 Specifications for Circuit Draw

1. The Circuit Draw VDU lay-out should resemble figure 4-1.

2. A menu of available choices should be displayed to user.

3. The user should be able to:

   - load a circuit diagram file from disk;
   - select a symbol from a table of symbols (symbol table) and transfer it to the circuit diagram. The user should be able to select whether the transferred symbol should overwrite the contents of the diagram; or whether it should be combined;
   - add text to the circuit diagram;
   - print the displayed circuit diagram to an Epson or compatible printer;
   - erase the displayed diagram;
   - rotate the selected symbol for the symbol table;
   - interactively modify any symbol of the symbol table;
   - create a new symbol and add it to the symbol table;
   - save the symbol table to file;
   - load the symbol table from disk; and
   - save the displayed circuit diagram to disk.

4. The program should run on an IBM PC or compatible micro-computer with a Hercules Graphics Card (HGC) fitted.

4.3 Design of the VDU lay-out

Figures 4-1, 4-2 and 4-3 give the lay-out of the Circuit Draw display. The Circuit Draw contains four windows:

- **Menu window.** This window contains all the choices available for the mode selected.

- **Circuit diagram window.** The circuit diagram may contain text and/or symbols. A maximum of 136 (17H x 8V) symbols can be drawn, which is adequate for the type of circuits to be demonstrated by Circuit Tutor. Characters drawn on the circuit diagram in DRAW DIAGRAM TEXT mode comprise of an 8 by 8 pixel array.

- **Symbol table window.** The symbol table contains 48 symbols which is adequate for drawing most simple circuits. Circuit symbols comprise of a 32 (horizontal) by 23 pixel array, thereby making it possible to define quite complex symbols.

- **Status/dialogue line.** Displays the status of Circuit Draw (e.g. draw diagram symbols mode) and indicates whether symbol/text OVERWRITE is on. When the user has to key in additional information (to a file name or confirmation request from Circuit Draw) the status line becomes the dialogue line.

With the exception of the characters displayed on the circuit diagram, Hercules text has been used.

4.4 Design of the program

As with Circuit Tutor the first step is to identify the main units. This will similarly be accomplished by developing an informal algorithm for the main module (program unit).

```
ALGORITHM Draw;
  initialise all modules and clear TerminateProgram flag;
  LOOP
    IF (TerminateProgram flag set) THEN
      halt program
    ELSIF (a keyboard character has been pressed) THEN
      read the keyboard character;
      IF (the keyboard character is an <Esc> character) THEN
        ask the user for confirmation and set the TerminateProgram flag
        accordingly
      ELSE
        ProcessKbdInput  (* process the keyboard character *)
      ENDIF;
    ELSIF (Draw is in 'define symbols' mode) THEN
      flash the symbol grid cursor
    ENDIF;
  ENDLOOP;
END Draw.
```

