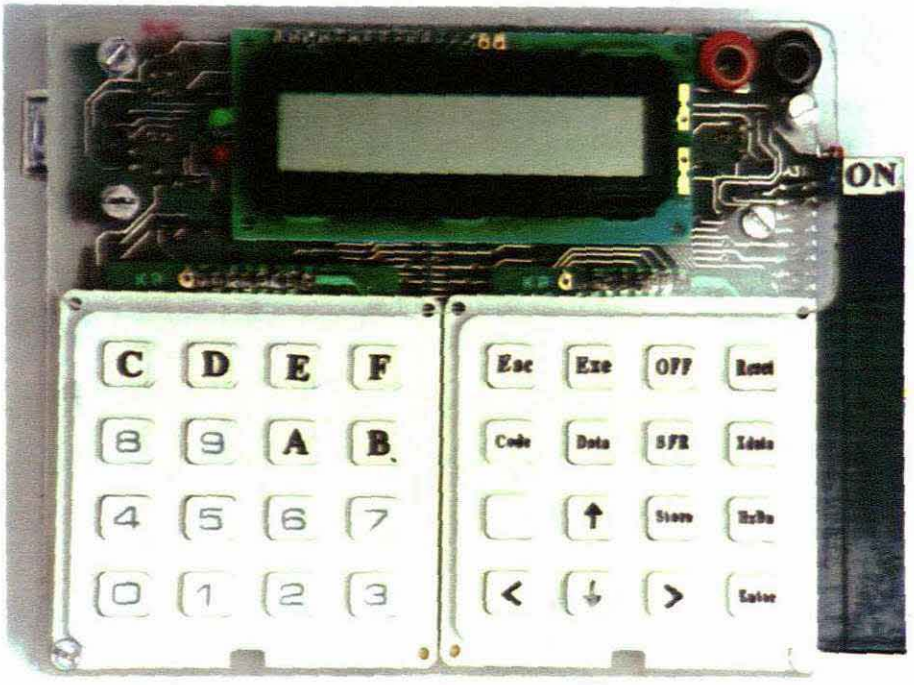
# The Development of an 8051 Micro-Controller Evaluation and Training Board

**Compiled by:** Daniel Nel de Beer




This thesis is submitted in fulfilment of the requirements for the Master Degree in Technology in the School of Electrical Engineering of the Cape Technikon

## *Statement*

It is hereby stated that the content of this thesis represents the own work of the candidate, and that the opinions herein are his own and not necessary a reflection of the opinions of the Cape Technikon.

The content of this thesis has not been submitted towards any other academic qualification.

A special word of acknowledgement must be included for the support given by the school and the staff of the School of Electrical Engineering of the Cape Technikon in terms of equipment, software, materials, time and finances.

DN de Beer Pr.Eng.

# *Summary*

The development of the 8051 Evaluation and Training Board was in response to fulfil a need to have a training board available for students at the start of a micro-controller course. This board must be used to get hands-on experience in the internal architecture, programming and operation of the controller through the testing of sample programs and exercises. It can act as an example of a practical micro-controller application board, and also as part of, or as an aid in the design and application of own projects.

The board had to be cheap enough so that each student can be issued with a personal board for the duration of the course. It had to be adequately selfsufficient to be portable and to operate independent of a host PC. In addition, it had to contain adequate "intelligence" to guide the student in the use of the board: have a quick re-programming turn-around cycle; and it must be possible to use the board for user program testing and debugging.

After drawing up an initial set of objectives and investigating the economic viability of similar systems in industry, an outline of the required design was made. This included the selection of suitable communication between the on board Operating System and a user; the easiest way to load user programs into the board memory; and methods to test and debug this program.

All the normal support circuitry required by a micro-controller to accommodate a minimum system for operation was included into a single Field Programmable Gate Array.

The execution of the project was therefore divided into three distinct sections, the hardware, the firmware (Programmable Array configuration) and the software. In the design, the harmony between these sections had to be consolidated to yield a successful final product. The simplicity and ergonomics of the operation and application from a user's point of view, had to be accentuated and kept in mind throughout.

In a design of the complexity such as this, careful planning and the investigation of various methods of approach were essential. The use of many computer-aided design and other relevant computer packages was incorporated.

Interaction between the user and the Operating System on the board was done through a standard 16-character by 1-line LCD Display Module and a 32-key keyboard. The main feature of the Operating System was to enable the inspection and editing of all the memory locations on the micro-processor. The Operating System also contained the initial intelligence on the board to load user programs from a PC through an RS232 serial link.

For the testing and debugging of a user program, three *Running Modes* are available: The *Single Step Mode*, the *Break Point Mode*, and the real time *Execute Mode*. In the *Single Step Mode* the control will be handed back to the Operating System after the execution of every instruction of the user program. For the *Break Point Mode* control will only be handed to the Operating System at user-defined break points. In the real time *Execute Mode*, the program will run at the design speed. By pressing an appropriate key, the user may create a temporary break or interrupt at any point. The control would then be handed over to the Operating System immediately.

With an intelligent display unit available, the options and selections in the Operating System was done by a so-called "menu driven" system. This makes the selection of a particular option between many, very easy. Unfortunately, with the many pathways available, it also made the software of the Operating System quite complex.

As an essential aid during the software design, various modifications to the conventional use of flow diagrams were developed to suit this specific type of application. This also proved its worth in the testing and debugging of the Operating System.

The hardware and firmware design had to cater for the decoding of all the input and output devices, the keyboard scanner and encoder, and also the external interrupt circuits. Various other features were also built into the firmware, such as the following:

Under normal circumstances the program of a micro-controller is stored in a non-volatile memory, such as an EPROM. The design of the controller chip itself does not cater in its instruction set or hardware controls, for writing to the code memory. To make it possible in this circuit to load and edit the code memory of a user program, a 32K byte battery backed-up RAM was used. Apart from the normal decoding at the lower 32K byte of the Code Memory map, it was also mapped and decoded as the *top* 32K bytes of External Data. This enabled the writing and reading of user code bytes to and from the code memory, and also the executing of these bytes as normal code.

The Operating System had to be completely transparent when a user program was tested. This means that the user program can be executed as if the Operating System was not there. To achieve this, the user program had to be placed at the normal start of the code memory map with all its user-interrupt vectors available. At power up, and when the

Operating System was taking over control, however, the Operating System requires the *same* reset and interrupt vectors address locations occupied by the user program. In fact, the whole *Single Step* and *Break Point Modes* of operation are based on a software generated interrupt and an "Interrupt 0" subroutine being available to the Operating System. The hardware was therefore designed to swap the address decoding of the first 48 code bytes of the user program with the addresses of the Operating System. Just before a user program step was executed, the *user vector addresses* would be selected. On power up or after a Single Step or Break Point interrupt, the *addresses of the Operating System* would be selected. It will be swapped long enough to make a "Long Jump" out of the vector address area into the Operating System code, higher up on the code memory map.

To facilitate the building and testing of own projects by the user, an extension socket was included on the board. It allows direct access to all the pins of the micro-controller. The board can therefore be used as an easy re-programmable minimum controller system, and the user only had to add his specific additional input and output circuits to the system. Exhaustive prototype testing can then be done before a final independent board is designed and built.

Fifteen of the training boards were built and issued to students of mixed experience in a micro-controllers course. It had to test the reaction to, and the shortcomings of the board. For the first time micro-controller students, the impact was a little overwhelming. It took a few weeks until they have gathered sufficient background and experience, before some smaller projects on the board were attempted. For the more experienced students, however, who have been through the *labourious* EPROM erase and re-programming

phases, and who were more acquainted with the structure of a micro-controller system, the ease of use and application of the board were more appreciated. Some shortcomings in the unconventional placement of the user program higher in the code memory map initially, only became apparent when program modules were written in the "C for Micro-Controller" language, and linked and placed automatically by the "Linker" program. These shortcomings have subsequently been corrected.

It can be concluded that the training board was only partially successful for new micro-controller students. Based on the observations in the course, it is recommended that the board should only be issued and used by students who are further advanced in the application of micro-controllers. A much simpler 8051 applications board would be more suitable for first time micro-controller students.

Based on the experience gained in the development of the training board, such a simpler board has subsequently been developed. It proved to be a huge success.

---

# *Samevatting*

Die ontwikkeling van die 8051 Evaluering and Opleidingsbord het ontstaan om die leemte te vul om 'n toepassingsbord vir studente onmiddelik beskikbaar te hê, reg vanaf die begin van 'n mikro-beheerder kursus. Hierdie bord kan as 'n *fisies betrokke* metode gebruik word om bekend te raak met die interne argitektuur, programmering en werking van die beheerder, deur die toetsing van gegewe programme en oefeninge. Dit kan ook dien as 'n voorbeeld van 'n praktiese mikro-beheerder toepassing, of as deel van, of as hulpmiddel in die ontwerp en toepassing van eie ontwerpe.

Die bord moet goedkoop genoeg wees sodat elke student uitgereik kan word met 'n persoonlike bord vir die duur van 'n kursus. Dit moet kompleks genoeg wees om draagbaar en op sy eie te kan werk, onafhanklik van 'n gasheer rekenaar. Ook moet dit genoeg "intellegensie" hê om die student te lei in die gebruik van die bord; dit moet gebruik maak van 'n vinnige metode vir herprogrammering; en die vermoë hê om gebruik te kan word in die toetsing en ontfouting van 'n gebruikersprogram.

Nadat die aanvanklike objektiewe daargestel en die ekonomiese vatbaarheid van soortgelyke stelsels in die industrie ondersoek is, is 'n geheelbeeld van die benodigde ontwerp gemaak. Dit het die keuse van toepaslike kommunikasie tussen die bedryfstelsel en die gebruiker; die maklikste metode om gebruikersprogramme in die geheue te laai; asook metodes om hierdie program te toets en te ontfout, ingesluit.

Al die normale ondersteuningstroombane benodig vir 'n minimum mikro-beheerder stelsel is ingebou op die board in 'n enkele Bedryfsprogrammeerbare Logiese Hek Reeks-stroombaan ("FPGA"). Die uitvoering van die projek was daardeur ingedeel in drie

definitiewe afdelings, die hardeware, die fermware (die samestelling van die Programmeerbare Reeks-stroombaan) en die sagteware. Die nougesette samewerking tussen hierdie stelsels was noodsaaklik vir die suksesvolle werking van die finale produk. Die eenvoud en ergonomie van die werking en toepassing vanuit 'n gebruikersoogpunt, moes beklemtoon en gedurig voor oë gehou word.

In 'n ontwerp met sulke kompleksiteit, is versigtige beplanning en die ondersoek van verskeie metodes van benadering noodsaaklik. Die gebruik van vele rekenaar gerugsteunde ontwerp- en ander toepaslike rekenaar pakkette is ingespan.

Interaksie tussen die gebruiker en die Bedryfstelsel van die bord word gedoen deur 'n standaard 16-karakter by 1-lyn LCD vertooneenheid en 'n 32-sleutel sleutelbord. Die hoof eienskap van die Bedryfstelsel is die inspeksie en nasien van al die geheue adresse van die mikro-beheerder stelsel. Die Bedryfstelsel bevat ook die nodige intelligensie op die bord om gebruikersprogramme vanaf 'n rekenaar te laai deur 'n RS232 serie verbinding.

Vir die toetsing en ontfouting van 'n gebruikersprogram is drie uitvoeringsmetodes beskikbaar, die *Enkel Trap Metode*, die *Breekpunt Metode* en die intydse *Uitvoeringsmetode*. In die *Enkel Trap Metode*, sal beheer oorhandig word aan die Bedryfstelsel na die uitvoering van elke enkele instruksie van die gebruikersprogram. Vir die *Breekpunt Metode*, sal beheer net oorhandig word aan die Bedryfstelsel op gebruikersgedefineerde punte. In die intydse *Uitvoeringsmetode*, sal die program teen ontwerpspoed loop. In hierdie metode kan die gebruiker die program op enige plek tydelik onderbreek deur 'n toepaslike sleutel te druk. Beheer sal dan onmiddelik oorgeplaas word na die Bedryfstelsel.

Met 'n intelligente vertooneenheid beskikbaar, is die opsies en keuses in die Bedryfstelsel beheer deur 'n sogenaamde "spyskaart-aangedrewe stelsel". Dit maak die keuse van een spesifieke opsie tussen vele ander baie maklik. Ongelukkig, met so veel moontlike uitvoeringspaaie beskikbaar, was die sagteware van die Bedryfstelsel redelik kompleks.

As 'n noodsaaklike hulpmiddel in die sagteware ontwerp is verskeie verbeterings gemaak aan die konvensionele gebruik van vloeidiagramme om aan te pas by die spesifieke toepassing van hierdie projek. Hierdie verbeterings het ook hulle nut bewys tydens die toetsing en ontfouting van die Bedryfstelsel.

Die hardeware en fermware van die ontwerp moes voorsien vir die dekodering van die inset en uitgang randapparate, die sleutelbord skandering en dekodering, asook die eksterne onderbrekingstroombane. Verskeie ander spesiale toepassings is ook ingebou in die fermware, waaronder die volgende:

Onder normale omstandighede sal die program van 'n mikro-beheerder in 'n nie-vernietigbare geheue, soos 'n EPROM, gestoor word. Die beheerder maak nie voorsiening in die instruksie stel of in die hardeware beheer vir die skryf van data na die kode geheue nie. Om dit moontlik te maak vir hierdie stelsel on gebruikersprogramme te laai en te verander in die kode geheue, is 'n 32K greep battery-gerugsteunde lees-en-skryf geheue-element (RAM) gebruik. Bo en behalwe die normale toegang as kode geheue op die onderste 32K greep helfte van die kode geheue, is die element ook gedekodeer om beskikbaar te wees vir lees en skryf op die *boonste* helfte en die eksterne data geheue. Dit was nou moonlik om masjienkode data grepe na die kode geheue te lees en te skryf, en hierdie grepe dan te kan uitvoer as gewone kode.

Die Bedryfstelsel moes geheel en al "deursigtig" wees terwyl 'n gebruikers-program uitgevoer word. Dit beteken dat die program uitgevoer moet word asof die Bedryfstelsel nie bestaan nie. Om dit te bewerkstellig moes die gebruikersprogram in die onderste gedeelte van die kode-geheue geplaas word, met al die onderbrekingsvektore van die gebruikersprogram beskikbaar. Tog moes die selfde herstel- en onderbrekingsvektor adresse beskikbaar wees vir die Bedryfstelsel nadat die stelsel aangeskakel is, of as die beheer oorgeplaas word vanaf die toetsing van 'n gebruikers program. Die hele *Enkel Trap* en *Breekpunt* uitvoeringsmetodes is inderdaad gebasseer op 'n sagteware-gegenereede onderbreking en die gebruik van sy eie "Onderbrekingsroetine nommer 0" deur die Bedryfstelsel. Die hardeware is dus ontwerp om die eerste 48 kode grepe van die gebruikersprogram om te ruil met dié van die Bedryfstelsel. Net voor 'n instruksie van die gebruikersprogram uitgevoer gaan word, word die *gebruikersadres-gedeelte* ingeskakel. Na die aanskakeling van die bord, of na 'n *Enkel Trap* of *Breekpunt* onderbreking moet die *Bedryfstelsel se adres-gedeelte* ingeskakel word. Die omruiling sal lank genoeg duur om uit die lae adres-gedeelte te spring met 'n "Long Jump" na die Bedryfstelsel hoër op in die kode geheue.

Om die bou en toets van eie projekte deur die gebruiker moontlik te maak, is 'n uitbreidingsok aangebring op die bord. Dit verleen direkte toegang na al die verbindings-pennetjies van die mikro-beheerder. Die bord kan dan gebruik word as 'n maklik herprogrammeerbare minumum stelsel. Die gebruiker hoef slegs sy eie addisionele inset en uitgangstroombane by te voeg. Uitgebreide en volledige prototipe toetsing kan dan gedoen word voor die ontwerp en bou van sy eie finale stroombaan.

Vyftien van die opleidingsbordjies is gebou en uitgereik aan studente met wisselende ervaring in 'n mikro-beheerder kursus. Dit sou die reaksie teenoor, en die tekortkominge van die bord toets. Vir nuwe mikro-beheerder studente was die impak 'n bietjie oorweldigend. Dit het 'n hele paar weke geneem voordat die studente genoeg agtergrond en ondervinding opgebou het om klein projekkies op die bord te toets. Meer gevorderde studente, wat reeds deur die *langdradige* EPROM uitvee- en herprogrammeringsfases was, en ook meer bekend was met die struktuur van 'n mikro-beheerder stelsel, het die gemak van die gebruik en die toepassing van die bord meer waardeer. Daar was 'n paar tekortkominge soos die ongerief van die onkonvensionele plasing van die gebruikersprogram oorspronklik hoër op in die kode geheue. Dit het eers duidelik geword nadat "C vir Mikro-Beheerder" programmodules outomaties deur die "Linker" gekoppel en geplaas is in die finale program. Hierdie tekortkominge is sederdien reggestel.

Die gevolgtrekking kan gemaak word dat die opleidingsbord net gedeeltelik suksesvol was vir *nuwe* mikro-beheerder studente. Gebasseer op die waarnemings gemaak tydens die kursus, word dit aanbeveel dat die bord net aan meer gevorderde studente uitgereik moet word. 'n Eenvoudiger 8051 toepassingsbord sal meer van toepassing wees op die nuwe mikro-beheerder studente.

Gebasseer op die kennis opgedoen in die ontwikkeling van hierdie opleidingsbord, is 'n eenvoudiger opleidingsbordtjie sederdien ontwerp. Dié het homself reeds bewys as 'n groot sukses.

---

# Table of Contents


| | | *Page* |
|---|---|---|
| | *List of Illustrations* | *viii* |
| | *Software Routine Listing Page References* | *xi* |
| | *Glossary of Terms* | *xiii* |
| **1.** | **Introduction** | 1 |
| 1.1. | The Development of Micro-Controllers | 1 |
| 1.2. | Teaching Micro-Controller Operation and Applications | 3 |
| 1.3. | The Basic Requirements of the Training Board | 4 |
| **2.** | **Design Objectives** | 6 |
| **3.** | **Investigation of Similar Existing Systems** | 11 |
| **4.** | **Design Considerations** | 12 |
| 4.1. | Basic Architecture and Memory Map of an 8051 Micro-Controller | 12 |
| 4.2. | Access to External Data | 14 |
| 4.3. | The Memory Map Organisation for this Project | 16 |
| 4.4. | Interrupt Code Memory Segment Swapping and Decoding | 18 |
| 4.5. | Hardware Address Decoding and Other I/O Circuits | 19 |
| 4.6. | The LCA Configuration | 20 |
| 4.7. | The Initial Seeding Process | 21 |
| **5.** | **Product Description** | 23 |
| 5.1. | General Layout and Component Identification | 23 |

5.2. Communication Between the User and the Operating System 25

5.3. An Overview of the Modes of the Operating System 27

5.4. The On-screen Help Facility 30

**6. Product Specification** 31

6.1. General Specifications 31

6.2. Computer Software Requirements and Data Files 32

6.3. The RS232 Serial Link 32

6.4. The 64K Byte Code Memory Map 32

6.5. The 64K Byte External Data Memory Map 32

**7. Operating Modes** 34

7.1. The Operating System Mode 34

7.2. The Single Step Mode 34

7.3. The Break Point Mode 35

7.4. The Execute Mode 35

**8. Operating Procedures** 36

8.1. Setting-Up Procedure 36

8.2. Power-Up and LCA Initialization 37

8.3. The Help Menus 39

8.4. The Main Menu and Sub-Menu Selection 39

8.5. Mode 1: Edit Memory Segment Mode 41

8.6. Mode 2: Edit Pre-Interrupt Registers 44

8.7. Mode 3: Store Internal Data and System Variables 47

8.8. Mode 4: Restore Internal Data and System Variables 47

8.9. Mode 5: Execute a User Program 48

8.10. Mode 6: Execute a User Program in the Single Step Mode 50

8.11. Mode 7: Execute a User Program in the Break Point Mode 51

8.12. Mode 8: Clear Internal Data Memory 54

8.13. Mode 9: Clear the User Program Area 54

8.14. Mode A: Clear the Break Point Table 55

8.15. Mode B: Load a Program or Data through the RS232 Link 55

8.16. Mode C: Dumping a Block of Data through the RS232 Link 58

8.17. Mode D: Move a Block of External Data 61

8.18. Mode E: Protect/Unprotect Memory 62

8.19. Mode F: Save Internal Data and Switch Off 62

8.20. The Seeding Process of the Board 63

**9. The Hardware and Firmware Description** 70

**9.1. An Overview of the *Xilinx* 2064 Series LCA** 71

9.1.1 The Configurable Logic Block (CLB) 71

9.1.2 The Storage Element 73

9.1.3 The Complete Configurable Logic Block 74

9.1.4 The Input/Output Blocks 75

9.1.5 CLB and I/O Block Interconnections 76

9.1.6 The LCA Special and General Purpose Pins 76

9.1.7 The Configuration Modes for an LCA device 77

9.1.8 The Design Entry to Realise an LCA Configuration 78

**9.2. The Hardware Circuits Required to be Built into the LCA** 80

9.2.1 The Address Latch 80

9.2.2 The Keyboard Scanner and Decoder 81

9.2.3 The LCA Control Register 84

9.2.4 The Code RAM Decoding 84

9.2.5 The External Data RAM Decoding 85

9.2.6 The Display Module Decoding 86

9.2.7 The Two User I/O Select Line Decoders 86

9.2.8 The Micro-Processor Reset Control 87

9.2.9 The Interrupt 0 and Reset Vector Swapping Circuits 87

**9.3. Additional Hardware Circuits External to the LCA** 89

9.3.1 The Power Control Circuits 89

9.3.2 The RS232 Serial Link to the PC 91

9.3.3 The Seeding Circuits and Jumpers 92

9.3.4 The LCA Configuration for the Seeding Process 94

9.3.5 The 62-Pin Extension Edge Connector 96

**10. Software Description** 98

10.1. Approach to Writing the Operating System 98

10.2. Assembler Language vs "C for Micro-Controllers" 101

10.3. An Overview of the Operating System 102

10.4. Internal Data System Variables and Stack Definition 103

10.5. System Input/Output Addresses 103

10.6. General Startup Routines after Power-Up or a Reset 104

10.7. Mode 0: The Main Menu 108

10.8. Mode 1: Edit Memory Segment 112

10.9. Mode 2: Edit Pre-Interrupt Registers 130

10.10. Modes 3 and 4: Store and Retrieve Internal Data and System Variables 141

10.11. Mode 5: Execute a User Program in Real Time 149

10.12. Modes 6 and 7: Run a User Program in the "Single Step" or "Break Point" Modes 150

10.13. Modes 8, 9 and A: Bulk clearing of Internal Data, the User Program Area and the Break Point Table 163

10.14. Mode B: RS232 Serial Load from a PC 167

10.15. Mode C: Dump External Data to the PC through the RS232 Link 179

10.16. Mode D: External Data Block Move 185

10.17. Modes E and F: Protect/Unprotect Memory and Save and Off Modes 191

10.18. The Help Subroutine 196

10.19. The Interrupt 0 Subroutine 201

10.20. Restore and Return to Execute the User Program Routine 217

10.21. Supplementary Subroutines: Delay, Display and Keyboard Drive 222

**11. Power Economy and Conservation** 230

**12. The Hardware and Software Debugging and Performance Tests** 232

12.1. Hardware Integrity and Power Supply Control Tests 232

12.2. Fault Finding the Rest of the Hardware 232

12.3. The Manual Input/Output Micro-Controller Emulator 233

12.4. Hardware and Firmware Testing 235

12.5. Software Testing and Debugging 236

12.6. Classroom Application 237

**13. Guidelines on the Writing and Assembling of User Programs** 240

13.1. Performance Testing a User Program 244

13.2. Test Results 247

**14. Conclusions and Recommendations** 251

**15. Future Modifications and Extensions** 259

**16. Other Project Developments as a Direct Result of this Project** 262

16.1. A Simple EPROM Emulator 262

16.2. A Simple 8051 Emulator 262

16.3. A Simpler 8051 Training Board for First Time Students 264

16.4. The PIC Development System 265

**17. Bibliography** 266

../Appendices

## Appendices


| | | *Page* |
|---|---|---|
| A: | The Complete External Board Schematic Diagram | 270 |
| B: | The Complete DECODER Schematic Diagram | 271 |
| C: | The Complete SHIFT REGISTER Schematic Diagram | 272 |
| D: | Numbers and Functions of the Extension Edge Connector Pins | 273 |
| E: | Memory Map Allocations and Bulk Storage Addresses | 277 |
| F: | The Pinouts and Functions of the *XC2064* PLCC68 LCA Package | 280 |
| G: | The Seeding Serial Data Download Cable | 283 |
| H: | Listing of the Seeding Driver Program: SEED.EXE | 284 |
| I: | Useful Subroutines Already Existing in the Operating System | 291 |
| J: | The LCD Display Module and Control Commands | 296 |
| K: | The Binary and INTEL Hex Format Files | 305 |
| L: | The Component List for the Training Board | 308 |
| M: | The Component Overlay and Connections | 312 |
| N: | The Micro-Controller Board Netlist | 313 |
| O: | Costing and Suppliers of the Components of the Board | 318 |

# *List of Illustrations*


| | *Page* |
|---|---|
| Fig. 4.1. Memory Maps Available to the 8051 Micro-Controller | 13 |
| Fig. 4.2. Access to External Memory | 15 |
| Fig. 4.3. Dual Mapping of the Code Memory IC | 17 |
| Fig. 4.4. Reset and Interrupt Vector Swapping | 19 |
| Fig. 5.2. The Layout of the Key Pads | 25 |
| Fig. 9.1. The CLB Combinatorial Logic Circuit | 71 |
| Fig. 9.3. The Combinatorial Circuit Options | 73 |
| Fig. 9.3. The CLB Storage Element | 73 |
| Fig. 9.4. The Storage Element Options | 74 |
| Fig. 9.5. The Complete Configurable Logic Block | 74 |
| Fig. 9.6. The LCA Input/Output Block | 75 |
| Fig. 9.7. The Address Latch | 81 |
| Fig. 9.8. The Keyboard Scanner and Decoder | 82 |
| Fig. 9.9. The Interrupt 0 Circuit Diagram | 88 |
| Fig. 9.10. The Power Control Circuits | 90 |
| Fig. 9.11. The RS232 Cable Through Connections | 92 |
| Fig. 9.12. The External Seeding and Select Circuits | 94 |
| Fig. 10.1. A Broad Based Flow Diagram | 100 |
| Fig. 10.2. The Initialisation Flow Diagram | 105 |
| Fig. 10.3. The Main Menu Operating Diagram | 108 |
| Fig. 10.4. The Main Menu Flow Diagram | 109 |

Fig. 10.5. Edit Memory Segment Operating Diagram (2 pages) 114

Fig. 10.7. Edit Memory Segment Flow Diagram (4 pages) 116

Fig. 10.11. Edit Pre-Interrupt Registers Operating Diagram 132

Fig. 10.12. Edit Pre-Interrupt Registers Flow Diagram (2 pages) 133

Fig. 10.14. Save Data and System Variables Operating Diagram 143

Fig. 10.15. Retrieve Data and System Variables Operating Diagram 143

Fig. 10.16. Save Data and System Variables Flow Diagram 144

Fig. 10.17. Retrieve Data and System Variables Flow Diagram 145

Fig. 10.18. Execute a User Program Mode Operating Diagram 151

Fig. 10.19. Execute a User Program Mode Flow Diagram 152

Fig. 10.20. Single Step Selection Operating Diagram 157

Fig. 10.21. Break Point Selection Operating Diagram 158

Fig. 10.22. Single Step & Break Point Selection Flow Diagrams 159

Fig. 10.23. Mode 8: Clear Internal Data Operating Diagram 163

Fig. 10.24. Mode 9: Clear Program Area Operating Diagram 163

Fig. 10.25. Mode A: Clear Break Point Table Operating Diagram 163

Fig. 10.26. Mode 8, 9 & A: Clear Memory Areas Flow Diagrams 164

Fig. 10.27. RS232 Load Operating Diagram 170

Fig. 10.28. RS232 Load Flow Diagram 171

Fig. 10.29. Select Baud Rate Operating Diagram 174

Fig. 10.30. Select Baud Rate and Enter 2 Keys Flow Diagrams 175

Fig. 10.31. Dump Data through the RS232 Link Operating Diagram 180

Fig. 10.32. Dump Data through the RS232 Link Flow Diagrams 181

Fig. 10.33. An Overlapping Block Move 185

Fig. 10.34. External Data Block Move Operating Diagram 186

Fig. 10.35. External Data Block Move Flow Diagram 187

Fig. 10.36. Protect/Unprotect Memory Operating Diagram 192

Fig. 10.37. Save and Off Operating Diagram 192

Fig. 10.38. Protect/Unprotect Memory and Save & Off Flow Diagrams 193

Fig. 10.39. Help Subroutine Operating Diagram 196

Fig. 10.40. Help Subroutine Flow Diagram 197

Fig. 10.41. Interrupt 0 Subroutine Operating Diagram 209

Fig. 10.42. Interrupt 0 Subroutine Flow Diagram (2 pages) 210

Fig. 10.44. Restore and Return Operating Diagram 217

Fig. 10.45. Restore and Return Flow Diagram 218

Fig. 10.46. Supplementary Subroutine Flow Diagrams 224

Fig. 12.1. The Manual Emulator Basic Circuits 234

Fig. A-1. The Complete External Board Schematic Diagram 270

Fig. A-2. The Complete DECODER Schematic Diagram 271

Fig. A-3. The Complete SHIFT REGISTER Schematic Diagram 272

Fig. A-4. The Seeding Data Cable Connections 283

Fig. A-5. The Component Overlay and Connections 312

# Software Routine Listing Page References


| | *Page* |
|---|---|
| Initialisation Routine | 106 |
| Main Menu - Mode 0 | 110 |
| Mode 0 Jump Table | 111 |
| Edit Memory - Mode 1 | 120 |
| Write Data to Variable SFR Subroutine | 128 |
| Read Data from Variable SFR Subroutine | 128 |
| Read Data from Selected Memory Segment Subroutine | 129 |
| Edit Pre-Interrupt Registers & Data - Mode 20 | 135 |
| Move or Increment Cursor Add for Mode 2 Subroutine | 140 |
| Save Internal Data & System Variables - Mode 30 | 146 |
| Save Internal Data and SFR Subroutine | 146 |
| Restoring Data & System Variables - Mode 40 | 147 |
| Execute a User Program - Mode 50 | 153 |
| Select Single Step Execution - Mode 60 | 160 |
| Edit Break Points and Start Execution - Mode 70 | 160 |
| Clear Internal Data - Mode 80 | 165 |
| Clear User Program Area - Mode 90 | 165 |
| Clear Break Point Table - Mode A0 | 166 |
| Load Data Through RS232 - Mode B0 | 172 |
| Receive HEX Format Byte Subroutine | 174 |
| Select BAUD RATE Subroutine | 176 |
| Enter 2 Bytes from Keyboard Subroutine | 177 |

Dumping Memory to RS232 - Mode C0 182

Transmit Nibble or Byte Subroutine 184

XData Block Move - Mode D0 188

Move Block by Incrementing Addresses Subroutine 190

Protect/Unprotect Memory - Mode E0 194

Write Protect/Unprotect OPSYS in Xdata Segment Subroutine 194

Save and Off - Mode F0 194

Help Subroutine 198

Help file Sequences 199

Help File Messages 199

Operating System Reset and Interrupt Vectors 212

Interrupt 0 Subroutine from 5003H 212

Restore & Exit Routine 219

Write some Data or Instruction to Display Module Subroutine 225

Wait Short Subroutine (40 $\mu$S) 225

Wait 1 or Wait a Part of a Second Subroutine 225

Clear Display Subroutine 226

Display 16-Character Message Subroutine 226

Read Keyboard Subroutine & Auto-off Timing 227

Binary Nibble to ASCII Conversion Subroutine 228

ASCII to Binary Conversion Subroutine 228

Messages Listing 228

User Test Program Listing 244

---

# *Glossary of Terms*

| | |
|---|---|
| Assembler | "Low level computer programming language" / "Software Tool program to convert an Assembler program into Machine Code Instructions". |
| Checkbyte | The byte value that, when added to the sum of a number of bytes (carries ignored), will make the total sum equal to zero. |
| Chip | Integrated Circuit |
| CLB | Configurable Logic Block |
| Code | Machine Code Instructions |
| DIL | Dual-In-Line (IC socket pins) |
| DIP | Dual-In-Line Pin Package |
| DMA | Direct Memory Access |
| DXF | A format of saving and transferring Drawing Files |
| EEPROM | Electrically Erasable Programmable Read Only Memory |
| EPROM | Erasable Programmable Read Only Memory |
| IC | Integrated Circuit |
| INTEL | International Micro-Processor Manufacturer |
| LCA | Logic Cell Array |
| LCD | Liquid Crystal Display |
| LED | Light Emitting Diode |
| LSB | Least Significant Bit |
| MAX232 | IC used to Convert Conventional Logic Levels to RS232 levels |
| MSB | Most Significant Bit |

| | |
|---|---|
| Netlist | List of the connections between the pins of all the components of an electronic circuit. |
| Non-Volatile | A memory IC that would *not* lose its contents when the power is removed |
| Peripherals | Surrounding Support Devices |
| PC | Personal Computer |
| PCB | Printed Circuit Board |
| RAM | Random Access Memory - (normally volatile) |
| Rasterscan | Scanning the rows and columns of a matrix, such as the keys of a keyboard. |
| RS232 | A Two-wire Computer Serial Communication Standard |
| SFR | Special Function Register(s) |
| Smart Socket | An IC socket for a memory chip that contains a backup battery |
| Seeding | The placing of the initial intelligence on the system |
| Userfriendly | Comfortable and predictable user interface with a machine or program |
| Volatile | A memory IC that would lose its contents when the power is removed |
| Vcc | The positive supply voltage of an electronic circuit |
| Xdata | External Data (Memory) |
| Xilinx | International Manufacturer of Logic Cell Arrays |

==========

# 1. Introduction

## 1.1. The Development of Micro-Controllers

During the last decade or more, two distinct directions of computer system development became apparent:

On one side was the development for faster systems, with bigger memory capacity to handle large amounts of data. Overall, they were more versatile as *general purpose* computers that can be used in a variety of applications. This resulted in, for example, the *INTEL* 386, 486 and Pentium *IBM* compatible family of micro-processors. Simultaneously it also introduced a whole family of support chips that were essential for effective operation. These included the Bus Controller, the Programmable Timer, the Programmable Interrupt Controller, the DMA Controller, the Maths Co-processor and Parallel and Serial Interfaces. Some of these *support* chips have been incorporated into the later processors, or consolidated in large customer-specific ICs, but the aim was still to support a large bank of RAM or Random Access Memory. This inevitably led to systems with a complex hardware layout and a multiple of parallel interconnections between the various chips.

To maintain the versatility and general purpose application of the system, *intelligence* must normally be loaded into the system from a magnetic disk every time the computer is used. The standard interfaces for communication to and from the user are the computer screen and a keyboard.

The second direction of development was to be able to use the intelligence of computer power in simpler and more specific practical applications. To reduce the cost the hardware and chip count was reduced to a minimum. Intelligence should be available in the system at power-up in a non-volatile form that would not be destroyed when the power is switched off. For dedicated systems only a minimum level of communication with external hardware is required. This made the normal keyboard and monitor screen interfacing with a user superfluous. Inputs and outputs must be limited to include only the essential elements to maintain effective control and userfriendlyness: Inputs may be limited to one or a few push-buttons or sensors for directional control, and the outputs may be limited to a LED lighting up, or a relay switch being closed. Once a system is switched on, it should be able to initiate and do what it was designed to do, without any additional help.

This direction resulted in the development of the so-called **micro-controllers**, where all the essential parts of a working system were incorporated into a single chip. Special emphasis was placed on economy, hardware flexibility to adapt to various applications, and also power conservation. One such family of micro-controllers is the **8051** series from *INTEL*.

In industry, the 8051 embedded micro-controller is a very popular and versatile electronic tool. It can be used where-ever one requires a control system to have a bit more intelligence than what can be achieved through straightforward logic circuits. A personal computer or PC is designed to be as flexible as possible, to be able to manipulate and store large amounts of data and to have easy interaction between the computer and the user. In contrast a micro-controller is usually

dedicated to perform a specific task by using a fixed program. For economical reasons it is provided with just sufficient memory and other essential hardware to do the task. User interaction may be cut down to the bare necessities such as a single push button or a LED indicator.

Devices, such as programmable microwave ovens, sewing- and washing machines, electronic musical instruments, and also portable and battery driven devices, such as calculators, instruments and clever toys, are all perfectly suited to be controlled and made intelligent by a micro-controller.

## 1.2. Teaching Micro-Controller Operation and Applications

Industry is continuously in need of technicians who are familiar with the application of micro-controllers. At the Cape Technikon it was decided to dedicate some subjects to the operation and application of the 8051 series of micro-controllers. This will acquaint the students with micro-controller systems up to a point where they can be of immediate use in industry.

It usually takes a few weeks for a student in such a course to know the basic micro-controller system sufficiently well to design and build his own applications board. Furthermore they need considerable additional time to sort out all the hardware problems before the circuit finally works. This leaves very little time, in a semester course, to complete even a single project and still has sufficient practice in writing software and obtains a wider experience in other applications of the micro-controller as well.

To enable the students to start getting experience in the software and the building of small application projects as soon as possible, it has been decided to develop and build a relatively simple but versatile evaluation and training board. This board will contain all the essential parts of a minimum system that can easily be adapted to use in a wide variety of student projects and other applications.

Having the essential parts of a micro-controller system available on such a board, only a few project-specific peripheral components, such as sensors, analogue-to-digital converters, opto-isolators, etc., need to be added on an extension board to complete the circuit. It must then be possible, with a minimum turnover time; to write and alter software programs on a PC; to load it into the micro-controller immediately; and to do a few sample runs to test and improve the hardware and software.

## 1.3. The Basic Requirements of the Training Board

While micro-controllers were designed to be as simple as possible and very application-specific, such a training board must make the micro-controller chip sufficiently versatile and flexible to be used in a large variety of applications. This board should not hide or limit the basic system in a cluster of specific peripheral hardware devices and operation system software. The user must continuously be aware of and in touch with all the basic features and elements of the micro-controller. The user must still be able to debug the software and hardware and re-program the device with ease.

To make the board portable and independent of a host PC, the board must contain sufficient input control and output monitoring capabilities. It must use the computing power of the chip itself to provide a built-in Operating System. Neither the software of the Operating System, nor the hardware circuits used for monitoring the execution of a user application, must interfere with the normal operation of a user program. When a PC is available, it must also be possible to download a program with ease, or upload a current program or data to the PC for disk storage.

---

# 2. Design Objectives

1. The board must be as simple and as cheap as possible, easy to assemble, and all the components must be readily available.

2. The board must be portable, selfsufficient and independent of a host PC.

3. For economy, the board must be based on the 80C31 micro-controller. It must also be possible to use any of the other micro-controller members of the 8051 family as well, such as the 80C32, the factory programmed 8051 and 8052, or the EPROM based 87C51 and 87C52. For initial testing, it must be possible to disable the onboard code memory of those devices with built-in code memory.

4. The system must have sufficient peripheral devices to enable sensible user communication with the board without any additional external equipment.

5. User data entry and the control of the system must be from two 16-key pads, one for normal hexadecimal values, and the other for general system control.

6. User monitoring of the Operating System must be through a 16-character by 1-line LCD Display Module.

7. The I/O devices must not limit or tie down the general use of any of the ports, so that the system can be used and extended for user applications without any prerequisites.

8. All the relevant signal lines of the micro-controller must be available for connection to external test and application circuits through a 62-pin extension socket.

9. The system must have approximately 32K bytes of Code memory and 32K bytes of external data memory available.

10. The code and data memory must be battery backed-up RAM based (Random Access Memory), using *smart sockets*, to make it possible to re-program the code memory without the normal twenty minute ultraviolet exposure for erasing and the special programming equipment required by EPROMs.

11. The code memory must be Random Access Memory (RAM). After a user program has been loaded into the system, it must be possible to edit and change the byte values on the board, without having to go through the complete program assembling procedure again.

12. The programming procedures and requirements must be simple and portable to any PC, and must not be limited to a few PCs with special hardware and software installations for programming.

13. Programming the code memory must be possible, without having physically to remove the device or any other IC from the board.

14. Using *smart sockets*, the system does not require an additional battery for memory backup to maintain the data when the system is transported between external power supplies. A small onboard battery must be included to make

the board portable and supply power to the board for normal operation when an external supply is not available.

15. The supply to the board must be either from the onboard battery, or from an external supply, with a *fool proof switch-over system* between the sources. This is essential to not charge and damage the onboard battery when an external supply is used.

16. The system must be protected against accidental supply reversal, and have an onboard regulator to desensitise and protect the circuit against supply voltage variations and accidental circuit over-currents.

17. The system must contain an onboard Operating System, which must incorporate the use of the LCD Display Module and the keyboard. This Operating System must enable access and editing of the Code, the Internal and External Data Memories and the Special Functions Registers of the micro-controller.

18. The Operating System must enable the loading of programs and data from a PC through the RS232 serial port, and also the dumping of programs and data to the PC for disk storage.

19. Through the Operating System, it must be possible to execute a user program in a real time *Execute Mode*; or run a program in the *Single Step Mode*; or run a program in user defined *Break Point Mode*. The last two modes should provide access for inspection and editing of all the registers and memory

locations after each step or breakpoint.

20. After any Single Step or Breakpoint interrupt, the storing of all the system variables and Special Function Register values must be possible. This data can be retrieved at a later stage, so that a program can be started again at the same program step and with exactly the same system variables and SFR values.

21. The Operating System must also enable bulk erasing of the Code segment, the Internal Data memory segment and the user defined breakpoint table.

22. Block movement of data of any size and direction must be possible between any combination of code, internal or external data memory.

23. The system must contain a three minute auto-off facility to conserve power, when the system is not used during the prescribed period. Before the system switches off, the micro-controller contents must be stored in the battery backed up RAM, so that important data and settings will not be lost.

24. On-screen help facilities must exist for all operating modes to assist a user in the explanation of each mode, and the action keys required for successful operation.

25. To place the initial intelligence on the board, a *seeding* process through a serial download connection to the parallel printer port of a PC must exist. This will load the Operating System into the battery backed-up code memory. Afterwards, the RS232 port can be used for all other downloading.

26. A program for the PC must be written to read the seeding-, the normal configuration-, and the Operating System data from a disk file and convert it into a suitable format for the *seeding* process.

27. To investigate the power requirements of the circuit, and adjust, redesign and adapt the circuit for minimum power dissipation and battery economy.

28. To compile a comprehensive operating- and maintenance manual for the system.

29. To test the effectiveness of the training board in a classroom situation, where each student will be issued with a personal training board for the duration of the course.

30. Based on the response obtained from the students, the hardware, firmware and software must be updated to suit any omitted requirements.

31. The general operation and programming of the board must be simple, ergonomic and userfriendly.

---

# 3. Investigation of Similar Existing Systems

There are various micro-controller development and training systems available. They mainly consist of some type of basic micro-controller board with some input and output devices for limited user communication and control. Most of these systems use EPROM based code memory for the Operating System and vollitile RAM for the user programs, mapped higher up in the code memory segment.

An EPROM based code memory system is totally unsuited for a classroom situation, as each student cannot afford his personal EPROM eraser and programmer. That means he has to queue up for erasure and programming of his EPROM chip, every time he makes the slightest change in his software development.

One system from DALLAS, the DS5000, does have programmable code memory, but software from a PC entirely drives this system, and can only work while it is connected to the serial port.

Various evaluation systems, similar to this project, have been designed and built by the author over the last fifteen years. All of these systems have had some shortcomings and limitations. These should be amended by this project.

None of the systems investigated complied with the economy, flexibility, portability and userfriendlyness that were required by the initial objectives of this project.

---

# 4. Design Considerations

To appreciate the narrow design limits in which this project has to fall, a brief discussion of the basic architecture and minimum system requirements of an 8051 micro-controller system will be discussed initially:

## 4.1. Basic Architecture and Memory Map of an 8051 Micro-Controller

The 8051-family of micro-controllers has been designed to be as versatile as possible. It can be used in many applications with the addition of the minimum of hardware. Instead of the normal address, data and control busses found in other micro-processor systems, the micro-controller does all its communication with the outside world through four 8-bit bidirectional ports. In some special applications, some port pins are sacrificed for specific functions. The general philosophy is that if a port pin is not used for its specific function, it must be available for general use. Built into the micro-controller are two general purpose timers or counters, a serial asynchronous communication port, two external interrupt sources and a limited amount of internal Random Access Memory. The internal data memory is limited to 128 bytes, and should more data space be required, up to 64K bytes of external data memory can be added.

Various options for the code or program memory storage are available in the micro-controller: It may be factory programmed memory, erasable programmable read only memory, one-time programmable memory, or no external code memory. The latter device is the most economical choice. If more code memory is required

for those devices with limited internal code memory, up to 64K bytes of external code memory can be added. With *internal* code memory, it may be possible to have an entire system consisting of a single IC. It will contain the code memory, some data memory, timers or counters, serial communication port, and thirty-two bidirectional port pins for interaction with the outside world. The ports are divided into four groups of eight pins each, "Port 0" to "Port 3".