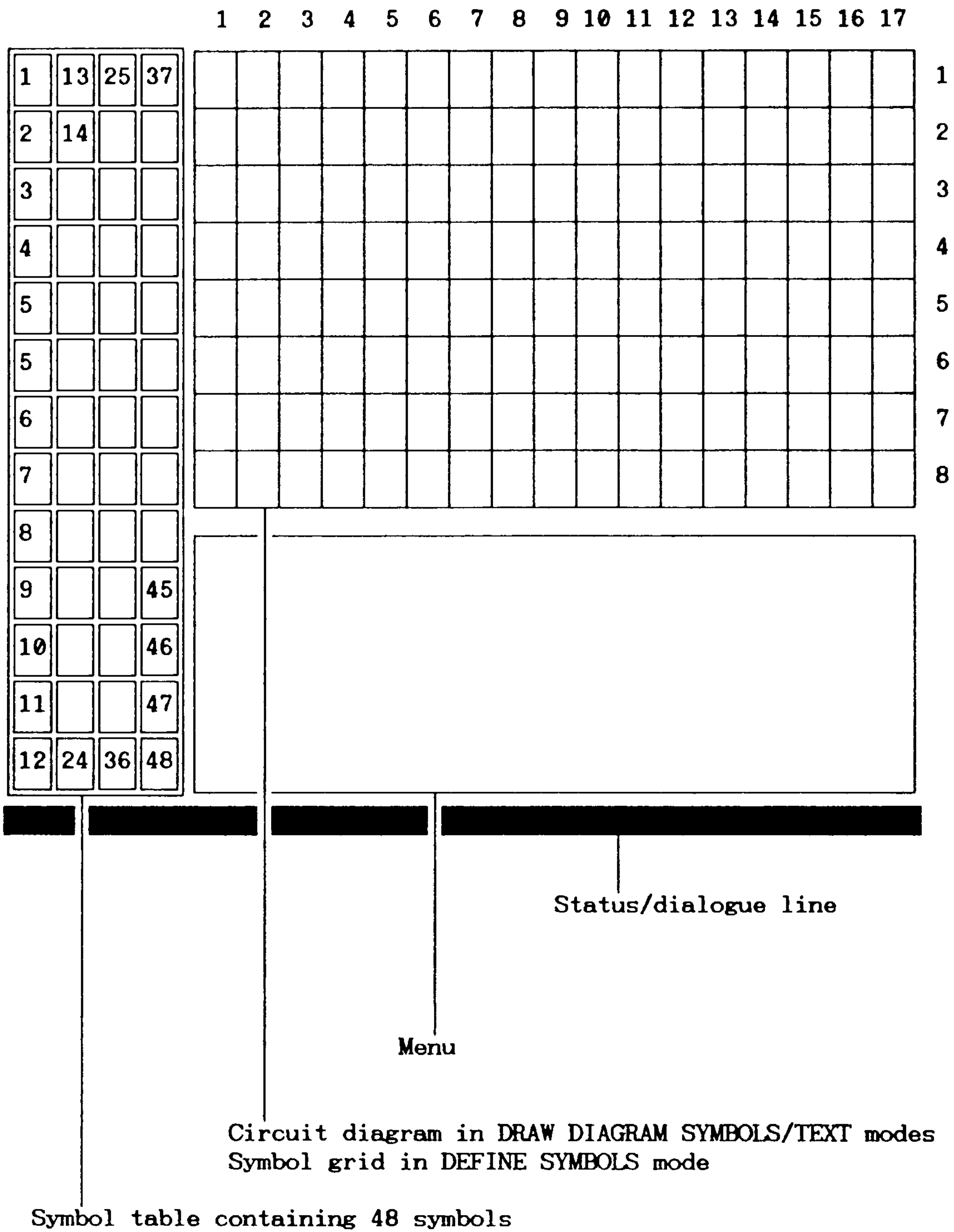


Figure 4-1. Lay-out of the Circuit Draw display.

```
ALGORITH ProcessKbdInput;

 IF (Draw is 'draw diagram symbols' mode) THEN
   CASE (keyboard character) OF
     Home     : move symbol table cursor up
   ! EndKey   : move symbol table cursor down
   ! PgUp     : move symbol table cursor left
   ! PgDn     : move symbol table cursor right
   ! UpArr    : move diagram cursor up
   ! DownArr  : move diagram cursor down
   ! LeftArr  : move diagram cursor left
   ! RightArr: move diagram cursor right
   ! F1       : save diagram to file
   ! F2       : load diagram from file
   ! F4,Ins   : toggle overwrite switch
   ! F5       : ChangeStatus(DrawDgmText)     (* put Draw in 'draw diagram text
                                                 mode *)
   ! F6       : print diagram
   ! F7       : ChangeStatus(DefineSymbols) (* put Draw in 'define symbols'
                                                 mode *)
   ! F8       : flip symbol vertically
   ! F9       : flip symbol horizontally
   ! F10      : erase diagram
   ! Enter    : draw a symbol at diagram cursor position
   ENDCASE;
 ELSIF (Draw is in 'draw diagram text' mode) THEN
   CASE (keyboard character) OF
     UpArr    : move text cursor up
   ! DownArr  : move text cursor down
   ! LeftArr  : move text cursor left
   ! RightArr: move text cursor right
   ! F1       : save diagram to file
   ! F2       : load diagram from file
   ! F3       : ChangeStatus(DrawDgmSymbols)  (* put Draw in 'draw diagram
                                                   symbols' mode *)
   ! F4,Ins   : toggle overwrite switch
   ! F6       : print diagram
   ! F7       : ChangeStatus(DefineSymbols)   (* put Draw in 'define symbols'
                                                   mode *)
   ! F10      : erase diagram contents
   ELSE
     write the character on the diagram at the text cursor position
   ENDCASE;
 ELSIF (Draw is in 'define symbols' mode) THEN
   CASE (keyboard character) OF
     Home     : move symbol table cursor up
   ! EndKey   : move symbol table cursor down
   ! PgUp     : move symbol table cursor left
   ! PgDn     : move symbol table cursor right
   ! UpArr    : move symbol grid cursor up
   ! DownArr  : move symbol grid cursor down
   ! LeftArr  : move symbol grid cursor left
   ! RightArr: move symbol grid cursor right
   ! F1       : save symbol table to disk
   ! F2       : load symbol table from disk
```

```
    | F3       : ChangeStatus(DrawDgmSymbols)  (* put Draw in 'draw diagram
                                                  symbols' mode *)
    | F4       : transfer the symbol from the symbol grid to the symbol table
                                               (* draw symbol on symbol table *)
    | F5       : ChangeStatus(DrawDgmText)     (* put Draw in 'draw diagram
                                                  text' mode *)
    | F6       : transfer the symbol defined by the symbol grid to the symbol
                 table
    | F8       : flip the selected symbol table symbol vertically
    | F9       : flip the selected symbol table symbol horizontally
    | F10      : erase the contents of the symbol grid
    | Enter    : invert symbol grid pixel
    ENDCASE;
  ENDIF;
END ProcessKbdInput;


ALGORITHM ChangeStatus(NewState : ProgramStatus);
  CASE NewState OF
    DrawDgmSymbols:
      SetOverwriteLbl(false);
      IF (previous state was 'define symbols') THEN
        set symbol grid cursor off;
        store symbol grid to RAM;
        restore diagram from RAM
      ELSIF (previous state was 'draw diagram text') THEN
        set text cursor off
      ENDIF;
      update status line;
      draw the menu for 'draw diagram symbols' mode;
      set the diagram (symbol) cursor on
  | DrawDgmText:
      IF (previous state was 'define symbols') THEN
        set symbol grid cursor off;
        store symbol grid to RAM;
        restore diagram from RAM
      ELSIF (previous state was 'draw diagram symbols' THEN
        set diagram (symbol) cursor off
      ENDIF;
      update status line;
      draw the menu for 'draw diagram text' mode;
      set diagram text cursor on
  | DefineSymbols:
      set diagram text/symbol cursor off;
      store diagram to RAM;
      restore symbol grid from RAM;
      set symbol grid cursor on;
      update status line;
      draw the menu for 'define symbols' mode
  ENDCASE;
END ChangeStatus;
```

4.4.1 Preparing a modular design chart

From the algorithm developed in the preceding section it is now possible to identify the main objects and associated functions.

object	functions
symbol table (SymTable unit)	• move symbol table cursor • flip symbol table symbol horizontally/vertically • save/load symbol table to/from disk file • draw symbol at symbol table cursor position
symbol grid (grid unit)	• store/restore grid to/from RAM • move grid cursor • set grid cursor on/off • flash grid cursor • erase grid contents • transfer symbol from symbol table to grid • transfer symbol form grid to symbol table
diagram (diagram unit)	• save/load diagram to/from disk file • store/restore diagram to/from RAM • print diagram • move diagram (symbol) cursor • set diagram (symbol) cursor on/off
text (DrawText unit)	• move text cursor • write a text character • set diagram (text) cursor on/off

5

THE CIRCUIT DRAW UNITS IN DETAIL

5.1 The main (program) unit

The main program initialises all Circuit Draw units, draws the menu and thereafter processes the keyboard input from the user.