Unlike 8086 systems, where the entire memory map is available and accessible for programs *and* data, a micro-controller uses a completely separate **Read Only** code memory map. An additional 64K bytes of external data memory can be added if required. The memory maps for an 8051 system, using external code and external data memory, are as follows:

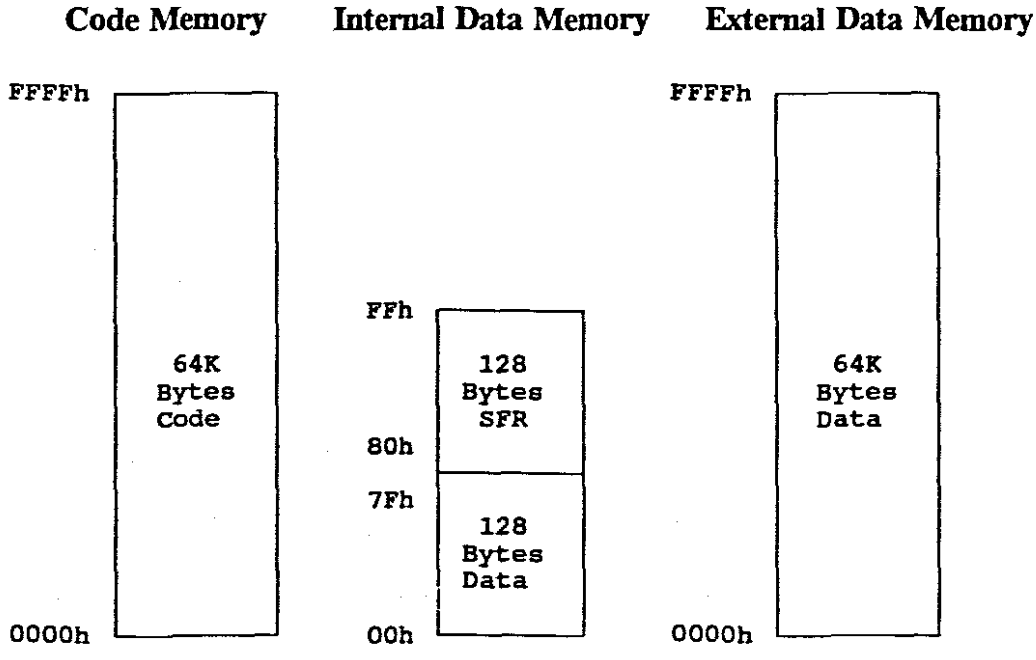



**Fig. 4.1. Memory Maps Available to the 8051 Micro-Controller**

**Note:** Addresses are shown in the hexadecimal notation.

## 4.2. Access to External Data

Two ports plus two additional control lines are sacrificed to enable immediate and high speed access to external memory. To address 64K bytes external addresses of 8 bits each, at least sixteen address and eight data lines are required. This is besides the read and write control lines. Port 2 is used for the higher eight bits of the sixteen bit address word. Port 0 is used for the low address byte and the data input/output in a time multiplexed system.

To access external code or data, the high address is placed on Port 2 simultaneously with the low address byte on Port 0. An **Address Latch Enable (ALE)** control line is taken high, and the external hardware must provide for an 8-bit address latch to capture the low address and keep it stable during the rest of the operation.

To read data from the code memory, the **Program Store Enable (PSEN)**, must enable the data from the Code Memory, and place it on the Port 0 pins to be read. The **ALE** and **PSEN** control lines are separate controls that are not included and are not part of the dual function port pins.

To read data from the External Data Memory, one of the dual function port pins of Port 3, the **Read Control**, must be used to enable the data output from the memory IC. Similarly, another dual function port pins of Port 3, the **Write Control**, must be used to write the data from the processor on Port 0 to the memory IC. The basic external memory address access system required by the 80C31 micro-controller is shown on the next page:

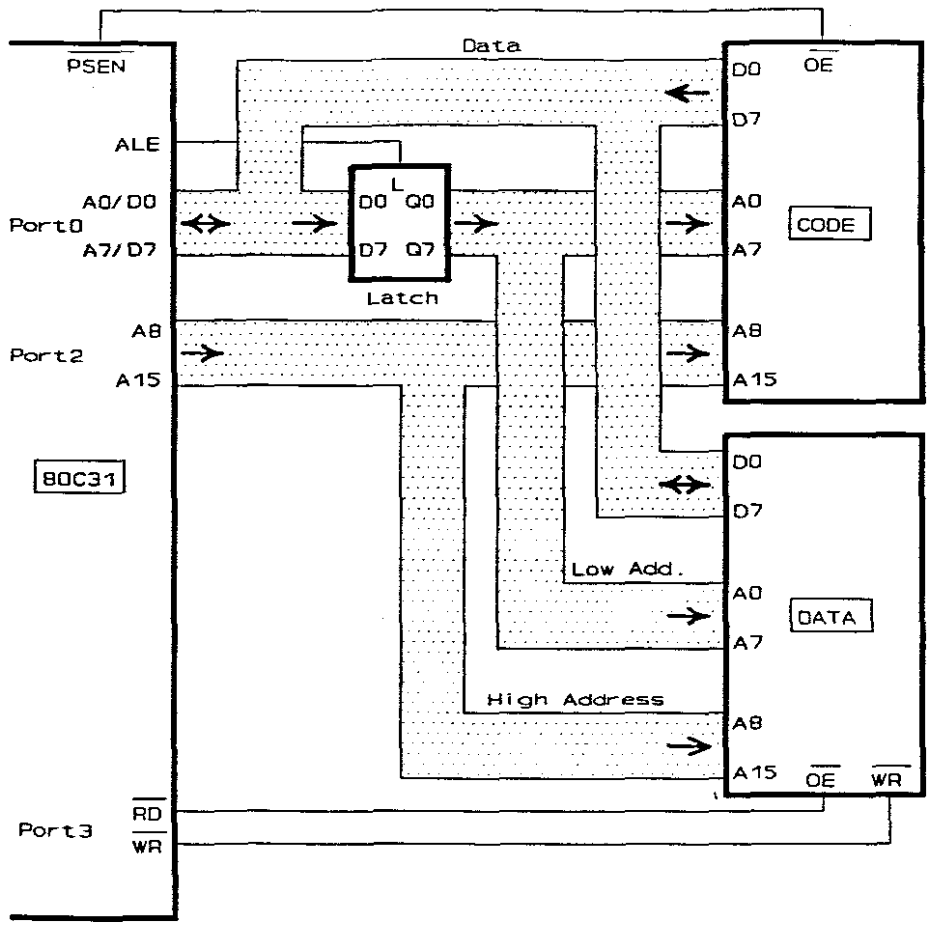



**Fig. 4.2. Access to External Memory**

Because a micro-controller system in a normal user application is supposed to be used in a dedicated system with pre-programmable EPROM code, no provision was made in the design of the micro-controller to write to and to alter the programming code bytes through software. The code memory can only be read, and no instructions or control lines are available for writing to the code memory. Apart from the normal *program instruction fetch and execute* access to the code memory, the facility to read a code byte directly as data, is also available. This facility allows access to code memory-based look-up tables and other fixed data. This is, however, also a *read only* function.

To activate the correct control lines for accessing external code or data memory, or to access internal data memory, different types of instructions are used for each segment. These are "MOVC" for code, "MOVX" for external data, and "MOV" for internal data.

## 4.3. The Memory Map Organisation for this Project

In an application development environment, where the software of a program may be altered and modified many times over, erasing and re-programming of an EPROM is a tedious and time-consuming process. If an eraser and programmer are not immediately available, one could at best do one or maybe two modifications per day. To enable any software modifications to be loaded directly into the code memory of a system, normal RAM can be used. To simulate the non-volatility of an EPROM, the RAM chip must be supplied with a *smart socket.* This contains a Lithium battery and a circuit to switch in the battery backup to the RAM when the power to the chip is removed.

New software will be loaded from the PC through the serial port in the INTEL Hex format. An onboard Operating System will use the micro-controller to write the new code into the code memory segment. To enable this *writing* to the code memory, for which no provision was made in the software or the hardware of the micro-controller chip, a special hardware decoding manipulation was devised: Two 32K x 8-bit RAM chips were used. Both were supplied with *smart sockets* for data retention when the power was removed.

For normal execution of a program, the 32K bytes of Code Memory for the system will be contained in a single IC. It will be decoded and read from the lower half of the available 64K bytes for the code segment. At the same time, 32K bytes of External Data RAM, contained in a second IC, will be available for reading and writing in the lower half of the available 64K bytes of the External Data segment.

The Code Memory chip must also be decoded to be accessed as read or write data memory in the *top* 32K bytes of the *External Data* segment. A byte written to the first memory address of the top half of the data segment at address 8000h, can then be accessed through either a read and write instructions for the data segment. It can also be read as program step number 0000h during the execution of a program, or accessed through direct code access using the "MOVC" command. The same chip therefore occupies both the lower half of the code segment, and the upper half of the data segment.

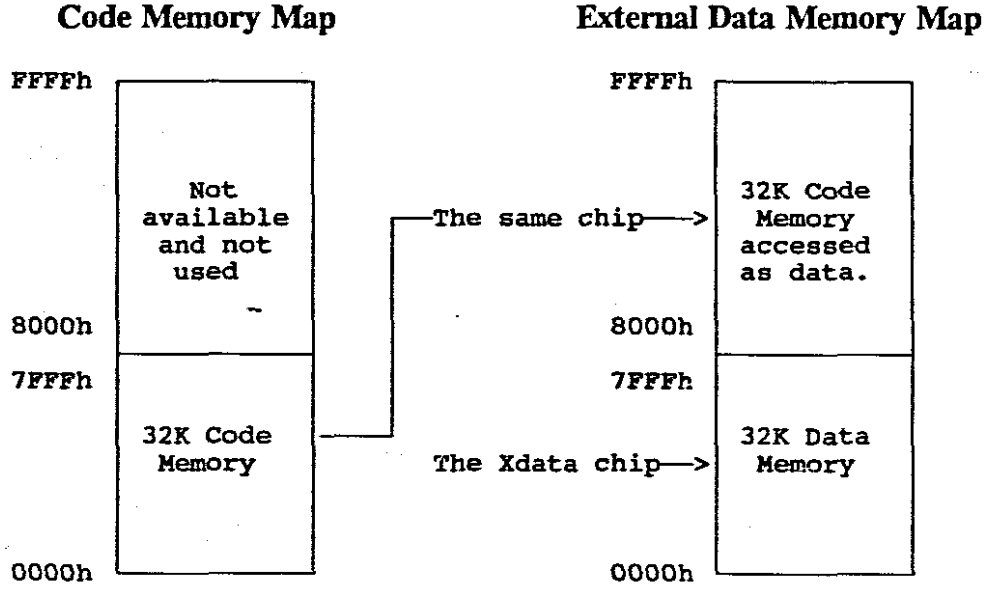



**Fig. 4.3. Dual Mapping of the Code Memory IC**

## 4.4. Interrupt Code Memory Segment Swapping and Decoding

The reset and interrupt starting addresses for an 8051 system are located at the start of the code segment at the following address locations:

0000h Reset Address

0003h External Interrupt 0

000Bh Timer 0 Roll-over Interrupt

0013h External Interrupt 1

001Bh Timer 1 Roll-over Interrupt

0023h Serial Port Receive and Transmit Interrupts

The onboard Operating System will use various interrupts to wake the system out of the low power consuming Idle mode, but when a user program is executed, it will require the same addresses for user interrupt subroutines. To make all interrupts available for a user program, the Operating System must be completely transparent while running a user program. There must be very little or no restrictions on the normal usage of the interrupt addresses for programming or the operation of a user program. To achieve this, some physical hardware address swapping had to be done to be able to read the correct code from the correct addresses in either the user program *Execution Mode*, or while the Operating System takes over control. In principle, two blocks of code memory, addresses 0000h to 0007h, or addresses 0000h to 002Fh will be swapped with their equivalent blocks at the lower end of the Operating System, which resides at code addresses 5000h to 6FFFh. The exact detail will be explained later, but the basic principle of operation is as follows:

After a system reset or when the system is controlled by the Operating System, the reset and interrupt subroutine starting addresses that are normally located at addresses 5000h to 502Fh, are decoded to appear at the lower end of the code memory map between addresses 0000h and 002Fh. Just before a user program is run, a bit in the hardware is set to swap back the normal lower portion of the code memory. This will contain the user defined reset and interrupt vectors. These can then be read and executed at their standard address locations.

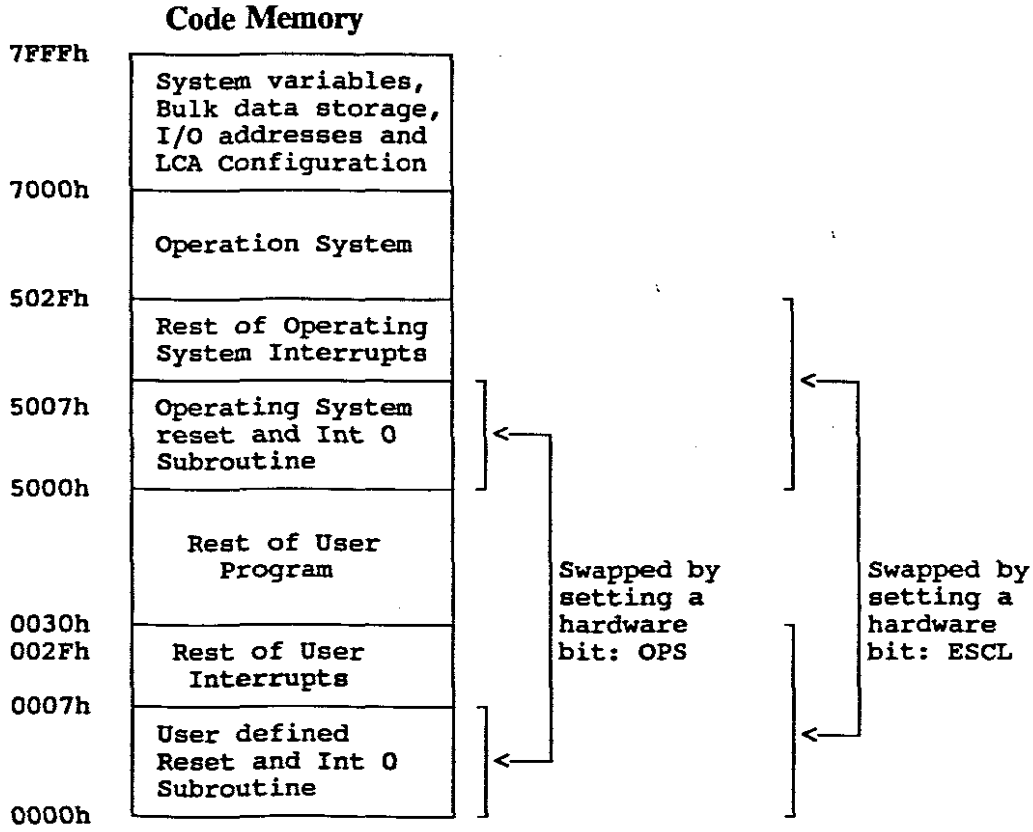



**Fig. 4.4. Reset and Interrupt Vector Swapping**

## 4.5. Hardware Address Decoding and Other I/O Circuits

The additional access of the code and external data memory, together with the address swapping of the reset and address vectors, imply a rather complex address

decoding circuit. It would need quite a few conventional logic circuits to realise such a system. Therefore, to also include all the other logic circuitry, such as the address latch, the rasterscan keyboard decoder, the address decoding of the display module and the additional user I/O select lines, a single Field Programmable Logic Cell Array, or LCA, was implemented. It incorporated all the logic circuits required by the system, apart from the two 32K byte RAMs, the micro-controller itself, and the display module. The only other two ICs on the board are the RS232 logic level converter, a MAX232, and an opto-isolator used for the software controlled auto-off circuits.

## 4.6. The LCA Configuration

The internal connections, or **configuration** of the field programmable LCA, are completely made up of volatile memory bits. It must be re-configured every time power is applied to the chip. On power up, the chip can be set to one of various modes to load the configuration from some source. The internal circuits will then be interconnected to emulate all the hardware circuitry it was designed to replace. Such a system enables changes to most of the hardware to be as simple as changes to the software of a normal program. This configuration of the LCA, not being made up of fixed physical hardware circuits, nor being software that can be changed while the system is in operation, but are fixed only for the duration of a current operating session, is referred to as the **firmware.**

The method of loading the LCA configuration depends on the logic levels set to three control pins of the LCA on power up. The method chosen for this system

was the so-called **Master Parallel Mode** (from high addresses to low addresses). On power up, the LCA will place address FFFFh on its configuration address lines. By means of a "Data Enable Control" line, it will read the configuration data in on its data pins. While the configuration is in progress, the micro-controller is kept at reset with all its port pins at high impedance. The pins that would be used for the low address latch during normal operation were chosen to be the same address and data lines used for the configuration uploading. Similarly, the pin that would enable the reading of the code RAM, was also chosen to be the "Configuration Data Enable" control line. Ignoring the address line "A15", the LCA will therefore start to load the configuration of approximately 1.5K bytes from the code memory IC, from address 7FFFh, downwards.

## 4.7. The Initial *Seeding* Process

The hardware connections between the LCA and the memory chips were also used for the so-called *seeding* process. This process places the initial *intelligence*, consisting of the Operating System and the normal configuration of the LCA, into the Code RAM of the system. Theoretically, this should only be required after the board has been assembled with unprogrammed memory chips, or if any modifications to the firmware or the Operating System have been done.

For the seeding process, the changing of a hardware jumper on the board chooses another configuration mode for the LCA, the **Serial Slave Mode**. In this mode, the configuration must be loaded in from some external source into the LCA through two pins, a "Clock" and a "Data-In" pin. This external source was chosen

to be two data pins from the parallel printer port of a PC. After power up, the seeding process will start to configure the LCA as a **Serial-In-Parallel-Out Shift Register**, and an **Address Counter**. After shifting eight data bits into the shift register, the data will be written in parallel to the code memory chip, and the address counter will be incremented or decremented by one. Two miniature slide switches on the board, that are normally used to select the Running Modes of the system, were also used for the seeding process. The switches were configured to reset the address counter to either 5000h for the Operating System, or to FFFFh for the normal configuration data, and to select either the up or the down counting of the address counter.

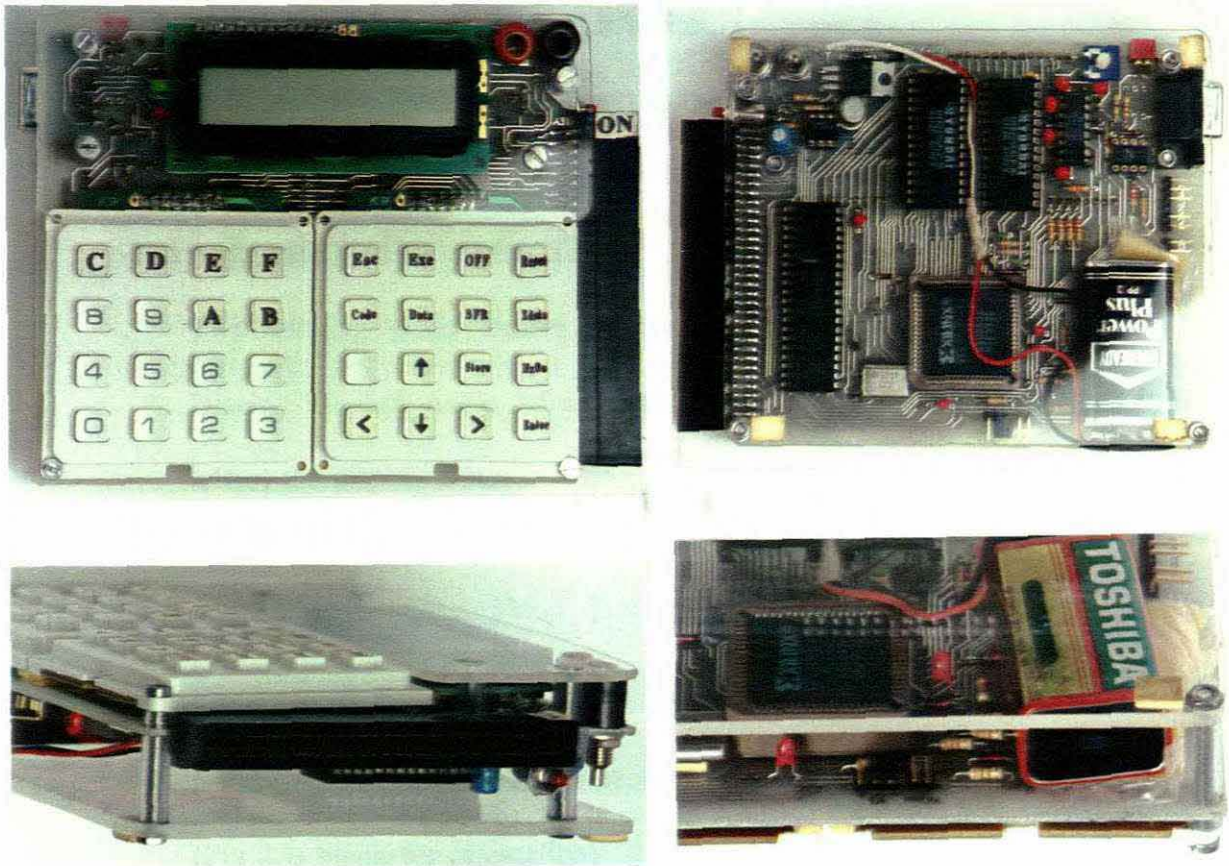
Three sets of data are therefore downloaded from the PC to the LCA through the parallel port for the seeding process: First the LCA is configured to be a Shift Register and an Address Counter. Then the data required to configure the LCA for normal operation is serially shifted into the shift register, and loaded into the top end of the code memory chip. Finally, the Operating System is serially shifted in and loaded into addresses 5000h to 6FFFh. The *smart socket* maintains the stored configuration and the Operating System data in the code memory chip.

The LCA, now configured in the **Master Parallel Mode**, will then be instructed to re-configure. It will read the configuration data from the code memory chip and configure itself to be the address latch, decoders and keyboard scanners required for normal operation. The micro-controller reset pin is then released, and normal operation of the system can continue.

---

# 5. Product Description

## 5.1. General Layout and Component Identification




The following user controls and intelligence exchange devices can be identified on the pictures of the product:

**From Above:**

* 16-Character by 1-line LCD Display Module.
* 16-Key hexadecimal keyboard.
* 16-Key Operating System Function keyboard.
* A Green LED - the ON indicator.
* A Red LED - the Busy Configuring Indicator.
* External Power Source Sockets. Red positive, Black negative.

**From the Right-Hand Side:**

* 62-Pin extension edge connector socket.
* ON/Configure Push Button - next to the 62-pin extension socket

**Through the Opening on the Top:**

* Two Running Mode Control slide switches.
* Display Viewing Angle Adjustment Control potentiometer - below and to the left of the display unit.

**From the Left-Hand Side:**

* 9-Pin female D-connector for the RS232 link with a PC.
* 6-Pin single-in-line header connector for "seeding" - next to the D-connector. (Pin 2 is blank for orientation)
* **Seed** Select Jumper - next to the 6-pin header.
* A 9 volt battery - secured on the component side of the PCB.

**Through the Opening on the Bottom End:**

* Serial PROM Enable jumper (two pins) - In the middle of the lower end on the component side.
* **EA** pin of the micro-controller (3 pins) - next to the serial PROM jumper.

**Through the Back Cover:**

From the rear end, the 8-pin DIL serial PROM socket for alternate configuration loading, the 40-pin micro-controller, the two 28-pin smart socketed Code and Data RAMs, the 44-pin PLCC LCA chip, the MAX232 voltage level converter, and the 8-pin opto-isolator can be identified. A voltage regulator in the TO-220 package makes the system independent of battery and external supply voltage variations.

## 5.2. Communication Between the User and the Operating System

Most of the communication and interaction between the Operating System and the user are through the 16-character by 1-line LCD Display Module and the thirty-two keys on the two key pads. The only other controls that may be used intermittently, are the 'ON/Configure' button to switch on the power to the board, and the two miniature Running Mode Select slide switches. These switches will select between using the Operating System, or running a user program in the *Single Step*, the *Break Point* or the real time *Execute Modes*.

The layout of the keyboard is as follows:

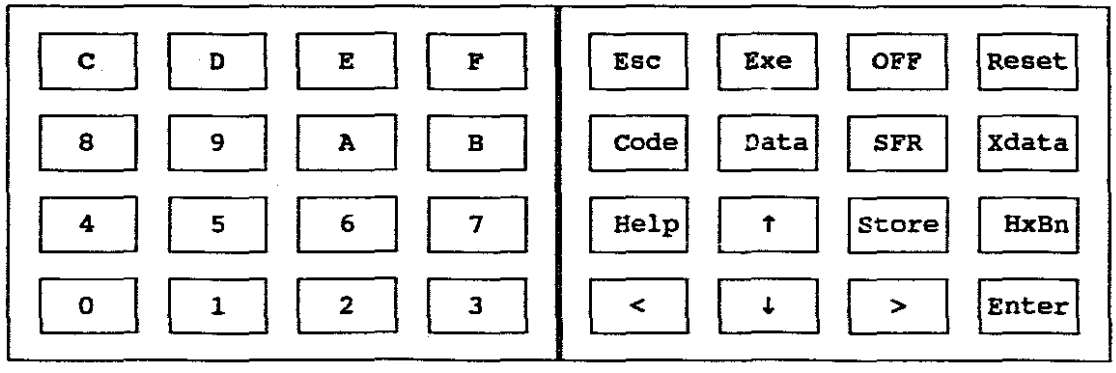



**Fig.5.2. The Layout of the Key Pads**

The left-hand key pad contains the normal hexadecimal numbers to specify addresses, numbers or data, when requested by the Operating System.

The right-hand key pad contains the special function keys. There are four cursor and menu direction keys for '<' (left), '>' (right), '↑' (up) or '↓' (down) selection, and an 'Enter' key to select the current option or to start the action. On the second line from the bottom, buttons will supply on-screen help on the current menu selection, a quick "Store Internal Data and Current System Variables" button, and a button that will convert and display the current data in either hexadecimal or binary digits.

The Binary or Hexadecimal Display Modes are very useful while in one of the "Memory Segment Inspection and Editing" menu options. A byte such as the **Program Status Word**, of which the bit settings have no relation to each other, would normally be inspected in the binary notation. The contents of, for example, an 8-bit counter, would rather be inspected in the hexadecimal notation.

The third line of buttons from the bottom is used as a quick way to select one of the four memory segments for inspection and editing. They are the 'Code', 'Data', 'SFR' and 'Xdata' select buttons.

The top line of buttons contains the 'Escape' key, which will move the control backwards out of the sub-menu selections. It is also used to create a break out of one of the execution modes of a user program and return control to the Operating System. The 'OFF' button automatically starts a "Store Internal Data and System

Variables" sequence, and switch the power off through software control. The same sequence is followed for the three minute auto-off facility. No hardware 'ON/OFF' switch as such is available to remove the power for the system. Finally there is a system 'Reset' button, which will interrupt any current action and reset the system.

## 5.3. An Overview of the Modes of the Operating System

The Operating System of the board contains sixteen different operating modes. These modes are selected on the display module through push button control and a menu driven system. Most of the menu selections also have sub-menus, and displayed messages will lead the user to enter the correct data when required. A user can move around freely between the menus and sub-menus, without fear of corrupting important data by accident. Before such corruption can occur, the system will normally ask the user to confirm his choice, or to press the 'Escape' key to cancel the selection and move back to the previous menu.

A brief description of each of the sixteen Main Menu Options is included below. More in depth discussions for each menu option are included in the Operating Procedures and the Software Description. The Main Menu options are:

| Option | Mode | Description |
|---|---|---|
| 0 | Main Menu | Starting point for the selection of the various other modes or options. |

| Option | Mode | Description |
|---|---|---|
| 1 | Edit Mem Segment | Inspect and editing of any one of the Code, Data, SFR or Xdata memory segments. |
| 2 | Edit Pre-Int Reg | Inspect and Edit the contents of the SFR and internal data byte values before a Single Step, Break Point or Escape break of a user program. |
| 3 | Store IData&SysV | Store the Internal Data and the System Variables, to enable a restart of a program from the same point and register values at a later stage. |
| 4 | Restore Data&SyV | Restore the Internal Data and the System Variables, to restart from the same point and register values stored previously. |
| 5 | Exe User Program | Execute a user program located at default address 0000H, with only 'Escape' key breaks. |
| 6 | Single Step Run | Execute only one instruction of a user program at a time, and return to the Operating System for memory inspection and editing. |

| Option | Mode | Description |
|---|---|---|
| 7 | Run with BrkPnts | Set up a break point table and execute a user program until the Program Counter matches one of the break points, before returning to the Operating System. |
| 8 | Clear Idata Mem. | Clear the internal data memory. |
| 9 | Clear User Prog. | Clear the user program area. |
| A | Clr BrkPnt Table | Clear the break point table. |
| B | RS232 Mem Load | Load data from a PC into the Code memory through the RS232 link. |
| C | RS232 Mem Dump. | Dump external data or code to a PC for storage on disk through the RS232 link. |
| D | Xdata Block Move | Move a block of external data. (All the other memory segments are also contained in the external data memory map.) |
| E | Prot/Unprot Mem. | Set or clear the software protection of the Operating System and the LCA configuration. |
| F | Save IMem + OFF | Save the entire internal memory and SFR values and switch off. |

## 5.4. The On-screen Help Facility

On-screen "help" facilities are available for the main menu and for each sub-menu. By pressing the 'Help' key, a few lines of help, describing the main features of the menu option, and a list of the available action keys for that selection, will be displayed.

---

# 6. Product Specification

## 6.1. General Specifications

* Physical Dimensions: W x L x H: 150 mm x 114 mm x 34 mm

* Power supplies: On-board 9 volt battery or a 7.5V to 35V External Supply

* Power dissipation: 22 ma @ 5 volt (after the voltage regulator)

* Micro-Controllers: *INTEL:* 8051, 8031, 8051AH, 8031AH, 8751H, 8751AH, 8052AH, 8032AH, 8752BH, 80C51BH, 80C31BH, 87C51, 80C52, 80C32, 83C51FA, 80C51FA, 87C51FA, 83C51FB, 80C51FB, 87C51FB.

  *DALLAS:* DS5000, DS5000T.

  *PHILIPS:* 87C751, 80C562.

* Display Unit: 16-character x 1-line LCD Display Module

* Keyboard: 16 hexadecimal keys and 16 Special Function keys

* Inputs and Outputs: Two Banana Sockets (External Power)

  9-Pin Female D-Connector (RS232 link to PC),

  5-Pin Header connector (Parallel PC Port),

  62-Pin IBM motherboard type edge connector socket.

  (connections to a user board)

* Hardware Controls: 2-Pole Running Mode Select slide switch,

  "Seeding Enable" jumper,

  "Serial PROM as the Configuration Source" jumper,

  "External/Internal Code select" jumper.

## 6.2. Computer Software Requirements and Data Files:

The SEED.EXE seeding driver, which requires the following hex format files:

* The SHIFTREG.MSC seeding Shift Register Configuration,
* The DECODER.MCS Configuration for normal operation
* The OPSYS.HEX Operating System.

The DOS COPY command, Version 3.3 or higher.

## 6.3. The RS232 Serial Link:

Baud Rates: 300, 600, 1200, 2400, 4800, 9600

Data Protocol: 8 data bits, No Parity, 1 Stop Bit

Connector on PC: Standard 25-Pin D-Connector (Male)

## 6.4. The 64K Byte Code Memory Map: (Read Access through PSEN)

| | |
|---|---|
| 0000 - 4FFFh | Available for User Programs (20K bytes) |
| 5000 - 6FFFh | Operating System (8K bytes) |
| 7000 - 7FFFh | Not available as code (4K bytes): System Variables, I/O Mapping and LCA Configuration. |
| 8000 - FFFFh | Code Addresses does not exist. |

## 6.5. The 64K Byte External Data Memory Map: (Access by RD and WR)

| | |
|---|---|
| 0000 - 7FFFh | External Data - 32K bytes available to the user. |
| 8000 - CFFFh | User Programs (20K bytes) - Xdata mirror of the Code. |
| D000 - EFFFh | Operating System (8K bytes) - write protected. |

| | |
|---|---|
| F000 - F2FFh | System Variables, Internal Data and SFR store. (See the appendix for a complete list and locations.) |
| F300 - F3FFh | As one WRITE byte to the Control Register, CREG, in the LCA, overlaid by a READ/WRITE memory byte to enable the reading of the current settings. |
| F400 - F4FFh | External Chip Select, CS0. (256 bytes) |
| F500 - F5FFh | External Chip Select, CS1. (256 bytes) |
| F600h | Display Module Command Instruction WRITE address. |
| F601h | Display Module Data WRITE address. |
| F602h | Display Module Status READ address. |
| F603h | Display Module Data READ address. |
| F604 - F6FFh | Display Module mirror image addresses. |
| F700 - F7FFh | One READ byte for the Keyboard and the current state of the Running Mode Select Switches. |
| F800 - FFFFh | LCA Configuration (2K bytes) - write protected. |

---

# 7. Operating Modes

Four types of *Running Modes* are available, and each mode is selected by the setting of the two **Running Mode Select** slide switches. These switch setting will be normally selected before power up, a hardware reset, or when prompted by the system to do so. It may also be selected while running in any one of the other modes. The four *Running Modes* and switch settings are as follows:

## 7.1. The Operating System Mode

Inspecting and editing of memory locations. The loading and dumping of user programs from and to a PC. Storing and retrieving of the 8051 internal data and current system variables. Memory block clearing and moving, etc.

Sw1 = OFF, Sw2 = OFF

## 7.2. The Single Step Mode

Execute only one instruction of a user program at a time, and jump back into the Operating System. All the system variables, SFR contents and the internal data of the micro-controller that would be corrupted by the Operating System, are stored in memory. It can be inspected and edited directly by the Operating System. Before the next user instruction is executed, all these values, including the altered ones, will be loaded back into the micro-controller:

Sw1 = ON, Sw2 = OFF.

## 7.3. The Break Point Mode

Before executing any instruction of a user program, the address of the first byte of the instruction is compared with a list of thirty-two possible addresses, previously defined by the user in a Break Point Table. Only if a match is found, the control will jump back to the Operating System. While running in the Break Point Mode, pressing the 'Escape' key will cause a break after the current instruction, even if no address match has been found:

Sw1 = OFF, Sw2 = ON.

## 7.4. The Execute Mode

The user program will be executed at normal speed, and will only be interrupted when the 'Escape' key is pressed. A break will also occur if a running mode selection switch is switched over to select another mode, while a user program is executing:

Sw1 = ON, Sw2 = ON.

---

# 8. Operating Procedures

For this description it will be assumed that the system has been *seeded* already, and the LCA configuration and Operating System are stored and maintained by the *smart socket* in the code memory chip. The procedure to *seed* the system will be described in detail at the end of this section.

## 8.1. Setting Up Procedure

An external supply can be connected to the two banana sockets on the board, if available. The applied voltage can be any value between 7.5 volt and 35 volt, but a good average of 10 volt is recommended. The circuit for the power supply was so designed that the power to the board will be taken from the higher voltage of either the external supply or the onboard battery. No provision has to be made to prevent the external supply to charge and damage the onboard battery. The changeover between the power from the battery or an external supply can even take place while the system is in operation, by just applying or setting the external voltage higher or lower that the 9 volt of the onboard battery.

The *seeding* connection cable to a PC must not be connected to the board, the SEEDING jumper must be open, the SERIAL PROM jumper must be open, and the central EA pin or the "External Memory Enable" jumper must be connected to the ground (-ve). The serial load RS232 cable from a PC can either be connected to the board or not.

The Running Mode Selection slide switches must be set to the desired mode, but unless a user program has already been loaded in before, only the **Operating System** selection (both switches OFF), will be appropriate.

The user program must be available on disk in the INTEL Hex format. This program will be downloaded into the board through the RS232 serial port, using the standard DOS **COPY** command.

## 8.2. Power Up and LCA Initialization

By pressing the 'ON/Configure' button, power will be supplied to the board, and the **Green** LED will light up to show that the power is available. Initially, the **Red** LED will also light-up briefly to show that the LCA is busy configuring. When the red LED stays on, it is an indication that the configuration data got corrupted, and that the board may have to be *seeded* again. This should not happen during the normal run of events, but if the onboard battery is starting to go flat, the supply voltage may not be sufficient to do a proper LCA configuration. Once an adequate supply voltage has been restored, proper configuration should take place without having to re-seed the board.

Assuming the Running Mode Selection switches have been set to select the Operating System, a brief "Cape Technikon" logo and a "$\mu$C Trainer V:4.3" message will be displayed on the LCD display module, before the system will go to the top of the main menu, showed by the displayed message: "Main Menu(↓,Ent)". The display screen of the LCD module is driven by a time-

multiplexed process, which limits the view angle of the screen to within specific margins. The view angle adjustment potentiometer is accessible through the top opening of the board, and can be used to adjust the view angle to suit the user.

The messages of the different menu options available from the main menu in the shortened 16-character abbreviation, as well as the full name, are shown with the mode numbers, below:

| Option | Mode | Full Name/Description |
|---|---|---|
| 0 | "Main Menu(↓,Ent)" | Top of the Main Menu |
| 1 | "Edit Mem Segment" | Edit a Memory Segment Byte |
| 2 | "Edit Pre-Int Reg" | Edit Registers before the Break |
| 3 | "Store IData&SysV" | Store Internal Data and System Variables |
| 4 | "Restore Data&SyV" | Restore Internal Data and Syst. Variables |
| 5 | "Exe User Program" | Execute a User Program |
| 6 | "Single Step Run " | Execute One User Instruction at a Time |
| 7 | "Run with BrkPnts" | Execute until a Break Point is reached |
| 8 | "Clear Idata Mem." | Clear the Internal Data Memory |
| 9 | "Clear User Prog." | Clear the User Program area |
| A | "Clr BrkPnt Table" | Clear the Break Point Table |
| B | "RS232 Mem Load.." | Load data from a PC |
| C | "RS232 Mem Dump.." | Dump Data to a PC |
| D | "Xdata Block Move" | Move a Block of External Data |
| E | "Prot/Unprot Mem." | Set or Clear the Software Protection |
| F | "Save IMem + OFF " | Save Internal Memory and Power Off |

## 8.3. The Help Menus

A few lines of on-screen help are available for all the major modes, giving a brief description of what the option does, and also the appropriate action that can be expected from the valid keys in that mode. The first line of each help menu is a help line on using the Help Menu itself, and will display the message "Next=↓,Esc=Exit". The available lines for a Help Menu can be stepped through by either pressing the '↓', the 'Help' or 'Enter' keys, or one can step backwards by pressing the '↑' key. The end of any help file will be shown by the message "End of Help.." and it will roll over back to the first line. The 'Esc' key will return the user to the mode from where the help was requested.

**Active Function Key Summary:** (In the Help Mode)

| | |
|---|---|
| ↓ or 'Ent' or 'Help' | - To the next help line. |
| ↑ | - To the previous help line. |
| Esc | - Back to the previous menu. |

## 8.4. The Main Menu and Sub-Menu Selection

The Main Menu can be considered to be Mode 0, and contain the headings of the sixteen available modes. The top of the main menu displays the message "Main Menu(↓,Ent)", showing that the rest of the menu selections will be displayed by stepping down through the selections using the '↓' key. The 'Enter' key will select the current displayed option. The '↑' key will step the options backwards, rolling over from Mode '0' to 'F'. Once the hexadecimal number allocation of specific options is better known, a required option can be selected quickly by just

pressing the corresponding hexadecimal number on the key pad. This feature provides a fast way of getting to a specific selection, without having to step through all the options one by one.

From the main menu, it is also possible to move directly to any of the Memory Segment Edit options through the 'Code', 'Data', 'SFR' and 'Xdata' keys. The "Store Internal Data and System Variables" mode can also be selected directly through the 'Store' key and the "Save Data and OFF" mode option can be selected through the 'OFF' key.

The 'Help' key will display a few lines of help on the Main Menu, and also a brief explanation on the action of the appropriate keys, valid for this mode.

If the control was given over to the Operating System by breaking out of the execution of a user program, the next user program step can immediately be selected by pressing the 'Exe' key.

The 'Esc' key will return the mode selection and the displayed message back to the top of the Main Menu.

**Active Function Key Summary:** (Main Menu)

| | |
|---|---|
| ↓ | Next mode heading. |
| ↑ | Previous mode heading. |
| Ent | Select currently displayed mode. |
| Help | Main Menu Help facility. |
| Store | Store internal data and system variables - Mode 3. |

| | |
|---|---|
| Code | Inspect and edit the code segment - Mode 1. |
| Data | Inspect and edit the data segment - Mode 1. |
| SFR | Inspect and edit the SFRs - Mode 1. |
| XData | Inspect and edit the external data segment - Mode 1. |
| Esc | Escape to the top of the main menu. |
| Exe | Execute the next user program step(s) - only when one of the Running Modes is active. |
| OFF | Store internal data & system variables and switch off - Mode F. |
| '0'-'F' | Quick selection of modes 0 to F. |

## 8.5. Mode 1: Edit Memory Segment Mode

This mode can be used to inspect and edit or alter any of the data bytes in any of the available memory segments, such as the Code, Internal and External Data as well as the current SFR values. Some Code memory bytes are protected against accidental corruption, such as the LCA configuration and the Operating System. This protection can however be disabled or re-enabled through the menu option 'E', but should be used with care. Some bytes do not exist physically, and will normally be shown to have the same value as the low address byte. It will show no reaction if an attempt is made to change the value.

Some values of the SFR and the first 24 bytes of the internal data shown in this mode, will reflect the current values used by the Operating System, and changing these values might lead to an unexpected reaction. When this happens, the 'Reset' key can be used to re-initialise the system back to the Main Menu.

Mode 1 is entered from the main menu by selecting one of the four memory segments select keys on the keypad: 'Code', 'Data', 'SFR' or 'Xdata'. The code segment can also be selected when 'Enter' is pressed while the message "Edit Mem Segm" is displayed, or through the *quick selection* numeric key '1'.

The display module will show the selected memory segment by the messages: "Code Add: ", "Data Add: ", " SFR Add: " or "Xdat Add: ", with the cursor blinking on the first digit of the hexadecimal address to be entered.

The format for the display in the hexadecimal notation consists of a four-digit hexadecimal address for the Code and the External Data, or a two-digit address for the SFR and Internal Data addresses. The data of the selected address follows as a two hexadecimal digit value.

When the address digits are entered through the numeric keys '0' to 'F', the values will show on the screen and the cursor will move one digit to the right automatically. After the last address digit has been entered, the appropriate data will be read from the address and displayed on the screen. To correct a digit, the cursor can be moved left or right to the appropriate digit through the '<' and '>' keys, and the correct value keyed in to replace the existing value. Moving the cursor to the right across undefined address digits, or pressing the 'Enter' key before the complete address has been entered, will fill the rest of the digits with zeros before displaying both the address and the specified data.

With the cursor on any of the two right-hand digits, data values can be inspected and altered. When changing a data digit, the new value will first be written to the

appropriated memory segment, and then read back before it is displayed. This will confirm that the address exits, are not write protected, and that the keyed-in data has been stored properly.

An address can be changed by just moving the cursor to the appropriate digit and entering the new address digit value. It is also possible to increment or decrement the current address through the 'Up' or 'Down' keys. The data for the new address will be displayed immediately.

Another memory segment can be selected by pressing the appropriate Memory Segment Selection key, which will move the control back to the address entry stage of the selected segment.

The data of a selected address can also be displayed in the *binary* notation. This format will show the segment as a single letter, e.g. 'C:', 'D:', 'R:' or 'X:', and the address in the normal hexadecimal mode, but the data byte will be displayed as eight binary digits, e.g.: "C:1234 01101001". This display mode can be selected by the 'HxBn' key, which will toggle between the hexadecimal and the binary notation modes.

The data in the binary mode can be altered individually through moving the cursor to the appropriate digit, and entering a '1' or a '0'. The other number keys will replace the entire high or the low nibble (4 bits) below the cursor.

The 'Help' key will display a brief description of this mode, and also a summary of the control keys valid for this mode.

**Active Function Key Summary:** (Edit Memory Segment)

| | |
|---|---|
| < | Move the display cursor one position to the left. |
| ↓ | Increment the current address. |
| > | Move the display cursor one position to the right. |
| Help | Call the help screen facilities. |
| ↑ | Decrement the current address. |
| HxBn | Toggle between hexadecimal and binary formats. |
| Code | Select to inspect and edit the Code Memory Segment. |
| Data | Select to inspect and edit the Internal Data Memory. |
| SFR | Select to inspect the Special Function Registers. |
| Xdata | Select to inspect and edit the External Data Memory. |
| Esc | Escape back to the main menu. |
| Exe | Continue the execution of the user program after a Single Step, Break Point or Normal Execute break, or if one of these modes was selected through the switches. |
| '0'-'F' | Enter Address or Data digits. |

## 8.6. Mode 2: Edit Pre-Interrupt Registers

This option is only available when one of the user program running modes is active. When a user program is interrupted while in the Single Step, Break Point, or by the Escape key in the Execute mode, the program will enter the Operating System through this mode. Before the jump, all the current values of the SFR and internal Data that may be corrupted by the Operating System, will first be stored

in the memory. This mode makes these values available for inspection and editing.

Only register and data bytes used by the Operating System, are available and are displayed in this mode, the rest can be inspected and edited by the normal Edit Memory Segment option. The order of the Pre-Interrupt Registers and the first twenty-four internal data bytes displayed in this mode, is as follows:

| | |
|---|---|
| `Prog. Cntr:dd dd` | Program Counter (when the break occurred) |
| `PSW  Acc :dd dd` | Program Status Word and the Accumulator |
| `DPH  DPH :dd dd` | Data Pointer bytes DPH and DPL |
| `SP  B Reg:dd dd` | Stack Pointer and B Register |
| `IP  IE :dd dd` | Interrupt Priorities and Interrupt Enables |
| `TCON  TMOD:dd dd` | Timer Control and Timer Mode registers |
| `TH0  TL0 :dd dd` | Timer 0 high and low bytes |
| `TH1  TL1 :dd dd` | Timer 1 high and low bytes |
| `D:00=dd dd dd dd` | Internal Data: 00H to 03H |
| `D:04=dd dd dd dd` | Internal Data: 04H to 07H |
| `D:08=dd dd dd dd` | Internal Data: 08H to 0BH |
| `D:0C=dd dd dd dd` | Internal Data: 0CH to 0FH |
| `D:10=dd dd dd dd` | Internal Data: 10H to 13H |
| `D:14=dd dd dd dd` | Internal Data: 14H to 17H |

('dd' = two hexadecimal digits)

**Note:** The Program Counter will display the address of the next instruction that will be executed. If the SFR or internal data values are altered in this mode, the

new values will be reloaded back into the SFRs and the first 24 internal data bytes, before the next user program instruction is executed. By altering the program counter value in Mode 2, it will define a new returning point when the user program is continued.

To continue the user program, the Execute key will return to the user program to execute the next instruction(s) until the next break occurs. The restoring of all the previous SFR and internal data values back to their previous locations, makes the Operating System completely *transparent*. It does not reserve or prevent the use of any internal data bytes or Special Function Registers in a user program.

This mode is very useful in the debugging of a program. It allows one to inspect and alter the values of the SFRs and internal data bytes at any specific point, just before a next instruction will be executed.

**Active Function Key Summary:** (Edit Pre-Interrupt Registers)

| | |
|---|---|
| < | Move the display cursor one position to the left. |
| ↓ | Go to the next registers/bytes. |
| > | Move the display cursor one position to the right. |
| Ent | Go to the next registers/bytes. |
| Help | Call the help screen. |
| ↑ | Go to the previous registers/bytes. |
| Esc | Escape back to the main menu. |
| Exe | Continue the execution of the user program. |
| '0'-'F' | Enter Address or Data digits. |

## 8.7. Mode 3: Store Internal Data and System Variables

This mode enables one to store all the current values of the internal registers and data of the micro-controller, as well as the system variables and the pre-interrupt register values of Mode 2. This is useful when a bug in a program is creating a system crash that can only be recovered by a hardware reset. The mode allows one to restart at the same point when the settings were saved, somewhere before the crash occurred. After a hardware reset, all the stored values may be recovered from memory after the normal initialization, and restored back into the internal registers and data bytes through Mode 4.