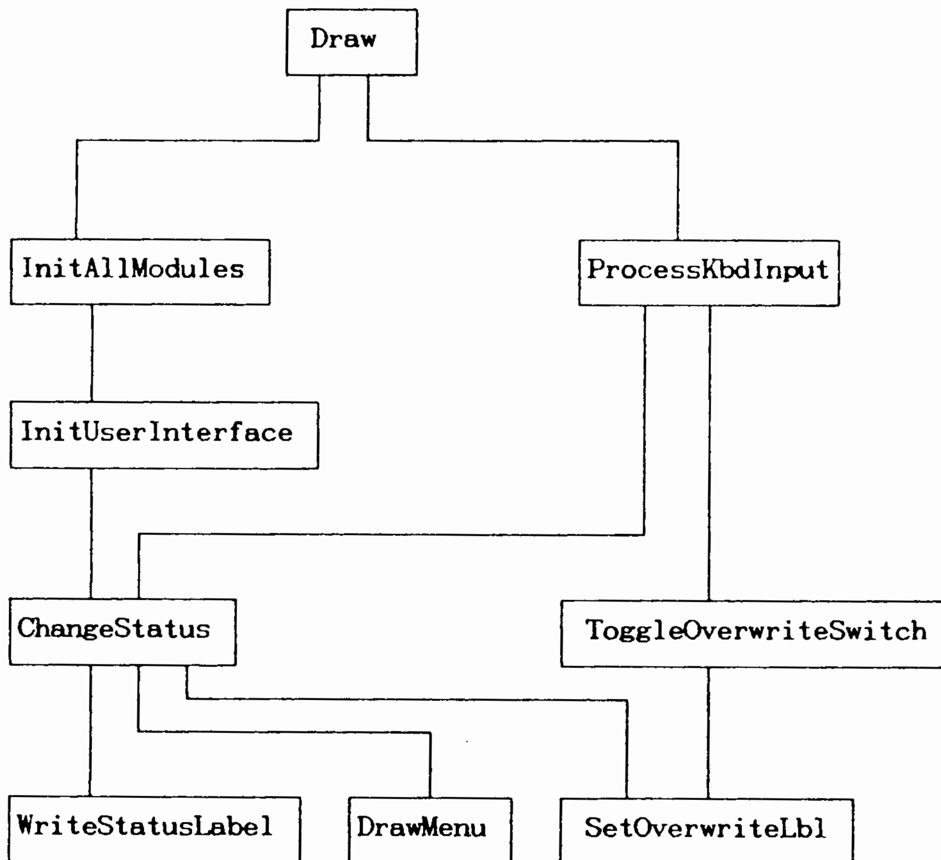


Figure 5-1. Draw program structure.

InitAllModules procedure

Declaration Procedure InitAllModules;

Function Initialises Circuit Draw. If essential files (14x9.fon, 8x8.fon and error.msg) are not found in the default directory the program will be halted.

The *DrawDgmSymbols* program state will be selected.

InitUserInterface procedure

Declaration Procedure InitUserInterface;

Function Initialises the user interface. Three Turbo Graphix Toolbox windows are defined, one for the menu of each mode (state) of Circuit Draw if sufficient memory is available - if not, menus are drawn each time the mode is changed.

The default state of the program is *DrawDgmSymbols*.

ChangeStatus procedure

Declaration Procedure ChangeStatus(NewState: ProgramStatus);

Function Changes between the various states of Circuit Draw. After initialisation these states can be selected:

- *DrawDgmSymbols*. In this state the user can draw circuit symbols. This state is indicated to the user as DRAW DIAGRAM SYMBOLS mode.
- *DrawDgmText*. In this state the user can draw text. This state is displayed to the user as DRAW DIAGRAM TEXT mode.
- *DefineSymbols*. In this state the user can create/edit symbols. This state is identified as DEFINE SYMBOLS mode to the user.

The screen display, excluding the symbol table, is modified when a new state is entered. If *DefineSymbols* is selected the circuit diagram window is replaced with the symbol grid.

WriteStatusLabel procedure

Declaration   Procedure WriteStatusLabel(StatusStr: String40);

Function   After initialisation Circuit Draw can be one of three states as described for the ChangeStatus procedure. This procedure will write a label string *StatusStr* underneath the menu to display the current state to the user.

DrawMenu procedure

Declaration   Procedure DrawMenu(Menu: MenuArr);

Function   Draw a menu comprising 14 fields (items). The *MenuArr* is declared in this unit as:

```
MenuArr = Array[1..14] of string[45];
```

SetOverwriteLbl procedure

Declaration   Procedure SetOverwriteLbl(ON: boolean);

Function   When the user draws symbols or text it is possible to overwrite the existing information on the circuit diagram, if overwrite is on. This procedure will set overwrite on if *ON* is true.

When overwrite is active *OVERWRITE* is displayed under the menu.

ToggleOverwriteSwitch procedure

Declaration   Procedure ToggleOverwriteSwitch;

Function   Toggles overwrite mode ON/OFF independantly for *DrawDgmSymbols* and *DrawDgmText* states.

ProcessKbdInput procedure

Declaration	Procedure ProcessKbdInput;
Function	Process keyboard input. All the functions displayed on the menus are accessed by the keys indicated . The keyboard characters can therefore be translated to associated function/procedure calls. For instance, in *DrawDgmSymbols* (DRAW DIAGRAM SYMBOLS) state, the F1 key will be translated to a SaveDgmToFile call. Procedure SaveDgmToFile will prompt the user for the filename.

The Grid unit provides routines to manipulate the symbol grid, used for creating/editing symbols.