**Active Function Keys:** (Store Internal Data and System Variables)

| | |
|---|---|
| Ent | Confirm the storing of the data. |
| Help | Call the help screen. |
| Esc | Escape back to the main menu - no storing. |

## 8.8. Mode 4: Restore Internal Data and System Variables

This mode enables one to restart the execution of a user program from a previous point stored by Mode 3, the "Save and OFF" option of Mode F, or just before the 3-minute auto-off facility. All the internal registers, the data, and the system variables will be restored to exactly the same values as they were during the previous store operation.

**Active Function Keys:** (Store Internal Data and System Variables)

| | |
|---|---|
| Ent | Confirm the restoring of the data. |
| Help | Call the help screen. |
| Esc | Escape back to the main menu - no restoring. |

## 8.9. Mode 5: Execute a User Program

This mode initiates the running of a user program at normal speed from within the Operating System. When selected, it will prompt the user to switch both the Running Mode Selection switches ON. Then the starting address can be selected. The default starting address for user programs is 0000H, but the option is available to start at any address between the limits of 0000H and 4FFFH, which is the memory area available for user programs.

The user program will be executed at normal speed, and can be interrupted any time by pressing the 'Escape' key. Control will then go over to the Operating System in the Edit Pre-Interrupt Registers Mode, displaying the code address of the next instruction that will be executed.

All the normal interrupt vectors are available to the user, with the following simple size and placement limitations to the interrupt subroutine:

The External Interrupt 0 routine must fit entirely into either code addresses 0003H to 0007H, or it must use a "Long Jump" to an address anywhere above 0030H. The other interrupt subroutines must either be limited to addresses 000BH to

002FH, or contain a "Long Jump" to any address above 0030H. They may not step over the 002FH to 0030H barrier normally, or through a relative or an absolute jump.

The reason for these limitations lies in the method used by the system to enable Single Step or Break Point operation. It uses interrupts extensively, but still facilitate the full implementation of all the user interrupts. This is done by swapping the interrupt starting addresses under certain conditions by hardware control, so that either the user-interrupt routines, or the Operating System interrupt routines are serviced. These conditions will be explained in detail under the software description section.

To continue the execution of the user program, the 'Exe' key can be pressed from within almost any one of the Operating System Modes. If the program does not resume, the 'Escape' key must first be pressed, followed by the 'Exe' key.

The system can also be set to start executing a user program after a hardware reset or at power up. It can be accomplished by switching both slide switches ON, and pressing the 'Reset' or the 'ON/Configure' buttons. After initialization of the display and the setting of some system variables, the user program will start to execute from the default address 0000H.

If a user program clears the **External Interrupt 0 Enable** bit, or the **Enable All** bit, the 'Esc' key cannot interrupt the operation until these bits are set again. A full summary on the guidelines on writing user programs for the system, is included later in this document.

**Active Function Keys:** (Execute a User Program)

| | |
|---|---|
| Ent | "Confirm the Switches were set" / "Run Program". |
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |
| Exe | Run the User Program. |
| '0'-'F' | Select a new start address. |

## 8.10. Mode 6: Execute a User Program in the Single Step Mode

For testing and debugging a user program, it is useful to be able to inspect all the registers and data bytes after every instruction. This will confirm that the program does what it is supposed to do. After the completion of each instruction in the Single Step Mode, the control will jump to the Operating System into the Edit Pre-Interrupt Registers Mode, displaying the code address of the next instruction that will be executed. Normal inspection of any memory location or the pre-interrupt registers, or any other function of the Operating System can also be selected.

To continue the execution of the user program, the 'Exe' key can be pressed from within almost any one of the Operating System modes. If the program does not resume, the 'Escape' key must first be pressed, followed by the 'Exe' key.

When entering this mode, the user is prompted to select the Single Step Running Mode by switching slide Switch 1 ON and Switch 2 OFF. Then the starting address can be selected. The default starting address is 0000h, but it may be

altered to any value within the address range available for user programs. The 'Escape' key will return control to the main menu.

The system can also be set to start executing a user program in the Single Step Running Mode after a hardware reset or at power up. This is achieved by first switching slide Switch 1 ON and Switch 2 OFF before the 'Reset' key or the 'ON/Configure' button is pressed. After initialization of the display and the setting of some system variables, the user program will start to execute in single steps from the default address of 0000H.

**Active Function Keys:** (Execute a User Program in single steps)

| | |
|---|---|
| Ent | "Confirm the Switches are set" / "Run the Program." |
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |
| Exe | Run the User Program. |
| '0'-'F' | Select a new start address. |

## 8.11. Mode 7: Execute a User Program in the Break Point Mode

This mode is similar to the Single Step Mode, except that the user can decide where breaks in the program must occur. A break is obtained through entering the address of the first byte of the instruction *just after the break* into a **Break Point Table.**

After the completion of each instruction, the address of the first byte of the next instruction will be compared with every entry in the break point table. Only if

a match is found will control be given over to the "Edit Pre-Interrupt Register" mode of the Operating System.

The displayed code address in Mode 2 will be that of the next user instruction that will be executed. Normal inspection of any memory location or the pre-interrupt registers, or any other function of the Operating System can also be done during this break.

Up to thirty-two break points can be specified, and they do not have to be in any specific order. If less than thirty-two break points are used, the list must end with an address value entry of 0000h.

To remove a break point from the table without influencing the rest of the break points, it must not be simply cleared to 0000h. The break point should be replaced by an address that does *not* fall on the first byte of an instruction. The address can also fall completely out of the address range of the program, so that it will never be reached, such as FFFFh.

The break point table is stored in a reserved space in the Code Memory IC. Therefore, no special instructions have to be included in the user program to cause a break. Such special instructions would move the relative positions of the programming instructions, that is undesirable. Being stored in a battery backed up memory, previous breakpoint settings will be retained during power down.

To continue the execution of the user program, the 'Exe' key can be pressed from within almost any one of the Operating System modes. If the program does not

resume, the 'Escape' key must first be pressed, followed by the 'Exe' key.

When entering this mode, the user is prompted to select the **Break Point Running** mode by switching slide Switch 1 OFF, and Switch 2 ON. The **Break Point Table** can then be inspected and altered. The thirty-two break points will be displayed from number 00H to 1FH, and the up and down arrows allows one to step forward or backwards through the table. To exit this editing mode, the 'Enter' key must be pressed. This will move on to the starting address selection. The default starting address of 0000h may be altered to any value within the address range for user programs. An 'Escape' key will return the main menu.

The system can also be set to start executing a user program in the Break Point Running Mode after a hardware reset or at power up. This can be achieved by switching slide Switch 1 OFF and Switch 2 ON, before pressing the 'Reset' key or the 'ON/Configure' button. After initialization of the display and the setting of some system variables, the user program will start to execute in the Break Point Mode from the default start address of 0000H.

**Active Function Keys:** (Execute a User Program with Break Points)

| | |
|---|---|
| Ent | "Confirm the Switches were set" / "Exit the Break Point Table" / "Run the Program." |
| < | Move the display cursor one position to the left. |
| ↓ | Increment to the next break point. |
| > | Move the display cursor one position to the right. |
| ↑ | Decrement to the previous break point. |

| | |
|---|---|
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |
| Exe | Run the User Program. |
| '0'-'F' | Select a new start address. |

## 8.12. Mode 8: Clear Internal Data Memory

This mode will write 00H to all the internal data memory locations from 00H to FFH, in the indirect addressing mode. In the 80C31, indirect addresses 80H to FFH do not exist. Provision has also been made for using an 80C32 as an alternate micro-controller on the board.

**Active Function Keys:** (Clear Internal Data Memory)

| | |
|---|---|
| Ent | Confirm the clearing of the data memory. |
| Help | Call the help screen. |
| Esc | Do not clear and escape back to the main menu. |

## 8.13. Mode 9: Clear the User Program Area

This mode will write 00H to all the user program code memory locations from 0000H to 4FFFH (20K Bytes).

**Active Function Keys:** (Clear the User Program Area)

| | |
|---|---|
| Ent | Confirm the clearing of the user program area. |
| Help | Call the help screen. |
| Esc | Do not clear and escape back to the main menu. |

## 8.14. Mode A: Clear the Break Point Table

This mode will write 0000H to all thirty-two available break point table locations.

**Active Function Keys:** (Clear the Break Point Table)

| | |
|---|---|
| Ent | Confirm the clearing of the break point table. |
| Help | Call the help screen. |
| Esc | Do not clear and escape back to the main menu. |

## 8.15. Mode B: Load a Program or Data through the RS232 Link

The serial data transmission protocol used, is 8 data bits with no parity, and the baud rate is selectable between the standard baud rates of 300, 600, 1200, 2400, 4800 and 9600 baud. The RS232 link uses the standard logic voltage levels for RS232 communication of -3 to -30 volt for a 'high', and +3 to +30 volt for a 'low'. These levels are generated and converted by the MAX232 IC before it is connected to the serial cable via a 9-pin female D-connector.

A link on the board connects the CTS (Clear to Send) handshaking line to the RTS (Request to Send) line. This generates an error on the PC if the board is not connected, but no handshaking lines are used by the system as such. Some PC communication systems also require a connection between the DTR (Data Terminal Ready) and the DSR (Data Set Ready) handshaking lines. This is taken care of in the 25-pin D-connector plugged into the RS232 serial port of the PC.

The default protocol setting for a PC is normally 7 data bits, even parity, 2 stop bits and a baud rate of 9600 baud. If a mouse driver was installed, it normally

changes to no parity, 8 data bits, and a baud rate of 1200 baud. The protocol of the PC may be altered by using the DOS external MODE command:

**MODE COM1 br,pr,da,sb,P**

where **br** is the baud rate from 110 to 9600,

**pr** is (O,E or N) for Odd, Even or No parity,

**da** is 7 or 8 for the number of data bits, and

**sb** is 1 or 2 for the number of stop bits.

Including **P** will cause continuous retries for timeout errors on the serial port.

A typical command will therefore be:

**C:\DOS>MODE COM1 9600,N,8,1,P**

The incoming serial data is expected to be in the INTEL Hex format, which is described in full in the appendix. It can be transferred from a pre-prepared INTEL Hex file through the serial port by the PC under DOS control, using the standard **COPY** command. A typical example is:

**C:\8051>COPY *filename*.HEX COM1**

No target starting address needs to be specified when this mode is selected, because the INTEL hex format contains the target address for each line. To compensate for the unique organization of this system to enable writing access to the code memory, the MSB of each address will be set before it is stored. The incoming code, specified to be at the normal code addresses, will therefore be loaded into the *top* 32K bytes of the external data memory map. This will then

be available as code at the *lower* 32K bytes of the code memory map.

A user can therefore prepare his program to start at the default starting address of 0000H, including any interrupt subroutines, and download it directly to the board. The most significant address bit will be set by the system to load the data from address 8000H, upwards. When, however, the program is executed, it will be read from the original code address of 0000H. The Operating System in the 8K bytes between 5000H and 6FFFH and the configuration data above address 7800H of the code memory, are write-protected. Any data specified for these addresses, will be lost.

The setting of the MSB of the address caters specifically for the downloading of the code of a user program. The system may also be used to download data into the lower 32K byte of the external data memory map. For this operation, some data manipulation is required: To load a block of data into the lower 32K byte external data memory, it must first be loaded into an unused section of the user program area, and then moved down by the Block Move Mode.

When entering the RS232 Load Mode, the Operating System will first request that the appropriate receiving baud rate be selected. Then it will display the message "RS232 Ready...", and go into a waiting state for incoming serial data. When data is received, a message will display "RS232 Loading..". The data will be stored at the appropriate address. A checkbyte will be accumulated up to the end of each line and reception will be ended if a checksum error occurs. The process will also stop after the INTEL hex format terminating line has been received, or

a key was pressed on one of the key pads. Messages on the screen will show if there was a "Checksum Error", if loading was "Aborted" by a key, or if reception was "Done" with no errors. The operation will return to the RS232 Memory Load heading of the Main Menu for additional loads.

The only place in the Operating System where the 3-minute auto-off timing does not occur, is while waiting for serial data to be received. If the user therefore wants the auto-off timing to be suspended, the system can be placed in this mode, and reactivated by pressing any key to abort the serial loading process.

**Active Function Keys:** (Baud rate selection for RS232 Memory Load)

| | |
|---|---|
| < or > | Move the cursor to select another baud rate. |
| Ent | "Select the Receive Baud Rate" / "Start Reception". |
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |

**Note:** Once reception was initiated, the pressing of any key will abort the reception mode.

## 8.16. Mode C: Dumping a Block of Data through the RS232 Link

When entering this mode, the transmission baud rate must first be selected. Then the starting address of the external data memory block and the number of bytes will be requested, before serial transmission will commence. The outgoing data will be converted into the INTEL Hex format. This will include a checksum, a carriage return character after each line, the terminating line, and an "End-of-

File" character, ASCII number 26 or 1Ah. This last character will stop reception by the PC and close the storage file on the disk.

The receiving PC must be prepared and set to the correct protocol and baud rate beforehand. The PC can receive and store the data directly on a disk file through the COPY command. Here, the copy command must specify the COM port as the source, and the file name as the destination, e.g.:

C:\8051>**COPY COM1** ***file*****name.HEX**

Because the file is in the hex format, all the characters are readable and printable characters, and the file can be inspected with any text editing or display package. Due to timeout errors on the PC while it waits for the data to come in, zeros may be read and stored on the disk file before the hex format data. This will not create any serious problems when the file is inspected, because every INTEL Hex format line can easily be identified: It must start with a colon and end with a carriage return. The zeros may be deleted through normal text editing in any word processing package, or left as they are. If the file needs to be reloaded back into the board at a later stage, the zeros will be ignored by the reception routine. The routine will only start to react after the initial colon of an Intel Hex format line was received.

Care must be taken with the restoring of a block of external data to the 8K byte that corresponds to the Operating System, between addresses 5000H and 6FFFH: The MSB of the address will be changed when it is restored, and it will be directed to the protected Operating System area. To prevent this problem, move

the block of data to an unused section of the user program area before dumping. After reloading, move the block of data back to the original location.

If a user needs to save the current settings of a user program at any break point to a PC disk file for later retrieval, the Bulk Storage Data can be dumped by using this mode. The following initial settings should be used:

**Starting Address:** F000h

**Number of Bytes:** 0200h

The byte addresses will be included in the INTEL Hex data file, and will automatically be restored to the same addresses when reloaded back from the PC. If the original program is still in the code memory, or reloaded from the PC, a user can continue from the same programming step, with exactly the variables and settings he had, before the storage.

**Active Function Keys:** (RS232 Memory Dump)

| | |
|---|---|
| < or > | Move the cursor to select another baud rate. |
| Ent | Select the current baud rate, or accept the displayed starting address or the number of bytes, and start transmission. |
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |

**Note:** Once transmission was initiated, pressing any key will abort the transmission mode.

## 8.17. Mode D: Move a Block of External Data

All the information in the code, internal and external data, and the Special Function Registers, is accessible on the external data memory map after the store operation of Mode 3.

The block move mode allows the moving of a block of data to another location within the 64K byte external data memory map of the controller, provided sufficient space is available. Moving can be done upwards or downwards, and block overlapping is also included. If the destination extends beyond the limits of the memory map, an error will be shown as "Too many bytes!". No provision has been made to warn the user when a block is moved into a protected area, and it assumes it was the intention of the user, and that the protection was disabled.

After this mode was entered, the lowest or starting address of the source block must be keyed in. Then the lowest or starting address of the destination block must follow and finally, the number of bytes to be moved.

**Active Function Keys:** (Block Move)

| | |
|---|---|
| < or > | Move the cursor to the left or right. |
| '0'-'F' | Set a new start address, destination address or the number of bytes. |
| Ent | Select the entered source or destination address, or the number of bytes displayed. |
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |

## 8.18. Mode E: Protect/Unprotect Memory

This mode toggles between "to set" or "to clear" the write-protection of the Operating System and the LCA configuration. Care must be taken with this mode, not to destroy important code in the Operating System or data in the LCA configuration. It could cause a system crash. Then the original Operating System and configuration must be restored through the *seeding* process, described later in this section.

It is further possible to download a modified LCA configuration into the RAM, which will then take effect only after the 'ON/Configure' button has been pressed again. A modified Operating System may also be loaded through the RS232 link, but the system may crash when the code for the serial receiving routines is overwritten. Under normal circumstances, it should never be necessary to unprotect the appropriate memory sections at all.

**Active Function Keys:** (Protect/Unprotect Memory)

| | |
|---|---|
| Ent | Toggles between the protect and unprotect modes. |
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |

## 8.19. Mode F: Save Internal Data and Switch Off

To enable the continuation of a program or process from the point where it was stopped before the system was switched off, all the internal data and the contents of the Special Function Registers will be stored in the battery backed up RAM

before the system will switch off. The same contents may then be restored back after the initialization of the micro-controller by the Restore Mode.

If the data and settings stored through Mode 3 must be kept and not overwritten by the current settings and data before switch-off, the previously stored data and settings must first be restored by Mode 4 before the *Store and Off* sequence can be initiated.

When this mode is selected, confirmation is first requested before the sequence begins. Confirmation can be done through any of the 'Enter', 'OFF' or 'F' keys. Key '4' will select the Restore Option if the stored values must be kept instead of the current settings.

**Active Function Keys:** (Store and Off)

| | |
|---|---|
| Ent/OFF/F | Initiates the *Store and Off* sequence. |
| '4' | Select Restore Data and System Variable Mode |
| Help | Call the help screen. |
| Esc | Escape back to the main menu. |

## 8.20. The Seeding Process of the Board

When the board is powered up for the first time, there is no intelligence in the code RAM to enable the configuration of the LCA, nor to use the serial port for serial reception. Similarly, if the LCA configuration or the Operating System has been badly corrupted, it will be impossible to use the board. The *Seeding Process* is the process to load the LCA configuration and the Operating System into the

code RAM from a cold start. This is done through direct configuration and downloading from the PC to the LCA and the code memory through the *seeding* download cable. The Xilinx LCA has six different ways to be configured after power is supplied to the IC. Three of these methods have been incorporated on the board.

The first method of LCA configuration is the **Master Parallel Mode**, which is used for the normal operation of the board. Here the LCA will generate addresses and read data in parallel from a memory, either from the top of the memory downwards, or from the bottom up. This downwards mode is used by the normal hardware configuration of the board, and initiated by pressing the 'ON/Configure' button.

The second method of configuration is the **Serial Slave Mode**, which is used initially for the seeding process. Here an external source, such as a PC, must generate a serial data train and a synchronous clock signal that must be applied to two pins of the LCA. This mode can be selected by closing the SEED jumper on the board. The data and clock signals are applied from a parallel port of a PC to the board, through a parallel download cable and the keyed 6-pin header connection on the board. The initial data in the configuration file contains the number of bits to follow. When this count has been reached, the LCA will come out of the configuration mode and start to operate.

The third method of configuration is the **Serial Master Mode**, and it can be selected by closing the "PROM" jumper in the board. This mode is similar to the

slave mode, except that the LCA supplies the synchronous clock signal, and the configuration can be loaded in from a serial PROM. Provision has been made for this mode by the inclusion of a blank 8-pin IC socket and a PROM select jumper. This mode was included to be able to configure the LCA for an application where the on board Code RAM will be disabled and not used at all.

The seeding process is initiated by selecting the Serial Slave configuration mode, and it will initially configure the LCA to be a **Serial-In-Parallel-Out Shift Register**, and an **Address Counter**. The data is stored in the INTEL Hex format on a disk file called SHIFTREG.MCS, and converted into a serial data train with a synchronous clock through a C-program called SEED.EXE. During the seeding process, the two Running Mode Select slide switches are used as the Address Counter Reset (Sw1) and the Count Direction selector (Sw2).

Once the LCA is configured and the "Seed" jumper removed, the address counter is reset and set to count from the top address downwards. The next serial data train from the PC is then shifted into the shift register, and loaded in parallel into the top end of the code RAM after every eight serial bits. This data contains the configuration that will be used by the final circuit, and is stored on a disk file called DECODER.MCS. This configuration will only take effect with the seed jumper removed to select to Master Parallel mode, and a re-configure command. The stored configuration will then be loaded into the LCA.

The next step is to reset the address counter again, and select the up counting mode. Data from the PC now contains the Operating System, and it is loaded

into address 5000H upwards in the code RAM. The Operating System is stored on a disk file called OPSYS.HEX.

Finally, the download cable is removed and the slide switches set to select the Operating System. When the 'ON/Configure' button is pressed, the LCA will re-configure and the micro-controller will start to execute the Operating System.

The step-by-step procedure of the seeding process is displayed on the screen of the PC by the SEED.EXE program. To start the seeding process, one must make sure that the SEED.EXE, SHIFTREG.MCS, DECODER.MSC and OPSYS.HEX files are all in the current sub-directory. The parallel download cable must be plugged into any parallel port (take note which one, e.g. LPT1, LPT2, etc.) and into the 6-pin header connection on the board. The "Seed" jumper must be closed, and the 'ON/Configure' button pressed. The **Red** LED will go on to show the board is ready and waiting. The SEED program can be run by typing in the seeding program name, "SEED" at the DOS prompt. After a brief description of what the program does, every step is prompted and clarified on the PC screen, as follows:

Step-by-Step Seeding Procedure:

1. Connect the Controller Board to a parallel port of the PC.
2. Close the "SEED" jumper on the Controller Board.
3. Apply power to the board.
4. Press the 'ON/Configure' button. The Red LED on the Controller Board should light up.

*Here the program will test if the files SHIFTREG.MCS, DECODER.MCS and OPSYS.HEX are in the current directory, and if not, it will flag a "*File not Found*" error message.*

5. Select the appropriate printer port of the PC:

    1. LPT1:

    2. LPT2:

    3. Quit.

    (1,2,3):_

*Here the selected printer port will be tested and if not ready, an error message will be given, e.g.:*

Parallel Port 1 may not be ready!

Check the cable and if power is available on the board.

*If the port is ready, it will clear the screen and display the message:*

Port 1 is selected...

6. Press ENTER to start the shift register configuration.

    Or press Q to Quit...

*Here the program will load the SHIFTREG.MSC file, convert it to a serial train, generate the clock pulse and download it to the board through the selected printer port. The screen will display the message:*

Configuring the Xilinx LCA to a Shift Register and Address counter...

*A bell and the word "Done!" will indicate completion.*

7. Open the "SEED" jumper. (Done with shift register configuration.)

8. Set Switch 1 OFF and then ON again, to reset the internal address counter. Switch 2 stays OFF to select loading from the top, downwards.

9. Press any key to continue loading the configuration data for normal operation, into the top of the RAM, or Q to quit:

*Here the program will load the DECODER.MSC file and download it to the board. The screen will display the message:*

Loading the configuration data to the RAM.....

*A bell and the word "Done!" will indicate the completion.*

10. Set Switch 1 OFF then both ON. Switch 1 is the address counter reset. Switch 2 selects the upwards count direction.

11. Press any key to continue loading the code of the operating system into Code RAM, or Q to quit:

*Here the program will load the OPSYS.HEX file and download it to the board. The screen will display the message:*

Loading the operating system.....

*A bell and the word "Done!" will indicate the completion.*

12. Remove the download cable, set all switches to OFF, and press the 'ON/Configure' button of the Controller Board.

13. Now the Controller Board is seeded and ready!

**Note:** A complete listing of the SEED.C program is included in the appendix.

---

# 9. The Hardware and Firmware Description

The hardware and firmware of this board are so interrelated that they have to be described together. Some unique features of the LCA (Logic Cell Array) have been used to enable the board to operate as it does.

The additional hardware to the micro-controller is mainly to add the essential hardware to attain a minimum system for operation. This includes a voltage regulator, the oscillator circuit, the external Code Memory and an address latch. In addition, sufficient hardware has been added to make sensible and userfriendly communication available between the user and the Operating System. A thirty-two-key keyboard and a 16-character by 1-line LCD Display Module were included on the board.

The Operating System enables direct access to all the internal registers and data of the micro-controller, and also the external code and data memories. The hardware either enables serial communication to and from a PC through an RS232 serial port for the downloading of user programs. Storage of the current data on the board to the PC for disk storage is also possible. Careful hardware design and some address line swapping enable a user program to run the micro-controller without any interference and virtually no prerequisites to cater for the existing hardware.

To appreciate the firmware design of the LCA, a brief description of the general operation and architecture of the LCA will be given initially. The Xilinx XC2064 PLCC68 LCA has been chosen, because it is the most economical device of this series that still suits all the requirements of the circuit.

## 9.1. An Overview of the Xilinx 2064 Series LCA

### 9.1.1 The CLB or Configurable Logic Block. (2000 Series)

Consider a combinatorial logic circuit with four inputs, and one (or two) output(s):

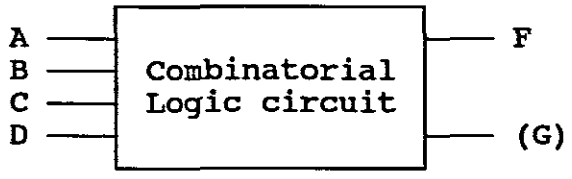



**Fig. 9.1. The CLB Combinatorial Logic Circuit**

Using normal logic AND, NAND, OR and NOR gate circuits, the outputs can be configured to be 1's or 0's for any combination of 1's and 0's on the inputs. The total number of combinations of the four input lines can be shown in a 16-line truth table. The state of the output can be defined to be a 1 or a 0 for any one of the sixteen combinations, shown below:

| A | B | C | D | Output |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 or 0 |
| 0 | 0 | 0 | 1 | 1 or 0 |
| 0 | 0 | 1 | 0 | 1 or 0 |
| 0 | 0 | 1 | 1 | 1 or 0 |
| 0 | 1 | 0 | 0 | 1 or 0 |
| 0 | 1 | 0 | 1 | 1 or 0 |
| 0 | 1 | 1 | 0 | 1 or 0 |
| 0 | 1 | 1 | 1 | 1 or 0 |
| 1 | 0 | 0 | 0 | 1 or 0 |
| 1 | 0 | 0 | 1 | 1 or 0 |
| 1 | 0 | 1 | 0 | 1 or 0 |
| 1 | 0 | 1 | 1 | 1 or 0 |
| 1 | 1 | 0 | 0 | 1 or 0 |
| 1 | 1 | 0 | 1 | 1 or 0 |
| 1 | 1 | 1 | 0 | 1 or 0 |
| 1 | 1 | 1 | 1 | 1 or 0 |

If, instead of using normal logic gates, one were to use a small 16 x 1 bit RAM, where the four inputs represent the address lines, and the output is the single data bit. The input combination will then select one unique address and the data stored at that address, will appear on the output. The advantage of using such a RAM in place of normal logic gates is that the represented logic circuit can be altered by simply re-programming the data in the RAM. A further benefit is that any combination will have the *same* delay through the circuit. Such a circuit can be defined by a general Boolean equation as a function with four variables:

$$F = f(A,B,C,D)$$

To enable a second output, using the same inputs and the same 16-bit RAM, the circuit can be configured to be two equations, each with only three variables:

$$F = f(A,B,C) \quad \text{or} \quad F = f(A,B,D)$$

$$G = f(A,B,D) \quad \text{or} \quad G = f(A,C,D)$$

Some limitations do exist of which inputs may be used together, but because the inputs can be swapped at will, this limitation can be bypassed.

Various programmable options exist for each block. They are selected by other stored RAM bits. These bits will be programmed to the selected options during the configuration of the LCA. The programmable configuration options available, over and above the combinatorial RAM content, are shown as multiplexers in the sketches on the next page:

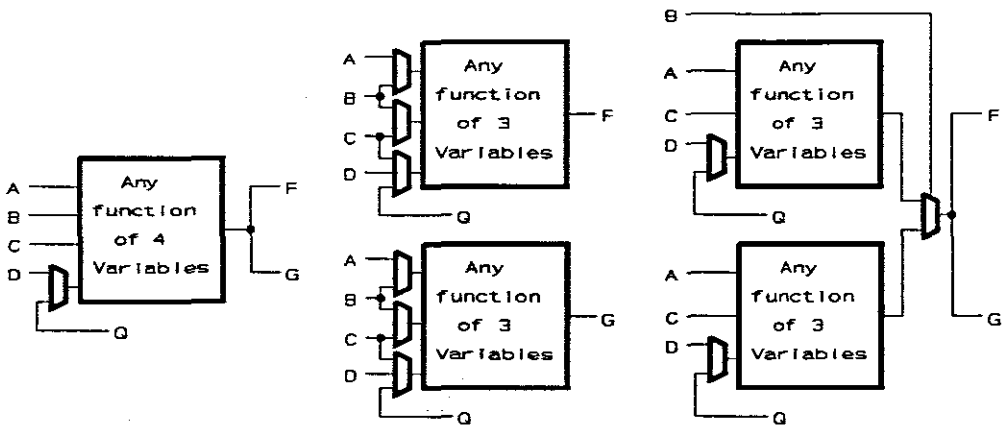



**Fig. 9.3. The Combinatorial Circuit Options**

## 9.1.2 The Storage Element

To complete the Configurable Logic Block, it also contains a storage element in the form of a D-flip-flop, with asynchronous SET and RESET controls.

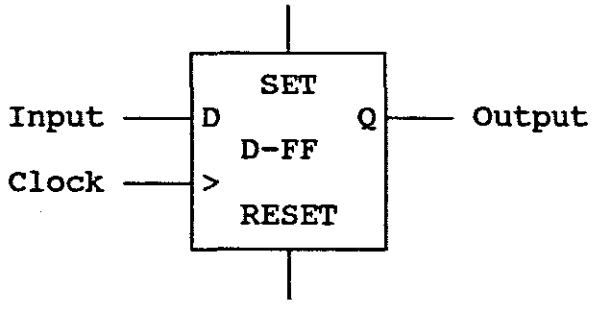



**Fig. 9.3. The CLB Storage Element**

The Input to the D-flip-flop is supplied from the F function of the combinatorial circuit above, and the Clock, Set and Reset inputs can be programmed to be supplied from various sources. The output can also be programmed internally to be fed back as an input to the combinatorial logic section.

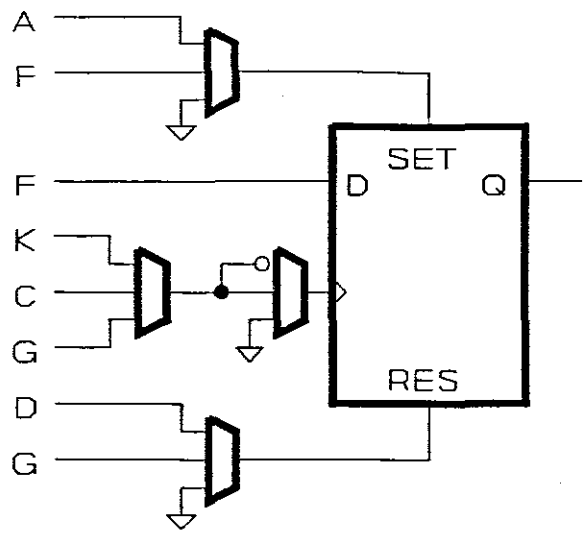



**Fig. 9.4. The Storage Element Options**

### 9.1.3 The Complete Configurable Logic Block

Such a CLB can now be programmed and used to fulfil a host of various functions, such as boolean logic functions, a storage element of a register, one bit of a synchronous or an asynchronous counter, multiplexers, state machines, etc.

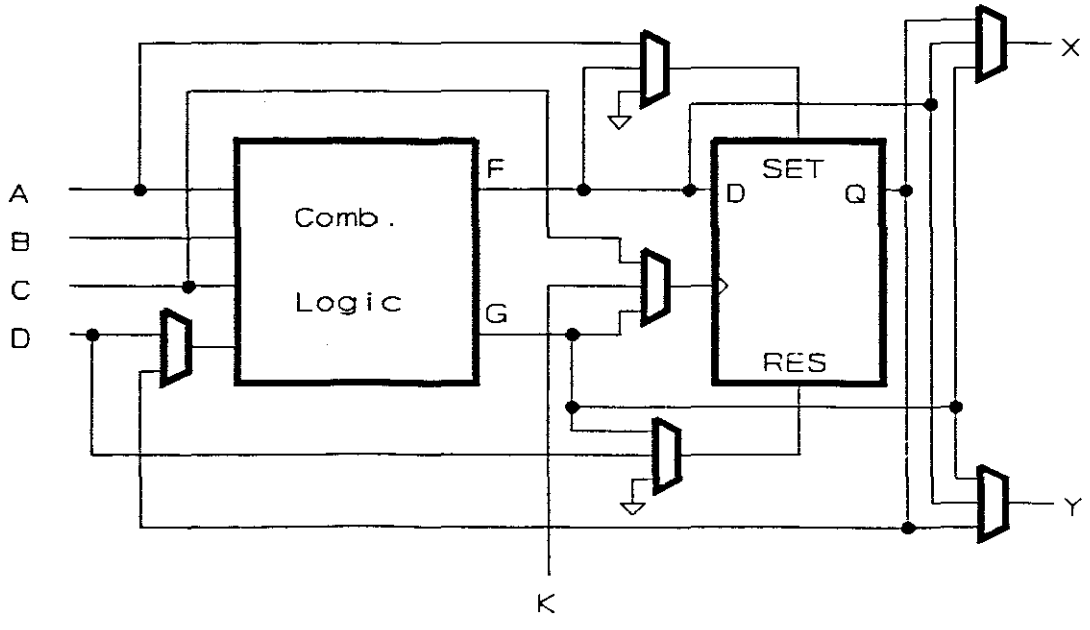



**Fig. 9.5. The Complete Configurable Logic Block**

In the XC2064 device, there are sixty-four of these CLBs, and each can be configured into any of the possible options available. The sixty-four CLBs are organised in a matrix of eight rows of eight CLBs each.

### 9.1.4 The Input/Output Blocks

To interface with the outside world, the LCA also contains I/O blocks, connected to the external pins of the chip. Each pin can be programmed to be either a dedicated output (with tri-state control), a dedicated input (with a storage latch if required) or a bidirectional port. The sketch below shows the architecture and various options available to each I/O Block.

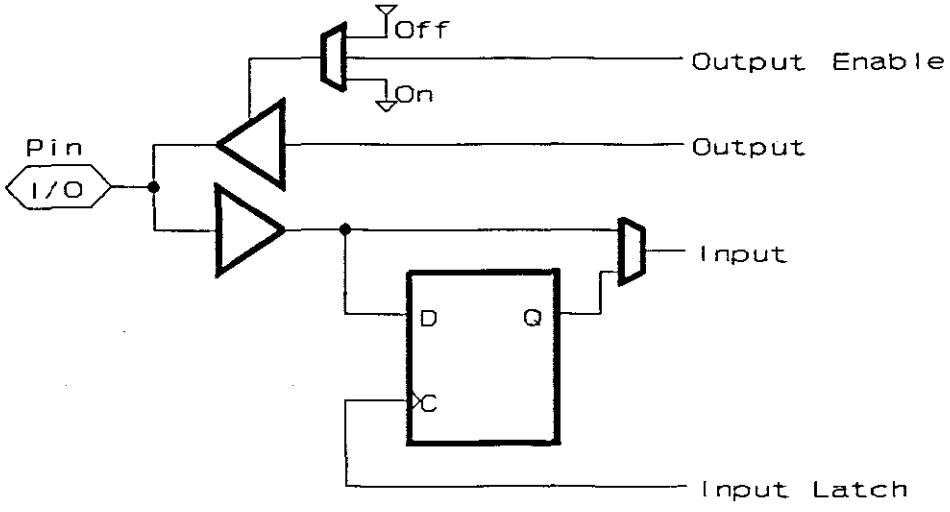



**Fig. 9.6. The LCA Input/Output Block**

There are 58 such I/O blocks available in the XC2064 device, and the I/O Blocks are placed along the four edges of the CLB matrix. The device is available in various packages, such as the PLCC44, the 48-pin DIP, a PLCC68 and a PGA84

package. For the smaller packages all the I/O Blocks are not physically connected to output pins. Nevertheless, the blocks still exist on the device and may be used as storage elements or to create delay lines.

### 9.1.5 CLB and I/O Block Interconnections

Between the rows and columns of the CLBs and I/O Blocks are a number of programmable interconnect lines that can virtually connect any input from any block to any output of the same or any other block. In this manner, a flexible and rather complex digital circuit can be build up. Because all the CLBs and all the I/O blocks are identical, it allows the swapping of the functions of the CLBs inside the chip for the smallest delay time. The swapping of the functions of the I/O pins can place a specific input or output pin at a specific location to suit the PCB layout.

### 9.1.6 The LCA Special and General Purpose Pins

Apart from the 58 I/O pins connected to the I/O Blocks in the device, there are also a few dedicated control pins for the LCA. These are a Master Reset, Power Down, Done/Program, three Mode Selection Control pins, as well as two Vcc and two Ground pins. During the configuration mode, some general I/O pins have dedicated functions, such as the configuration address, data and control lines.

The positions of these special function configuration pins are fixed and cannot be swapped in the design like the general purpose I/O pins for normal operation.

### 9.1.7 The Configuration Modes for an LCA device

There are five modes available to configure the XC2064 devices, and the appropriate mode is selected by the logic levels on the Mode Selection Pins M0, M1 and M2 when power is applied, or when the "Done/Program" control line, **DP**, is pulled low for reconfiguration. While the configuration is in process, the "High During Configuration" pin, **HDC** is kept high, and the "Low During Configuration" pin, **LDC** is kept low. When the configuration is completed, the "Done/Program" line is made high by the device, and the **HCD** and **LDC** pins become general purpose I/O's. Pins that have no special functions during configuration, will be pulled high through internal resistors.

The configuration modes, with a brief description of each, follow below:

**Master Serial Mode.** (M0,M1,M2 = 000b)

The LCA will generate a clock pulse to a serial PROM, that will place the stored data on the Configuration Data pin of the LCA. (This mode is available as an alternate configuration mode for this project.) These serial PROMs also have a RESET pin that should be pulled low to reset the internal address counter of the PROM. This pin is normally connected to the "Done/Program" line, but if the PROM is not reset, a second set of configuration data can be loaded from the PROM, if requested by the hardware.

**Master Parallel Upwards Mode.** (M0,M1,M2 = 001b)

The Configuration Address Lines of the LCA will place a 16-bit address on an external memory, such as a RAM or EPROM. The "Low During Configuration"

pin can be used to enable 8-bits of parallel data output from the memory. This data will then be read into the LCA through the Configuration Data Lines. The address will start at 0000h and increment upwards until the required number of bytes has been read (approximately 1.5K bytes).

**Master Parallel Downwards Mode.** (M0,M1,M2 = 011b)

This mode operates in the same fashion as the Master Parallel Upwards Mode, but the address starts at FFFFh, and decrement downwards until the required number of bytes has been read. (This mode is used by this project for normal operation.)

**Peripheral Parallel Mode.** (M0,M1,M2 = 101b)

In this mode, the device must be mapped and decoded through three Chip Select pins as an 8-bit Parallel Output Device. An intelligent external device, such as a micro-controller, must download the configuration onto the data pins.

**Serial Slave Mode.** (M0,M1,M2 = 111b)

Only two pins are required for this mode, and an external device must generate a clock signal and a serial data train. (This mode is used for the seeding process of this project.)

### 9.1.8 The Design Entry to Realise an LCA Configuration

With the multitude of CLB, I/O block and interconnection options, it is virtually impossible to configure all the connections for a medium to complex digital design by hand and to set up a configuration bit stream to configure the LCA successfully. This is all done by the *Xilinx* program packages on the PC.

Various possible methods of design entry exist and a set of conversion, routing and testing programs will do the layout and report on any errors and excessive routing delays. The layout is then converted into a bit stream that is used to configure the LCA. Modifications to the configuration are relatively easy. It can be done by just changing the original design and feeding it through the entire "mapping and routing" process to generate a new configuration file. Simple modifications, such as inverting the logic of a signal, can be done changing a sign in a formula in the final layout, before a new bitstream is generated.

The design entry method used for this project was the ORCAD DRAFT schematic capture package. It was used to draw the hardware design schematic diagram that must be represented by the LCA. Special symbols, supplied with the package, were used to generate the drawing. These symbols are equivalent to and include all the normal logic gates, such as 2-, 3- and 4-input AND gates, or any combination of one or more inputs being inverted before it is applied to the gate. Similar symbols also exist for NAND, OR, NOR, XOR and XNOR gates. Then there are many types of flip-flops, counters, shift registers and multiplexers, in fact, equivalents for nearly all the normal ICs available to standard digital designs.

The schematic symbols used for the design entry, are special in that they are so defined that a specific type of netlist can be generated once the components have been placed and the connections made on the schematic drawing. Each symbol must be given an unique name, and some signal lines may also be named. These names will be carried through to the final LCA layout, so one can recognise which part of the original circuit is represented by which CLBs and internal connections

of the final layout.

Once the circuit has been converted, the software will normally select a suitable device automatically. One may also choose a specific device and package type, and where specific I/O pins must be located in the final layout. It is also possible to set the "Place and Route" program of the Xilinx package to do repetitive layouts. This will enable a user to select the best layout with the shortest delays and the most effective CLB implementation.

## 9.2. The Hardware Circuits Required to be Built into the LCA

To keep the minimum chip count on the final training board, all the logic circuitry required by the system had to be placed inside the LCA. The requirements, schematic diagrams and operational descriptions for these circuits follow below. These circuits are only extractions out of the final circuit and a complete design entry schematic circuit diagram of the LCA is included in the appendix.

### 9.2.1 The Address Latch

The address latch is used to capture the multiplexed low address byte for external code and external data access. The "Address Latch Enable" pin, **ALE**, from the micro-controller is used to latch the low address byte from Port 0 during the address/data multiplexing cycle. The configuration of the LCA is done in the Master Parallel Mode from the Code RAM, and therefore the data line inputs and the address line outputs had to coincide with the fixed function pins used for the configuration.

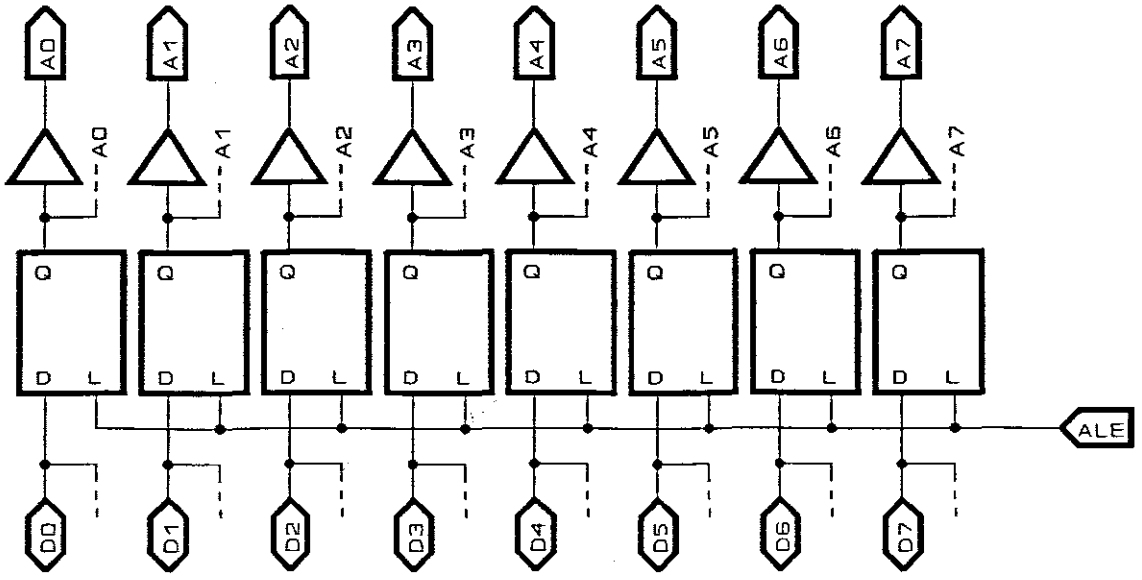



**Fig. 9.7. The Address Latch**

## 9.2.2 The Keyboard Scanner and Decoder

The thirty-two keys of the keyboard are raster scanned by eight row outputs and four column inputs, as shown on the sketch on the next page. A '0' is placed on each row in a successive sequence, and if a key was pressed, the appropriate column input will be pulled low. When this is detected, the current column and the row logic levels are decoded and stored as a binary value between 00h and 1Fh. At the same time a "Data Available" control line generates an Interrupt 0 to the micro-controller. The five bits representing the key value pressed, the "Data Available" signal, DAV, and the state of the two Running Mode Selection slide switches are available to be read by the micro-controller from an internal 8-bit Keyboard Register, KREG.