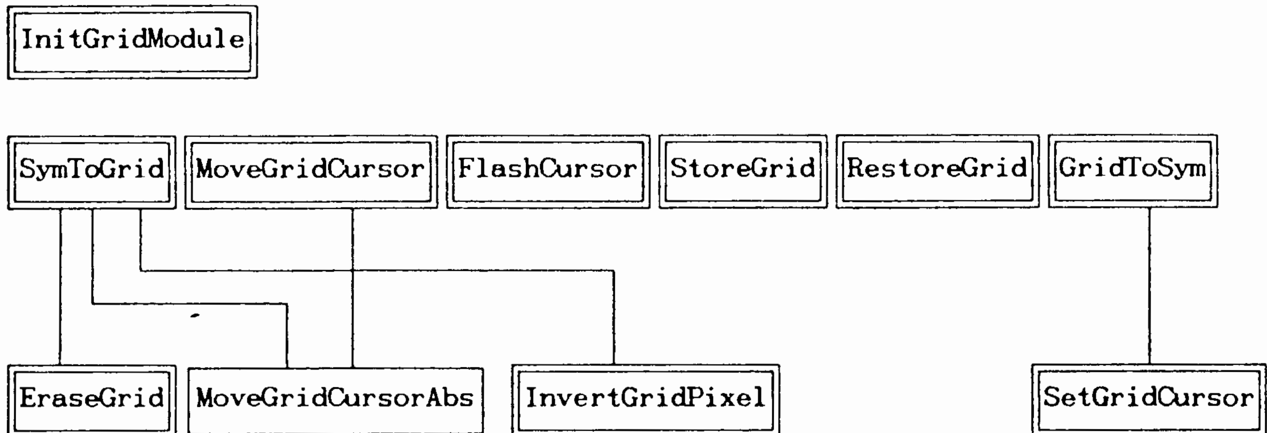


Figure 5-2(a). Grid unit structure.

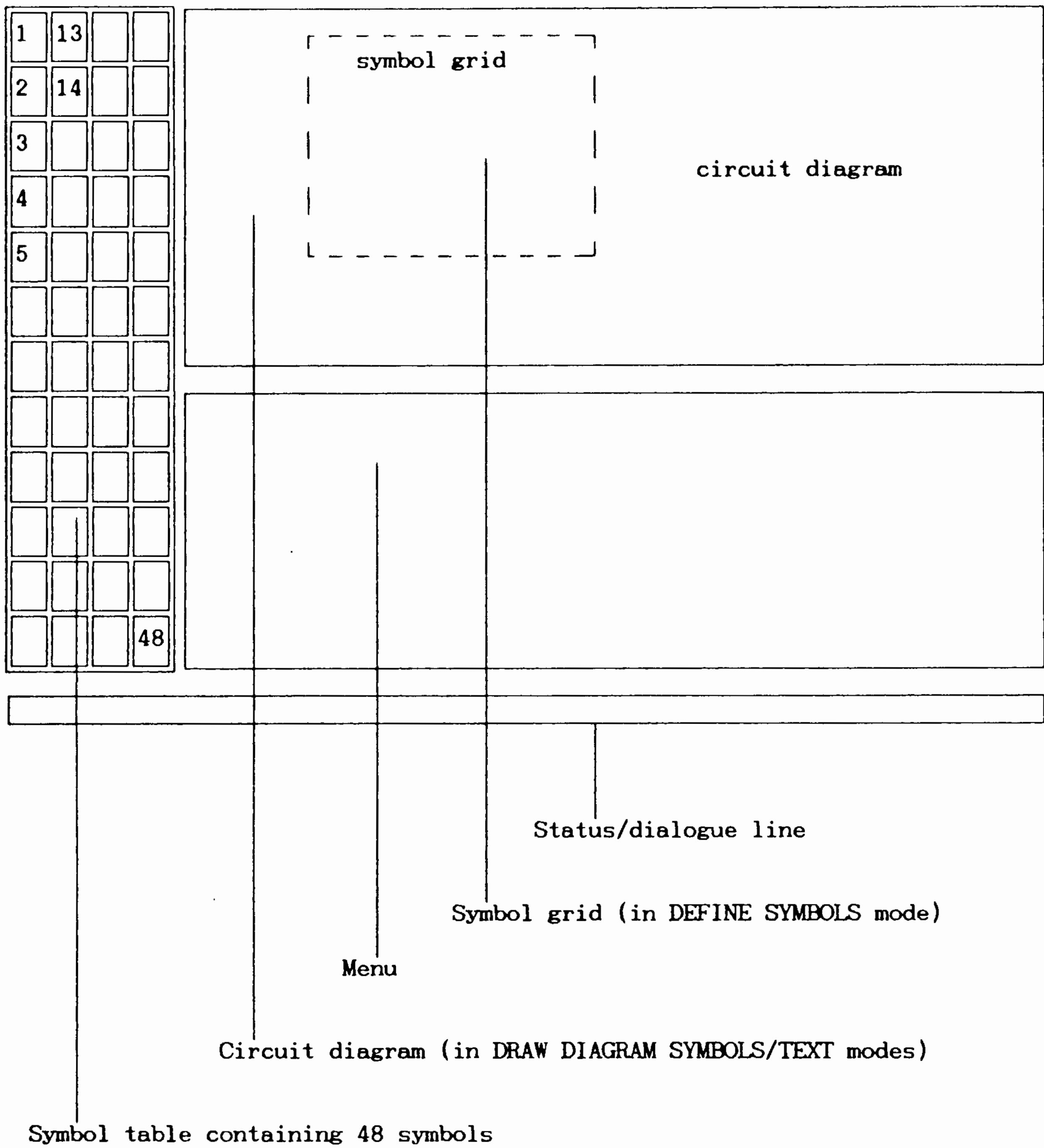


Figure 5-2(b). Lay-out of the Circuit Draw display.

The grid represents a 32H x 23V dot matrix array. When Circuit Draw is in *DefineSymbols* (DEFINE SYMBOLS mode), one of the dots on the grid will flash. This flashing dot, which is implemented by defining it as a Turbo Graphix window, is termed the grid cursor.

5.2.2 Description of routines

StoreGrid procedure

Declaration	Procedure StoreGrid;
Function	Store the Grid window to (dynamic allocated) memory.

RestoreGrid procedure

Declaration	Procedure RestoreGrid;
Function	Restore to Grid window from memory.