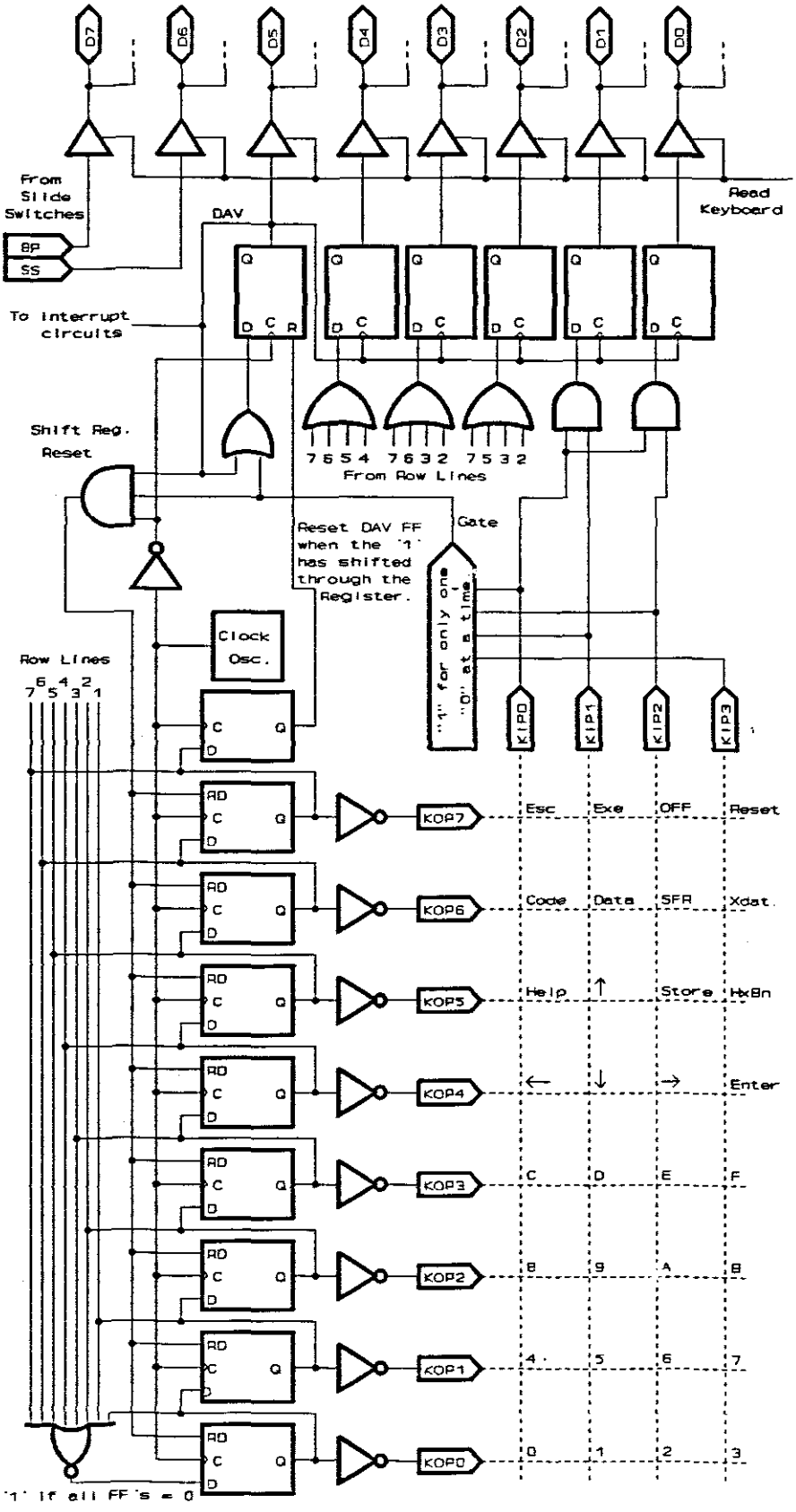



**Fig. 9.8. The Keyboard Scanner and Decoder**

Shown on the schematic drawing on the previous page, the row strobe is generated by an 8-bit shift register. The outputs of the shift register are inverted by the output buffers to supply the pins, and an 8-bit NOR gate from the registers supplies the serial-in data bit of the shift register. Only when all the flip-flops are empty ('0'), a '1' will be placed on the input flip-flop. After that zeros will be placed on the input flip-flop until the '1' has shifted through. An additional flip-flop at the end will capture the '1' and reset the "Data Available" flip-flop, indicating that the key has been released and "*No key has been pressed for the last scan*".

When a key is pressed and the appropriate column line goes low, the GATE signal line will go high to clock a '1' into the DAV flip-flop halfway through the shift cycle on the negative going edge of the clock. The current levels on the column lines, and the logic levels on the row lines are decoded to place an equivalent binary value on the inputs of the keyboard register to be latched by the DAV signal. The DAV signal, the GATE signal and the inverted clock will also reset the column shift register to start scanning from the beginning. This allows sufficient delay between key sampling to eliminate switch bounce. Once the DAV flip-flop is set, the feedback from the output will keep the output high until the key has been released and reset by a '1' moving through the row shift register.

If two column keys are pressed simultaneously, the GATE signal will not go high, and the keys will be ignored. If two row keys are pressed simultaneously, only the first key encountered will be stored. If the key is changed to another legal key while the DAV signal is still high, it will be ignored. The previous value has

already been stored in the keyboard buffer by the positive going edge of the DAV signal.

The DAV signal will also cause an interrupt to the controller. The five data bits, the DAV value, and the state of the two Running Mode Selection switches can be placed on the data bus through the Keyboard Read select line.

### 9.2.3 The LCA Control Register

Three bit functions are controlled by the micro-controller in a "Control Register" in the LCA. They are; the *Write Protect* bit, that would prevent writing to the addresses of the Operating System and the LCA configuration in the Code RAM; the *OFF* bit for the software power off control; and the *Operating System/User Program* select bit, OPS. The OPS bit must be cleared to swap the addresses of the interrupt vectors between the Operating System addresses 5000h to 502Fh, and the user program interrupt vectors between 0000h to 002Fh. The swapping is achieved by applying XOR functions to address lines **A12** and **A14**:
(5000h = 0101 0000 0000 0000b).

### 9.2.4 The Code RAM Decoding

The Code RAM must be enabled for reading code addresses between 0000h and 6FFFh, with the PSEN control line. It must also be enabled for reading and writing to the top 32K byte addresses as external data. The Operating System, mapped between D000h and EFFFh, and the LCA configuration, mapped between F800h and FFFFh on the external data memory map, may only be written to when

the "Write Protect" bit in the LCA Control register is set. The system I/O devices, such as the Keyboard Read register and the Display Module and User I/O Select lines, are all mapped between external addresses F400h and F7FFh, and the Code RAM must be disabled for these addresses. The LCA Control Register is only a *write* register at address F300h (to F3FFh). This address is also overlaid by a code RAM byte, so that the previous contents written to the CREG register can be read by the software. The Boolean Equation implemented in the LCA for the Code RAM Enable output, CRM, is as follows:

$$\overline{CRM} = \overline{(A15 + A14 \cdot A13 \cdot A12)} \cdot \overline{PSEN} + \ldots$$ User Prog & Opsys [0000h - 6FFFh]

$$+ \overline{(\overline{A15} \cdot A14 \cdot A13 \cdot A12 \cdot \overline{A11} \cdot A10)} \cdot \overline{RD} + \ldots$$ Sys.Vars. & CREG [8000h - F3FFh]

$$+ A15 \cdot A14 \cdot A13 \cdot A12 \cdot A11 \cdot \overline{WR} \cdot WP + \ldots$$ Opsys as Xdata [D000h - EFFFh]

$$+ A15 \cdot (\overline{A13} \cdot \overline{A12} + \overline{A14}) \cdot \overline{WR} + \ldots$$ User Prog as Xdata [8000h - CFFFh]

$$+ A15 \cdot A14 \cdot A13 \cdot A12 \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{WR}$$ Sys.Vars. & CREG [F000h - F3FFh]

## 9.2.5 The External Data RAM Decoding

The external data RAM is only available for reading and writing in the lower 32K bytes of the external data memory map. They are used with the Read and Write control pins from the micro-controller. The Boolean equation for the decoding of the External Data RAM IC follows below:

$$\overline{DRM} = \overline{A15} \cdot (\overline{RD} + \overline{WR})$$ [0000h - 7FFFh]

### 9.2.6 The Display Module Decoding

Apart from the eight data lines and the positive active "Enable Control" line, ENA, the display module has two additional mode select control lines, a "Data/Command" line, and a "Read/Write" line. The ENA line is decoded to become active when any external data address between F600h and F6FFh is accessed, and the address lines **A0** and **A1** will select between the "Data/ Command" and "Read/Write" options. The view angle of the multiplexed LCD screen of the module is set by the voltage from a preset potentiometer on the **Vo** pin of the unit. A description of the internal registers and the programming requirements for the LCD Display Module is included in the appendix.

The Boolean equation for the ENA control line is:

$$ENA = A15 \cdot A14 \cdot A12 \cdot \overline{A11} \cdot A10 \cdot A9 \cdot \overline{A8} \cdot (\overline{RD} + \overline{WR}) \quad \text{... [F600 - F6FFh]}$$

### 9.2.7 The Two User I/O Select Line Decoders

For convenience, two user select lines, CS0 and CS1, are available. Each covers 256 address bytes for reading and writing between F400h to F4FFh and F500h to F5FFh on the external data memory map. Further decoding can be done on a user board using address lines **A0** to **A7**, and the Read and Write control lines.

The Boolean equations for the two Chip Select lines are:

$$\overline{CS0} = A15 \cdot A14 \cdot A12 \cdot \overline{A11} \cdot A10 \cdot \overline{A9} \cdot \overline{A8} \cdot (\overline{RD} + \overline{WR}) \quad \text{.. [F400 - F4FFh]}$$

$$\overline{CS1} = A15 \cdot A14 \cdot A12 \cdot \overline{A11} \cdot A10 \cdot \overline{A9} \cdot A8 \cdot (\overline{RD} + \overline{WR}) \quad \text{.. [F500 - F5FFh]}$$

### 9.2.8 The Micro-Processor Reset Control

To keep the micro-controller in the reset state during configuration of the LCA, the micro-controller reset pin is connected to the "High During Configuration" pin, HDC. Once configured, this pin becomes a general purpose I/O pin, the "$\mu$-Controller Reset", URS. It is connected to generate a reset when the 'Reset' button is pressed. This reset signal will also clear the CREG register and enable the Code RAM *Write Protect* bit. It will also disable the power *OFF* control and clear the *OPS* bit. The cleared OPS bit will swap the reset addresses 0000h with 5000h to start the execution at the beginning of the code of the Operating System.

### 9.3.9 The Interrupt 0 and Reset Vector Swapping Circuits

For power conservation, and while the Operating System waits for a key response from the user, the micro-controller will be in the idle mode for most of the time. To wake the controller out of the idle mode when a key is pressed, the "External Interrupt 0" is used. When a user program is running in the real time *Execute* mode, the program can be interrupted by pressing the 'Escape' key. This will create an "External Interrupt 0" to return the control to the Operating System. *Single Step* and *Breakpoint* operation make use of a *software generated* interrupt, just before the next user instruction is executed.

To prevent interference from user hardware on the Interrupt 0 pin of the micro-controller, the user must use the special IRQ pin on the edge connector for his Interrupt 0 input. This signal will be directed by the LCA to the Interrupt 0 pin of the micro-controller.

To indicate the source of the interrupt request to the micro-controller, so that the correct action can be taken, the three external interrupt sources, the keyboard, the 'Escape' key and the User Interrupt, will each set a bit in a flop-flop in the LCA. This will keep the Interrupt pin low, until either the keyboard is read to clear the 'Escape' and 'Keyboard' interrupt latches, or the CREG is accessed to clear the External User Interrupt latch. The Interrupt 0 Driver Schematic Diagram is shown below:

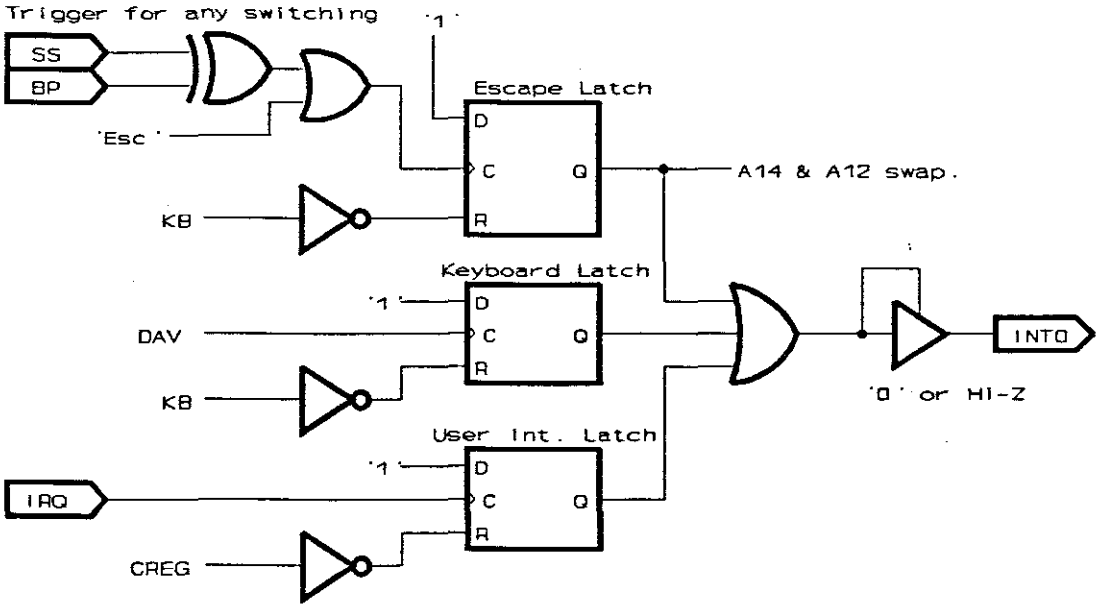



**Fig. 9.9. The Interrupt 0 Circuit Diagram**

After a system reset, the OPS and the Escape Latch bits in the LCA will be cleared. This will swap the code addresses 0000h to 002Fh with addresses 5000h to 502Fh, so that the Operating System can start and take control of the system. Before a user program step is executed, the OPS bit is set to swap the addresses back to normal operation. The user program will then start at address 0000h with all it's normal interrupt vectors in place. After a single step execution, the OPS

bit will be cleared again to enable the Operating System to use its own interrupt vectors, instead. When the 'Escape' key is pressed during normal user program execution, the Escape Latch bit will create an address swapping of only addresses 0000h to 0007h, with addresses 5000h to 5007h. This includes only the Reset and Interrupt 0 vectors. The interrupt subroutine will return control to the Operating System.

The Boolean equations for the code address swapping are:

$$Y = \overline{A15}\cdot\overline{A14}\cdot\overline{A13}\cdot\overline{A12}\cdot\overline{A11}\cdot\overline{A10}\cdot\overline{A9}\cdot\overline{A8}\cdot\overline{A7}\cdot\overline{A6}\cdot(\overline{\overline{A5}\cdot\overline{A4}\cdot OPS} + \overline{A5}\cdot\overline{A4}\cdot\overline{A3}\cdot ESCL)$$

[0000 - 002Fh] [0000 - 0007h]

```
A14O (Output) = A14  XOR  Y
A12O (Output) = A12  XOR  Y
```

A complete list of the pinouts and functions of the XC2064 PLCC68 LCA package is included in the appendix.

## 9.4. Additional Hardware Circuits External to the LCA

### 9.4.1 The Power Control Circuits

To enable a 3-minute auto-off feature for power conservation when the circuit is not in use, the on/off switching of the system has to be software controlled. The circuit is switched on by pressing the "ON/Configure" button, but switching *off* must be done by executing a specific software routine. When power is applied, the micro-controller has to be kept in a reset state while the LCA is configured. The circuit to comply with all these requirements, is shown on the next page:

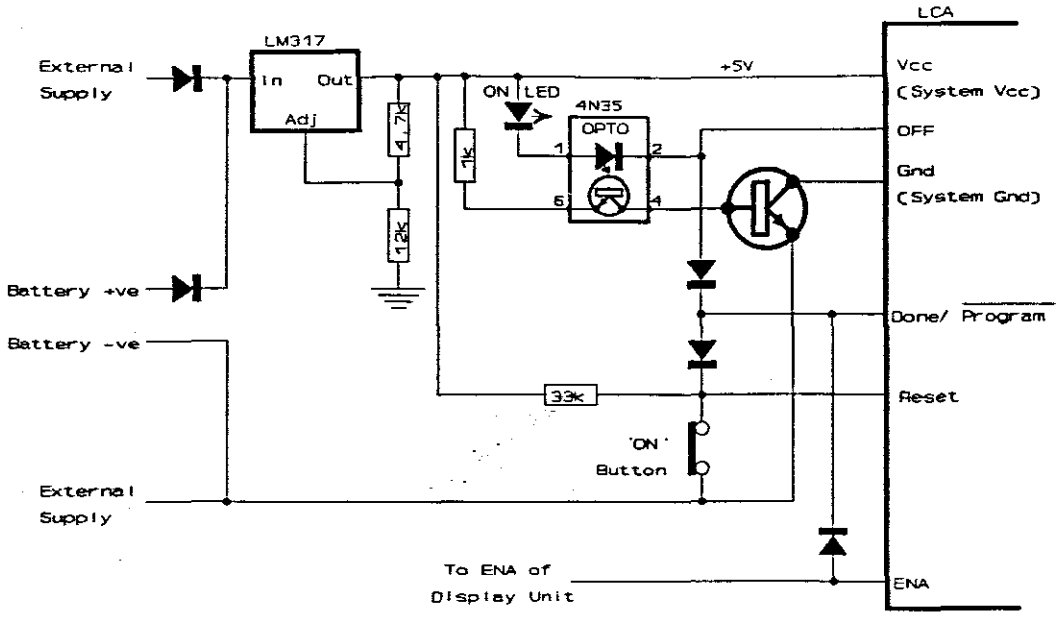



**Fig. 9.10. The Power Control Circuits**

The 'ON/Configure' button will make the initial power connection through an opto-isolator and a current enhancement transistor. After the LCA Reset and Done/Program pins have both been pulled low and the Reset pin made high again by the release of the button, the configuration will start. The Done/Program pin will become an output and it will stay low to keep the power active during configuration, and it will go high after completion. Then the OFF pin will become active low, take over, and keep the power on. The OFF pin is selected to be the output of the OFF flip-flop in the CREG register, which will switch the power off when set by software.

The general purpose I/O pins not used during configuration, will be pull high by internal resistors. Only the positive active ENA control of the display unit will be affected badly by such a high signal, and a diode to the Done/Program pin will keep this control signal low and inactive.

If the LCA Reset pin is pulled low during configuration, the configuration process will restart, but once completed, this Reset control will act as a global reset for all the registers in the LCA.

### 9.4.2 The RS232 Serial Link to the PC

The voltage level protocol required for RS232 communication is that a 'low' be represented by any voltage between +3 and +30 volt, while a 'high' is presented by any voltage between -3 and -30 volt. There are no suitable negative voltages on the board, so a MAX232 logic level converter had to be incorporated. This IC will convert the normal +5 volt from a digital circuit to +10 and -10 volt. It has two input buffers and two output buffers that will interface the RS232 protocol voltage levels to the normal digital circuit logic levels of 0 and +5 volt.

The serial port pins of the controller, RX and TX, are connected through the MAX232 and the serial cable to the TX and RX pins on the serial port of the PC. To enable the use of the RX pin for normal I/O operation while the serial port is not in use, a diode will pull the controller pin low when a '0' is received. The internal pull-up of the controller will pull the pin high for a received '1'. When the serial cable is removed, a MAX232 internal resistor will keep the output of the chip high, so it will not interfere with the other uses of the RX pin.

The RS232 handshaking lines are not used by the system as such, but the "Request to Send" (RTS) and the "Clear to Send" (CTS) lines were connected and looped through the RS232 device. Thus if the cable is not plugged into the board, or the power to the board is off, a break in the loop through the system will be created.

This can be detected by the software of the PC. The RTS and CTS can also be connected to the "External Interrupt 1" and the T0 pin through wire jumpers to enable the implementation of the handshaking lines in a communication system.

The other two RS232 handshaking lines, the "Data Terminal Ready" (DTR) and the "Data Set Ready" (DSR), are shorted in the 25-pin female D-connector, plugged into the PC. A test in the software for the integrity of this connection, will detect if the serial cable is not plugged into the PC. The through connections from the PC to the controller are shown in the sketch below:

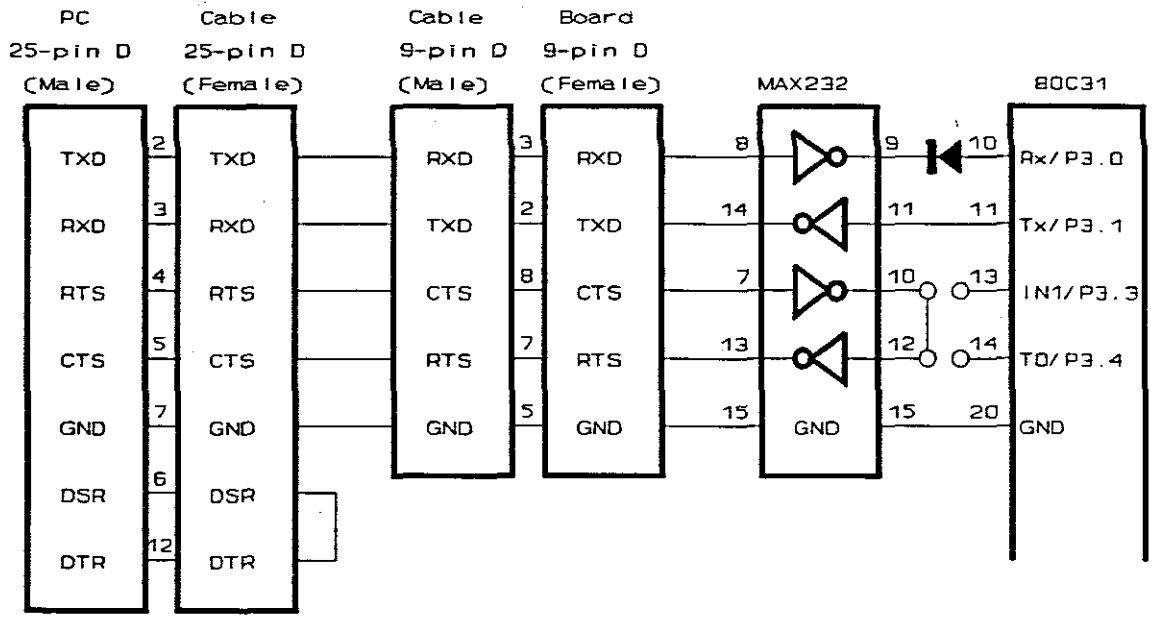



**Fig. 9.11. The RS232 Cable Through Connections**

## 9.4.3 The Seeding Circuits and Jumpers

Three configuration modes are possible on the system, the *Master Parallel Downwards Mode* for normal operation, the *Serial Slave Mode* for the seeding process, and the *Serial Slave Mode* when the normal configuration must be loaded

from a serial PROM. The logic levels of the Mode Control pins are as follows:

| Configuration Mode | M0 | M1 | M2 | Description |
|---|---|---|---|---|
| Master Parallel(down) | 0 | 1 | 1 | LCA supply addresses to CRAM |
| Serial Slave | 1 | 1 | 1 | Ext clock and data from PC |
| Master Serial | 0 | 0 | 0 | LCA clock data from Serial PROM |

To change from the Master Parallel(down) mode to the Serial Slave mode, only M0 need to be pulled up. This is achieved by pulling down the M0 pin on the board through a resistor and the **Seed** jumper will pull the pin up to Vcc.

To alternate between the Master Parallel(down) mode and the Master Serial mode, both the M1 and M2 pins must be pulled low. This is achieved by pulling both pins high through a resistor, and the **PROM** jumper will pull and keep both pins to ground. When a PROM is used for the configuration, the configuration is already stored on the board and the seeding process to load the configuration is not required.