InvertGridPixel procedure

Declaration	Procedure InvertGridPixel;
Function	Inverts the grid pixel selected by the flashing grid cursor.

FlashCursor procedure

Declaration	Procedure FlashCursor;
Function	Set the grid cursor off and then on again, thereby flashing it.

SymToGrid procedure

Declaration	Procedure SymToGrid;
Function	Draw the symbol, selected in the symbol table, on the grid. The contents of the grid will be erased.

GridToSym procedure

Declaration Procedure GridToSym;

Function Transfers the contents of the grid to the selected symbol table symbol.

MoveGridCursor procedure

Declaration Procedure MoveGridCursor(dir: direction);

Function The grid cursor will be moved in direction *dir* (up, down, left, right).

MoveGridCursorAbs procedure

Declaration Procedure MoveGridCursorAbs;

Function Move the grid cursor to grid co-ordinate position [CursorX,CursorY], where CursorX and CursorY are variables declared in the implementation section of this unit.

SetGridCursor procedure

Declaration Procedure SetGridCursor(On:boolean);

Function If *On* is true, then the grid cursor is displayed.

EraseGrid procedure

Declaration Procedure EraseGrid;

Function The contents of the grid are erased. This is achieved by restoring the initial grid window.

InitGridModule procedure

Declaration Procedure InitGridModule;

Function Initialise the Grid unit. The grid is drawn and then saved to a Turbo Graphix window.

5.3 SymTable unit

The SymTable unit provides routines to manipulate the Symbol table.

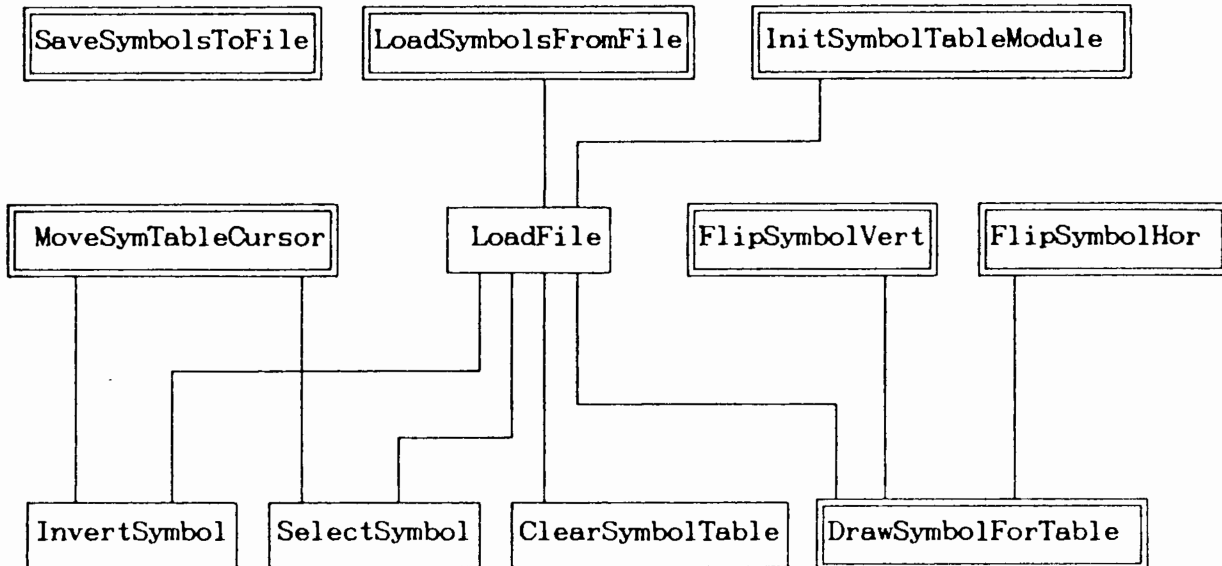


Figure 5-3. Structure of SymTable unit.

SaveSymbolsToFile procedure

Declaration Procedure SaveSymbolsToFile;

Function Prompts the user for a filename (including the pathname, but excluding the .sym extension). The symbol table - comprising of 48 symbols - is then saved to this file. The SymbolFile type is defined in this unit as :

```
SymbolFile    : File of SymbolArray;
```

where the SymbolArray type is defined in the DrawGlobals unit.

LoadSymbolsFromFile procedure

Declaration Procedure LoadSymbolsFromFile;

Function The user is prompted for a filename (including the pathname, but excluding the .sym extension). The entire symbol table - comprising of 48 symbols - will be loaded from the file specified. The existing symbol table will be overwritten.

LoadFile procedure

Declaration Procedure LoadFile(FileName: String80);

Function Checks whether the symbol file *FileName* exists. If the file exists then the symbol table is overwritten with the symbol table read from *FileName*. If the file does not exist an error message is displayed to the user.

DrawSymbolForTable procedure

Declaration Procedure DrawSymbolForTable;

Function Draws the symbol previously selected with the SelectSymbol procedure.

The Diagram unit exports routines to perform operations on the diagram and to draw diagram symbols.

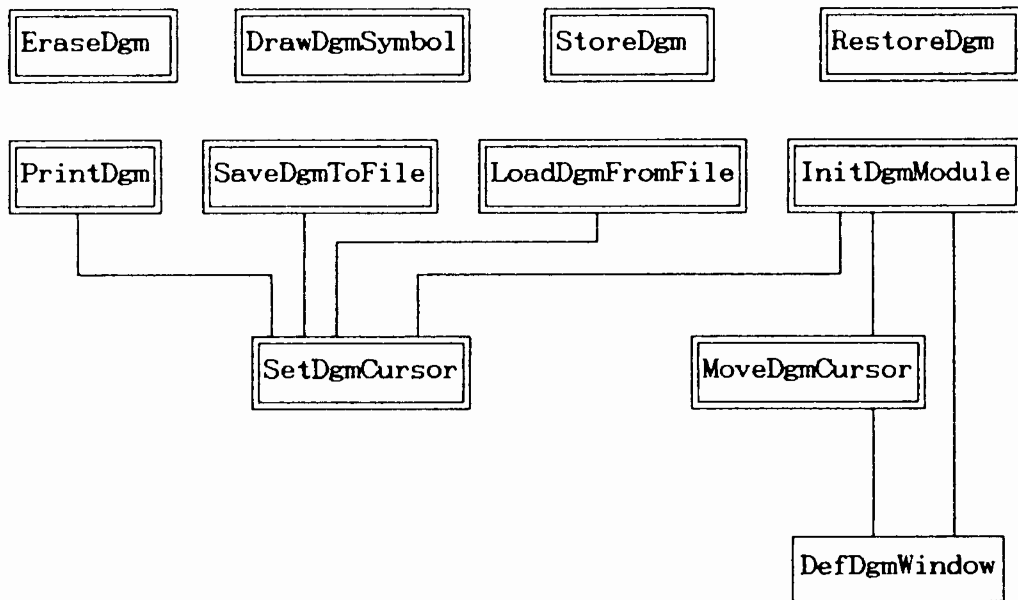


Figure 5-4. Structure of Diagram unit.

5.4.1 Description of exported variable

SymbolOverwrite variable

Declaration Var SymbolOverwrite: boolean;

Purpose When SymbolOverwrite=true then the DrawDgmSymbol procedure will clear the display memory 'under' the symbol. If false the display memory will be logically AND'ed (pixel for pixel) with the symbol.

5.4.2 Description of routines

EraseDgm procedure

Declaration Procedure EraseDgm;

Function Erase the contents of the diagram.

DrawDgmSymbol procedure

Declaration Procedure DrawDgmSymbol;

Function Draw the symbol currently selected (in the symbol table). The symbol comprises of a 32 (horizontal) by 23 (vertical) pixel array.

StoreDgm procedure

Declaration Procedure StoreDgm;

Function Store the diagram to (dynamically allocated) memory.

RestoreDgm procedure

Declaration Procedure RestoreDgm;

Function Restore the diagram from memory.

PrintDgm procedure

Declaration Procedure PrintDgm;

Function Prints the diagram to an Epson compatible printer (9-pin printers only).

SaveDgmToFile procedure

Declaration Procedure SaveDgmToFile;

Function Saves the diagram to a disk (or diskette) file. Prompts the user for a destination filename.

LoadDgmFromFile procedure

Declaration Procedure LoadDgmFromFile;

Function Load the diagram from disk (or diskette) file. Prompts the user for a source filename.

InitDgmModule procedure

Declaration Procedure InitDgmModule;

Function Initialise the Diagram unit.

SetDgmCursor procedure

Declaration Procedure SetDgmCursor(on:boolean);

Function Set the diagram cursor (i.e. the un-highlighted symbol) on or off. If on=true then the diagram cursor is displayed.

MoveDgmCursor procedure

Declaration Procedure MoveDgmCursor(dir: Direction);

Function Moves the diagram cursor up, down, left or right.

DefDgmWindow procedure

Declaration Procedure DefDgmWindow;

Function Defines the diagram window.

5.5 DrawText unit

This unit exports routines to draw text on the diagram.

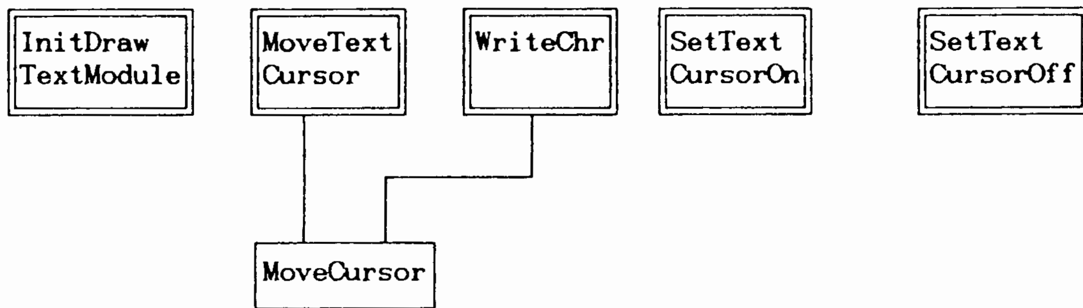


Figure 5-5. Structure of DrawText unit.

5.5.1 Description of exported constant

CctDgmX1 constant

Declaration	Const CctDgmX1 = 22;
Purpose	Defines the text co-ordinate of the lefthand corner of the diagram.

5.5.2 Description of routines

MoveTextCursor procedure

Declaration	Procedure MoveTextCursor(dir: direction);
Function	Moves the diagram text cursor one position relative to the existing position. The direction is specified by *dir*. If an attempt is made to move the text cursor beyond the specified limits of the diagram the cursor will not be moved and a beep will be emmitted. The co-ordinates of the top left and bottom right corners of the diagram are [DgmLeft,1] and [XMaxGlb, DgmBottom] respectively. *XMaxGlb* is exported by the HGC unit, whilst *DgmLeft* and *DgmBottom* are constants declared in the implementation part of the DrawText unit.

WriteChr procedure

Declaration	Procedure WriteChr(ch: char);
Function	Write a text character on the diagram. The text character *ch* will be displayed at the diagram text cursor position.
See also	MoveTextCursor procedure

SetTextCursorOn procedure

Declaration Procedure SetTextCursorOn;

Function Display the diagram text cursor. The diagram text cursor is a small normal video window on the diagram (a large inverse video window).

See also SetTextCursorOff procedure

SetTextCursorOff procedure

Declaration Procedure SetTextCursorOff;

Function This procedure sets the display of the diagram text cursor off.

See also SetTextCursorOn procedure

GetTextCursorOn function

Declaration Function GetTextCursorOn: Boolean;

Function Indicates whether the diagram text cursor is on.

MoveCursor procedure

Declaration Procedure MoveCursor;

Function Moves the diagram text cursor to absolute position [x,y] (window co-ordinates). *x* and *y* are variables declared in the implementation part of the DrawText unit.

See also MoveTextCursor procedure.

InitDrawTextModule procedure

Declaration Procedure InitDrawTextModule;

Function Initialises the DrawText unit.

This unit contains miscellaneous routines that cannot be associated with any of the other units. The following routines contained in this unit are identical to those described in the Misc unit of Circuit Tutor:

SwapBitsOfWord function;
ReadStr, HaltIfAbsent and Beep procedures.

The remainder of the routines exported by this unit provide a facility for displaying a message to the user.

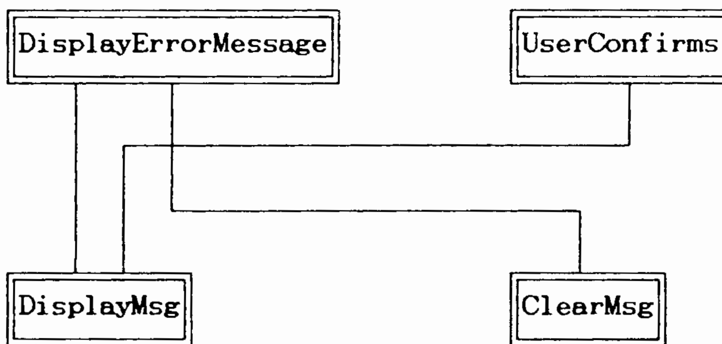


Figure 5-6. Structure of DrawMisc unit (routines described in Circuit Tutor not indicated).

DisplayErrorMessage procedure

Declaration Procedure DisplayErrorMessage(TypeOfError: String40);

Function Display an error message *TypeOfError*, which should identify the problem e.g. out of memory. The user can clear the message by pressing any key. '- PRESS ANY KEY TO CONTINUE' will be added to the message string *TypeOfError* by this routine.

ClearMsg procedure

Declaration Procedure ClearMsg;

Function Clears the displayed message.

DisplayMsg procedure

Declaration Procedure DisplayMsg(Msg: String80);

Function Display the 80 character long string *Msg*. The existing display is saved i.e. after executing ClearMsg the contents of the display will be restored.

UserConfirms function

Declaration Function UserConfirms(msg:string80): boolean;

Function Displays string *msg* to the user. *msg* should be a question that requires confirmation from the user e.g. 'CONFIRMATION'. '(Y/N)?' will be added to the message string by this routine.

This unit exports data and program objects - two variables and four procedures - to facilitate the draw of 8x8 pixel text characters to a Hercules Graphics display in graphics mode. All the routines and variables, except the DrawSymbol32x23 procedure, are identical to those of the same name in unit Chr8x8 of Circuit Tutor.

DrawSymbol32x23 Write8x8 Init8x8Module

DisplayChar8x8 StrToChar GotoXY8x8 Load8x8Font

Figure 5-7. Structure of HgcPlus unit.

DrawSymbol32x23 procedure

Declaration Procedure DrawSymbol32x23 (Symbol : font32x23;
XPos,YPos : byte;
XOffset,YOffset : byte;
Highlight : boolean;
SymbolOverwrite : boolean);

Function Draws a symbol comprising of a 32 horizontal by 23 vertical dot matrix. The symbol is drawn at circuit diagram co-ordinates [*XPos*, *YPos*]. It is however possible to draw the symbol anywhere on the screen by means of the window co-ordinate offset values [*XOffset*,*YOffset*]. If *Highlight* is true then the symbol is drawn in inverse video. When *SymbolOverwrite* is false the existing video RAM contents is not overwritten - a bitwise OR of the symbol and video memory is done.

5.8 DrawGlobals unit

This unit contains data objects global (exported) to all Circuit Draw units.

5.8.1 Miscellaneous contants

- **Version.** Current version number of Circuit Draw. When Circuit Draw is modified this number must be incremented. The current version number is 1.04/hgc, where *hgc* indicates that the program requires a Hercules Graphics Card. A minor modification will result in a new version number *1.05/hgc*. A major modification will result in a new version number *2.00/hgc*. If version *1.04* is implemented for example for the Enhanced Graphics Adaptor, then *Version* will become *1.04/ega*.
- **SymbolHeight.** Height of the symbols (number of lines).
- **SymbolWidth.** Width of symbols in 8 bit chunks.
- **MaxSymWords.** Number of words to store symbol.
- **MaxSymbols.** Number of symbols for the symbol table.

5.8.2 Graphics window constants

The Turbo Graphix toolbox (Hgc unit) permits the definition of a number of screen windows. To avoid confusion and to permit program readability each defined window has been given a constant identifier as follows:

- **SymTableCrsrW.** Symbol table cursor.
- **DgmCrsrSymW.** Circuit diagram symbol cursor.
- **DgmCrsrTxtW.** Circuit diagram text cursor.
- **DgmWindow.** Diagram window.
- **IO_Window.** User dialogue window.
- **MenuWindow.** Menu window.
- **OverwriteLblW.** Window to display overwrite status.
- **StatusWindow.** Window to display program state.
- **GridWindow.** Grid window.
- **MenuWindow1.** Store the menu for program mode 1.
- **MenuWindow2.** Store the menu for program mode 2.
- **MenuWindow3.** Store the menu for program mode 3.
- **GridCursorW.** Grid cursor.
- **GridWindowBlank.** Stores the blank grid.
- **BootMsgW.** Initialisation message window.

5.8.3 Types and Variables

The following types and variables are exported by the DrawGlobals unit:

```
type
  Font32x23=                              { Data structure to store a single
    array [1..46] of integer;               symbol. }
  SymbolArray=                            { Data structure for storing a symbol
    array[1..MaxSymbols] of Font32x23;      symbol table. }
  SymbolNumber  = 1..MaxSymbols;
  ProgramStatus =(InitState,              { State after initialisation. }
                  DrawDgmSymbols,         { Draw cct diagram symbols. }
                  DrawDgmText,            { Draw 8x8 font text. }
                  DefineSymbols);         { Define cct diagram symbols. }
  Direction = (up, down, left, right,     { Directions for moving cursors. }
               FarLeft, FarRight);
  string80  = string[80];
  string40  = string[40];

var
  Symbol       : SymbolArray;             { Contain symbol table. }
  Status       : ProgramStatus;           { Circuit Draw program state. }
  SymbolSel    : SymbolNumber;            { Symbol selected for symbol table. }
```

1. Fidler, J.K., and C.Nightingale, Computer Aided Circuit Design. Thomas Nelson, Hong Kong, 1987. pp. 112-118.

2. Gleaves, R., Modula-2 for Pascal Programmers. Springer-Verlag, New York, 1984.

3. Gray, P.E., and C.L. Searle, Electronic Principles, Physics, Models, and Circuits. John Wiley and Sons, New York, 1969. pp. 760-761.

4. Huckle, B., The Man-Machine Interface: Guidelines for the Design of the End-user/System Conversation. Savant Research Studies, Lancashire, U.K., January 1981. pp. 98-100, 135.

5. Logitech Modula-2/86, Software Development System, (3rd ed.). Logitech Inc., Redwood City, California, 1986.

6. Pinson, L.J., and R.S. Wiener, Object-Orientated Programming and Smalltalk. Addison-Wesley, New York, 1988.

7. Smith, D.J., and K.B. Wood, Engineering Quality Software. Elsevier Applied Science, London, 1987. ch. 9.

8. Stevens, W.P., Using Structured Design. John Wiley and Sons, New York, 1981.

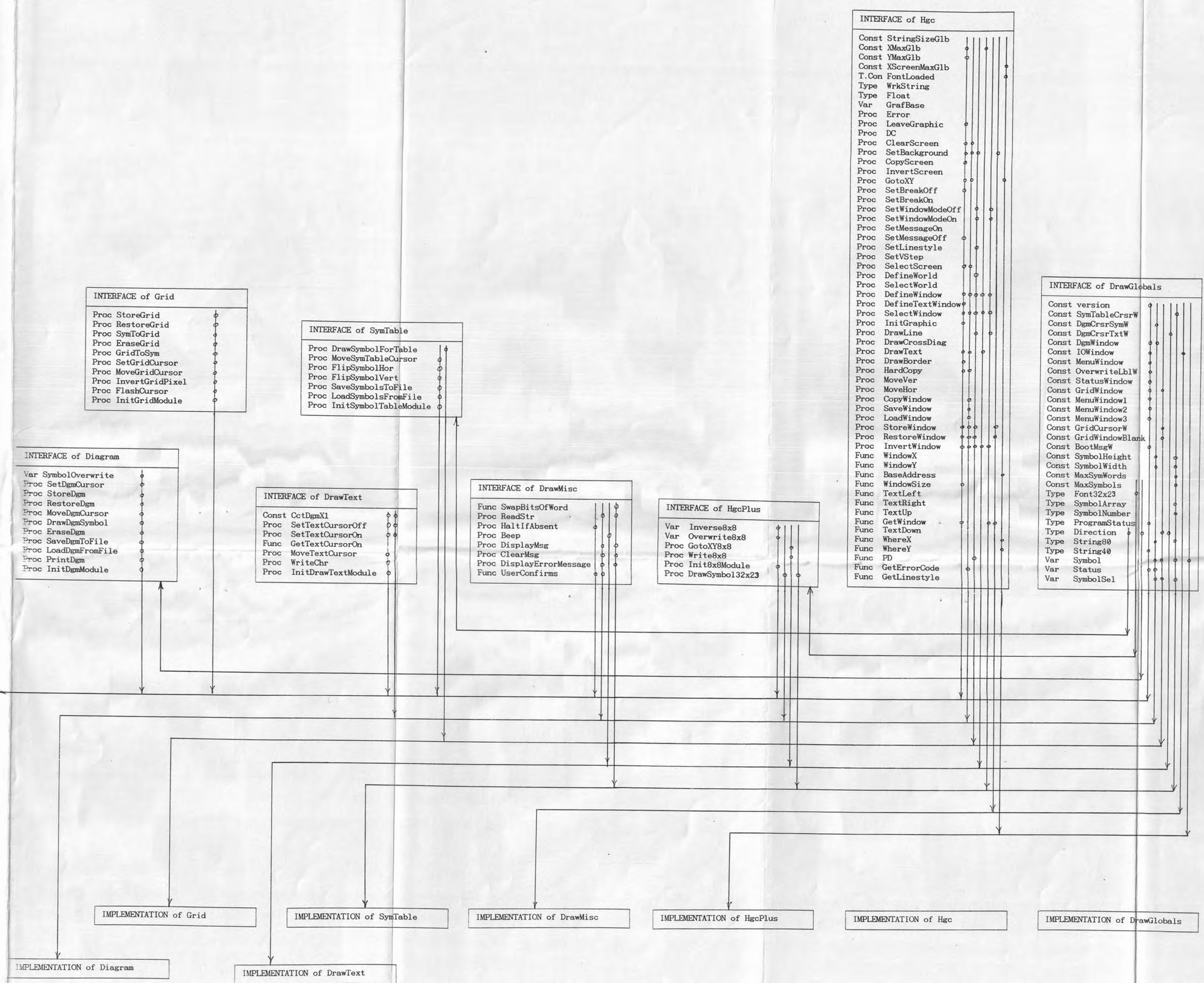
9. Turbo Pascal Graphix Toolbox, version 4, Owner's Handbook. Borland International Inc., Scotts Valley, California, 1987.

10. Turbo Pascal, version 4, Owner's Handbook. Borland International Inc., Scotts Valley, California, 1987.

11. Wiener, R.S., and R. Sincovec, Software Engineering with Modula-2 and Ada. John Wiley and Sons, New York, 1984.

12. Wiener, R.S., An Introduction to Object-Orientated Programming and C++. Addison-Wesley, New York, 1988.

13. Wirth, N., Programming in Modula-2, (2nd ed.). Springer-Verlag, New York, 1983.



Modular design chart for Circuit Draw.

Modular design chart for Circuit Tutor.