The "Low During Configuration" pin is used to enable the Code RAM during configuration. The Red 'CONF' LED between the HDC and LDC pins will light up during this time. When the SEED jumper is inserted, it will pull the OUTPUT ENABLE and CHIP SELECT pins of the Code RAM high. The data output from the Code RAM will then not interfere with the serial data being loaded from the PC. The LDC pin from the LCA is also selected as the "Code RAM Enable Control", CRM, for normal operation. The schematic diagram for the *seeding* circuits and connections is shown on the next page:

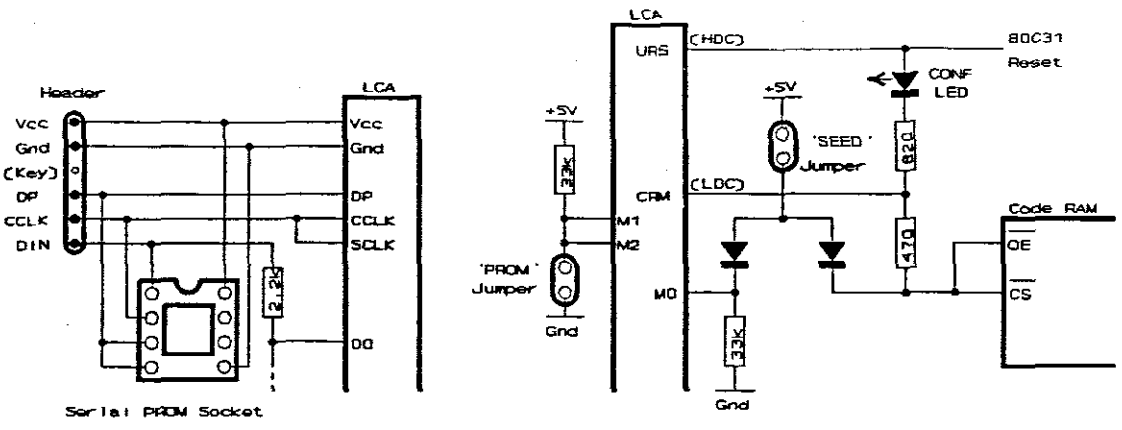



**Fig. 9.12. The External Seeding and Select Circuits**

The Seeding cable from the PC can be connected either to the 6-pin header supplied on the board, or to the empty 8-pin DIL serial PROM socket. The Done/Program line is also available on the header to enable the use of the serial download program and cable supplied by *Xilinx*. The connections for the Seeding cable are included in the appendix.

### 9.4.4 The LCA Configuration for the Seeding Process

The internal configuration of the LCA during the seeding process consists of a Serial-In-Parallel-Out Shift Register, a Bit Counter and an Address Counter. Only two signal lines from the PC are required to first configure the LCA in the Serial Slave Mode as the shift register, then to load the normal configuration into the top of the Code RAM, and finally, the Operating System into its appropriate addresses between 5000h and 6FFFh.

The two signal lines from the PC consist of a Clock signal and Data signal, both generated by the "SEED.EXE" program from the PC. This program will load the

seeding configuration from the file called SHIFTREG.MCS, and clock the data out on the parallel printer port through the seeding cable. For the Serial Slave Mode, the clock signal must be applied to the "Configuration Clock" pin, CCLK, while the data must enter through the "Data-In" pin, DIN. The DIN pin is also the D0 pin for parallel configuration and normal data access from the memory. The Done/Program, HDC and LDC pins have the same logic levels as for any other configuration type. The "Done/Program" pin will keep the power on and the HDC will keep the micro-controller in the reset state. The LDC will attempt to enable the Code memory, but it will be kept inactive high by the "seed jumper". The complete circuit diagram of the Seeding Configuration is included in the appendix.

Once the LCA is configured as a shift register, the OFF and Display Enable pins will be kept low permanently. The reset to the micro-controller, as well as the address pins **A15** and **A14** will be kept high permanently. Address **A15** is not connected to the Code RAM and address **A14** must be high for all the addresses of both the configuration from FFFFh downwards, and the Operating System from address 5000h upwards. The program will remind the user on the PC screen to remove the SEED jumper to enable the control of the Code RAM. It will also request that the slide switches must be set to select down counting of the address counter and to preset the counter to FFFFh. The SEED.EXE program will load the normal configuration from the disk file called DECODER.MCS and it will start to clock the data out to the shift register in the LCA.

The clock signal will now be applied to the user defined Serial Clock, SCLK, and the data will still be entered at the D0 pin, set to be an input. The most significant

bit of a data byte is shifted in first at the positive transition of the clock pulse. After 7 clock pulses, the least significant bit will be on the input pin D0. On the negative transition of the CLOCK, a modulus-7 counter will activate the Code RAM enable and the WRITE control pins to the RAM. The seven shifted-in data bits, and the LSB on the D0 pin will be stored in the RAM. The following positive transition of the clock will disable the writing, reset the modulus-7 counter and decrement the address counter. Now the shift register is ready to receive the next data byte.

After the normal configuration has been loaded into the RAM, the user is prompted by the PC to select the up counting and to preset the counter to 5000h. This is done through the setting of the slide switches. Then the Operating System is read from the file OPSYS.HEX, converted to serial data and shifted into the LCA in a similar process as for the configuration data.

Once the loading is completed, the seeding download cable is removed and the 'ON/Configure' button pressed. The LCA will now re-configure in the Master Parallel Mode and extract the data from the top end of the Code RAM.

### 9.4.5 The 62-Pin Extension Edge Connector

The extension edge connector socket is a standard IBM motherboard bus connector and blank IBM prototype cards can be used to add circuitry to the Training board. The pin numbers run from A1 to A31 on the side that faces the user, which will probably be the component side of the extension card. B1 to B31 are on the underside. All the pins of the micro-controller are directly connected to the pins

of the extension socket, and a few extra features that may be useful for external circuits. The pin numbers, names and functions of some of these few additional features are included in the table below. A complete list is included in the appendix.

| Pin | Name | Description |
|---|---|---|
| A1 | GND | Circuit Ground (NB: Not the battery ground!) |
| A2 | Vcc | +5 V circuit positive (NB: Not the battery positive!) |
| A3 | $\overline{CS0}$ | External Chip Select 0. Addresses F400h to F4FFh. (256 bytes. Further decoding can be done with addresses **A0** to **A7**, and the $\overline{RD}$ and $\overline{WR}$ control lines.) |
| A4 | $\overline{CS1}$ | External Chip Select 1. Addresses F500h to F5FFh. |
| A5 | $\overline{IRQ}$ | User External Interrupt 0, in place of Port 3.2. A high to low transition on this pin will latch an interrupt to P3.2. The latch must be reset by a write instruction to the LCA control register. |
| A28 | ON | External ON control. (Connect to BT- to switch on) |
| A29 | BT+ | Unregulated positive supply to the extension card. |
| B28 | OFF | OFF control from LCA (Goes positive to switch off) |
| B29 | BT- | To Negative Battery Terminal (NB: Not Circuit Ground) |

# 10. Software Description

## 10.1. The Approach to Writing the Operating System

In the initial planning, it soon became apparent that the type of software required to achieve the objectives of the Operating System, would need very careful planning. A multi-menu key-driven system can divide the flow of the program into many possible pathways. The selection of each pathway must follow a simple logic that would be easy to understand, predict and anticipate for any user. Each pathway chosen must give sufficient feedback to confirm that the correct action has been taken, and it must follow through to return to the central core of the program in a logic way. Many keystrokes at various points in the program will give the same results, and these pathways must be identified, grouped together, and reduced to single subroutines. The planning of such a system can only be done effectively with the extended use of sequential lists and flow diagrams.

Numerous modifications during the planning and design stages of such a system are inevitable. After some initial planning and sketches on paper, it was decided to use the drawing package "ACAD" to do draw all the flow diagrams. This allows one to make modifications easy, without having to scratch out and overwrite modified sections, or to redraw sections that stay the same. After each modification a new neat printout can be made to base the rest of the planning on.

It was soon realised that the conventional method of using flow diagrams does not simplify the understanding and the ability to have an overview of a specific

section of the system as one would like it to be. The conventional method used for flow diagrams has therefore been modified to establish two different types of flow diagrams. To identify each type and for the lack of established traditional names, the two types of flow diagrams were called *Broad Based Flow Diagrams* and *Single Line Flow Diagrams.*

The **Board Based Flow Diagram** will be used to clarify a point in the program where a number of equal options are available. A typical situation is where a user may press any one of a number of keys of equal preference. This is indicated as a horizontal line with a single entry and multiple exits. An example of a Broad Based or Operating Flow Diagram is shown below:

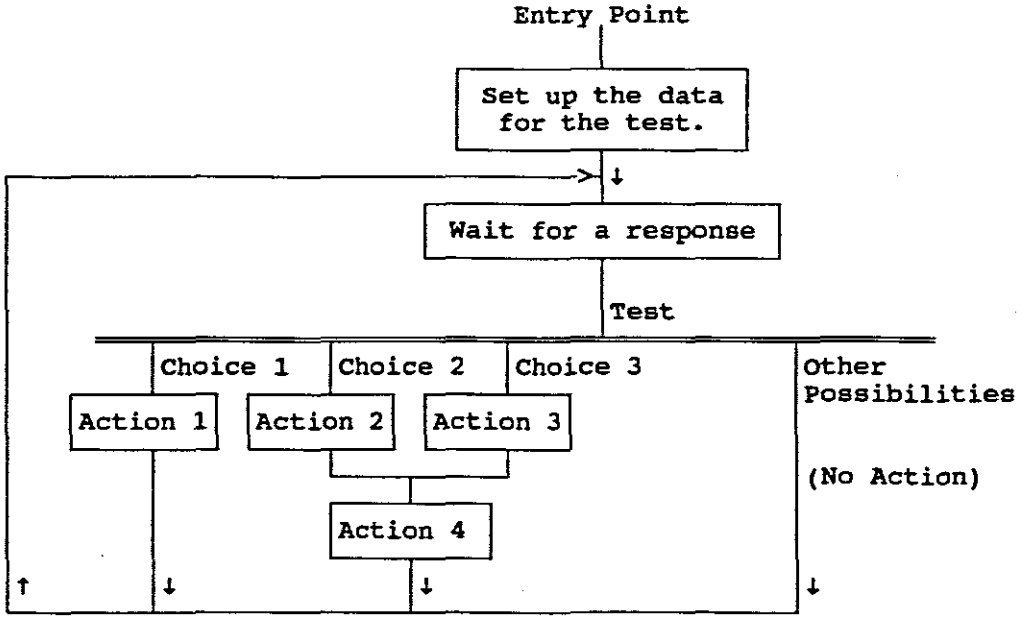



**Fig.10.1. A Broad Based Flow Diagram**

Each exit will be represented by the pressing of a selected key or group of keys. Some actions taken will be the same or closely related, while others may each

have its own set of instructions to follow. After the completion of these actions, some choices may go back to the starting point of this selection. Others may fall through to the next broad selection line, or move the flow of the program completely over to another section of the program. The logic of this type of flow diagram is relatively easy to follow at a glance and invaluable for debugging the program during the performance tests.

Another suitable name for this type of flow diagram in this specific application, is an **Operating Diagram**, showing how the logic and choices of each part of the program operated.

A **Single Line Flow Diagram**, on the other hand, relates more closely to the final assembler programming steps below one another, and the sequence of how the various programming modules will be placed in the whole program. Forward and backward jumps in the program are indicated as lines moving past program sections to specific entry points. The convention was adopted to place forward jump lines on the right-hand side of the flow diagram symbols and the backward jump lines to the left. Each entry point of a jump must be given an unique name or "label". These labels were already selected and defined in flow diagrams. A label must make sense in describing the action of the section, but still be unique in the program.

The composition of the single line flow diagrams was derived directly from the broad based flow diagram. Starting on the left option of the multiple choices, one may use simple "*test and jump if not true, or stay if true*" instructions. The

test will be followed by the set of action statements, and finally a jump statement to wherever the program flow must go from there. Then the next test instruction will follow, etc. The single line flow diagram also proved to be very valuable to select positions for intermediate jumps. They are required to extend the range of relative jumps that are limited to about 128 bytes forwards or backwards.

Once the single line flow diagrams were completed and all the labels selected and placed, it was very simple to convert this into the actual programming steps. The action and branching blocks of the single line flow diagrams were mostly kept in the descriptive form of an operation. Using "Get Switches", or test and jump if "Key > 0Fh", is easy to understand at a glance, but simple enough to convert directly to the actual programming steps. Modifications to the program were made very easy by starting at the broad based flow diagrams, transfer it to the single line flow diagrams, and finally to the specific programming instructions. The unique labels made the location of the routines very easy.

## 10.2. Assembler Language vs "C for Micro-Controllers"

Consideration was given to writing the entire Operating System in C, but after a few trail tests, it was found that the code generated by a routine in C was "traditionally" long and cumbersome. It also made such extensive use of subroutines that the stack area required was far too big to afford. One of the objectives of the Operating System was to make all the internal data and registers available for the user program. It meant that a large portion of the user data had to be stored temporarily for later re-use, whenever the control were handed back

to the Operating System. To keep a solid control of the variable placing and especially the stack requirements of the Operating System, it was decided to rather write the program in Assembler. The use of the single line flow diagrams helped to make this choice easy and simple to write, to locate and to modify.

## 10.3. An Overview of the Operating System

The Operating System consists of sixteen modes and each mode is available to the user for a specific task. The system was designed as a so-called "menu-driven" system. It uses the display unit to specify the options available and the keyboard to make the choice. Mode 0 can be considered to be the "Main Menu", and the other fifteen each a sub-menu. Each sub-menu returns to the main menu after the action was completed, or cancelled by the 'Escape' key. The keystrokes required in every mode were made as clear as possible, so that a user can go directly to that section he wants to go to, and do what needs to be done with the minimum of effort.

Help files for all the major modes were also included. Each Help file explains the basic requirements for the current mode and the action available from the keys. Various one second flash messages were included at various points to explain what was required, such as "Select Baud Rate", before the actual options are displayed on the screen. The aim was to enable a user with some basic knowledge of the internal architecture and operating requirements of a micro-controller, to operate the system effectively with the minimum or no reference to the handbook.

## 10.4. Internal Data System Variables and Stack Definition

To limit the internal data area required for the operation system to the minimum, the system variables of all routines were limited to the first 8 data bytes of internal data memory, and the next 16 bytes for stack operations. The global and local system variables are shown below:

| Address | Name | Description |
|---|---|---|
| 00h | Mode | Specifies current mode |
| 01h | LoAdd | Low Address of memory byte being edited |
| 02h | HiAdd | High Address of memory byte being edited |
| 03h | CurAd | Current Cursor Address |
| 04 - 07h | various | Local variables (defined in various modes) |
| 08 - 17h | Stack | (Using the default Stack reset value of 07h) |

## 10.5. System Input/Output Addresses

| | |
|---|---|
| F300 - F3FFh | As 1 WRITE byte to the Control Register, CREG, in the LCA, overlaid by a READ/WRITE memory storage byte to enable the reading of the current setting. |
| F400 - F4FFh | External Chip Select, CS0 (256 bytes) |
| F500 - F5FFh | External Chip Select, CS1 (256 bytes) |
| F600h | Display Module Command Instruction WRITE address |
| F601h | Display Module Data WRITE address |
| F602h | Display Module Status READ address |
| F603h | Display Module Data READ address |

F604 - F6FFh (Display Module mirror image addresses)

F700 - F7FFh 1 READ byte for the Keyboard and the current setting of the Running Mode Select Switches.

## 10.6. General Startup Routines after Power Up or a Reset

After power up and the configuration of the LCA, or after a system reset, the following initial settings are done:

* The Operating System and configuration are write protected.

* The OPS bit in the Control Register in the LCA, for swapping the user and Operating System reset vectors, as well as the OFF bit, are cleared. This is actually done by the hardware, but to have a copy of the current setting of the Control Register in the System Variables, these instructions were included.

* The Display Module is initialised. A complete description of the initialisation process for the Display Module is included in the appendix. Two user definable characters are defined on the display unit as the "up arrow" (↑) and the "down arrow" (↓). These two characters are extensively used by the Operating System display messages.

* The Break Point Table End, just after the thirty-two Break Point Addresses, is defined as 0000h, to stop any further breakpoint address comparisons at this point.

* The External Interrupt 0 is set at high priority and the Interrupt sources are set to be edge-triggered. All the interrupt flags are cleared.

* Two 1 second messages are briefly displayed on the screen. They are "Cape Technikon" and "$\mu$C Trainer V:4.3".

* The Running Mode Selection Switches are then read to either go to the Operating System, or go directly to execute a user program from address 0000h in the *Single Step*, the *Break Point* or the real time *Execute Mode*.

The Single Line Flow diagram, and the Assembler code listing for the initialisation routine, are shown on the next few pages:

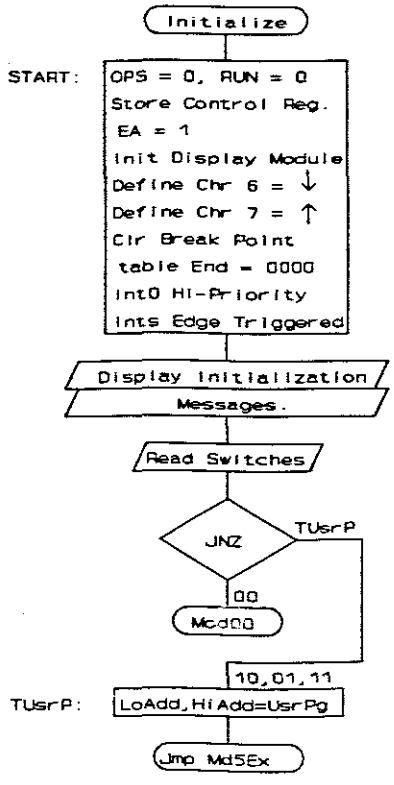



**Fig.10.2. The Initialisation Flow Diagram**

## Assembler Listing

```
;*************************************************************
;        Initialization Routine
;*************************************************************

;        Initialize LCA Control Registers

START:   MOV DPTR,#CntRg            ;Reset LCA Control Register
         MOVX A,@DPTR               ; XData as was,
         ANL A,#08H                 ; Write Protect = 0,
         MOVX @DPTR,A               ; OFF = 0, OPS = 0
                                    ; to swap 0000 - 002F ->5xxx
;        Initialise Display

         SETB EA                    ;Enable ALL
         MOV DPTR,#0FFB5H           ;15 msec
         CALL WPSEC

         MOV DPTR,#DspCm            ;Display Command Address
         MOV A,#3BH                 ;1st Display Setup Instruction
         MOVX @DPTR,A               ;Write Instruction

         MOV DPTR,#0FFE0H           ;Wait 6.4 msec (32 x 0.2 ms)
         CALL WPSec

         MOV DPTR,#DspCm
         MOV A,#3BH                ;2nd Display Setup Instruction
         MOVX @DPTR,A

         MOV DPTR,#0FFFFH           ;Wait 200 µsec
         CALL WPSec

         MOV DPTR,#DspCm
         MOV A,#3BH                 ;3rd Display Setup Instruction
         CALL WrDsp
         CALL WrDsp                 ;Write Instruction 4th time
         CALL ClDsp                 ;Reset Display
         MOV DPTR,#DspCm
         MOV A,#06H                ;Set Cursor to Inc & Disp Steady
         CALL WrDsp
         MOV A,#0DH                ;Disp ON with Cursor Char Blink
         CALL WrDsp

;        Define User Characters 06 and 07 (Down & Up Arrows)

         MOV A,#70H                 ;Add of User Character 6 (Dn)
         CALL WrDsp
         MOV DPTR,#DspDa
         MOV A,#00000100B           ;Define Down Arrow Chr = #6
         CALL WrDsp
         CALL WrDsp                 ;         1
         CALL WrDsp                 ;         1
         CALL WrDsp                 ;         1
         MOV A,#00010101B           ;         1
         CALL WrDsp                 ;     1   1   1
         MOV A,#00001110B           ;       1 1 1
         CALL WrDsp                 ;         1
         MOV A,#00000100B
         CALL WrDsp
         MOV A,#00000000B
         CALL WrDsp
```

```
        MOV A,#00000100B        ;Up Arrow Chr = #7
        CALL WrDsp
        MOV A,#00001110B        ;         1
        CALL WrDsp              ;       1 1 1
        MOV A,#00010101B        ;     1   1   1
        CALL WrDsp              ;         1
        MOV A,#00000100B        ;         1
        CALL WrDsp              ;         1
        CALL WrDsp              ;         1
        CALL WrDsp
        CALL WrDsp

        MOV A,#00000000B
        MOV DPTR,#DspCm
        MOV A,#80H              ;Cursor on 1st Character
        CALL WrDsp


;       Set up system variables in Memory

        MOV DPTR,#BPEnd
        CLR A
        MOVX @DPTR,A            ;0000H at end of BP table
        INC DPTR
        MOVX @DPTR,A
        MOV IP,#01H             ;EX0 high priority
        MOV TCON,#01H           ;ExInt0 = Edge and Clr Flags
                                ; for SS, BP & Exe running modes
        MOV DPTR,#MsgI1         ;Display Init Messages
        CALL DspMg              ;'Cape Technikon'
        CALL W1sec
        MOV DPTR,#MsgI2         ;'µC Trainer V:4.3'
        CALL DspMg
        CALL W1sec
        MOV DPTR,#KyBrd
        MOVX A,@DPTR
        ANL A,#0C0H             ;Mask in Switches only

        JNZ TUsrP               ;To Execute User Program or
        JMP Mod00               ; to Operating System

;       Run User Program from Initialisation

TUsrP:  MOV DPTR,#UsrPg         ;Normal User Program Start
        MOV LoAdd,DPL           ;Start Add in Hi & LoAdd
        MOV HiAdd,DPH
        JMP Md5Ex               ;To Mode 5 RET Exit

;*************************************************************
```

## 10.7. Mode 0: The Main Menu

The current mode value is stored as a number between 0 and Fh in the most significant nibble of the Mode system variable byte. Sub-modes are stored in the least significant nibble. The Operating Diagram as a Broad Based Flow Diagram, showing the various options and paths available, the Flow Diagram and the Assembler Listing for Mode 0 are shown on the next few pages:

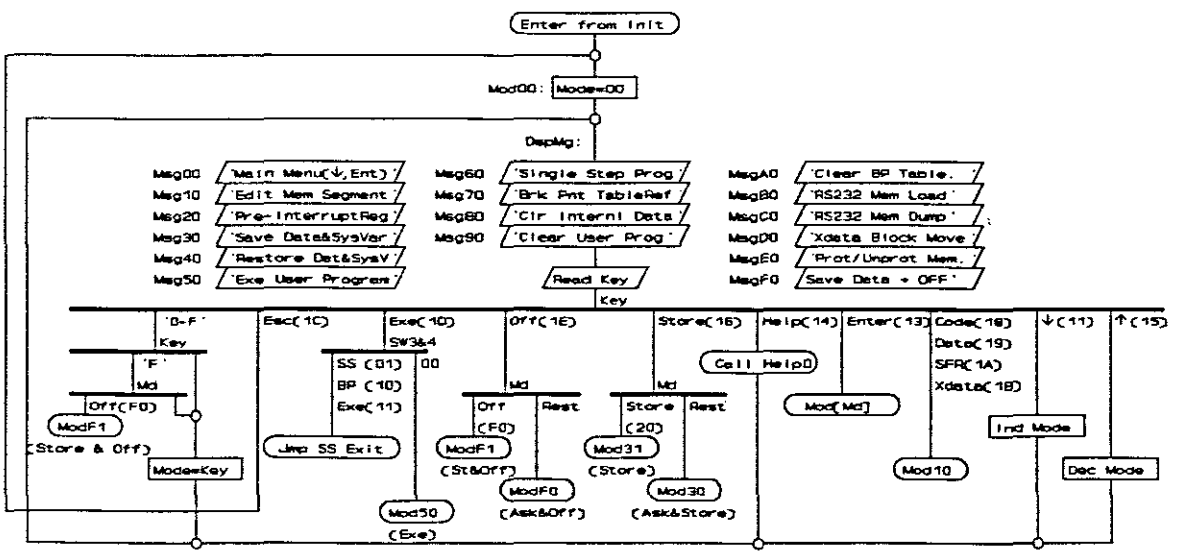



**Fig.10.3. The Main Menu Operating Diagram**

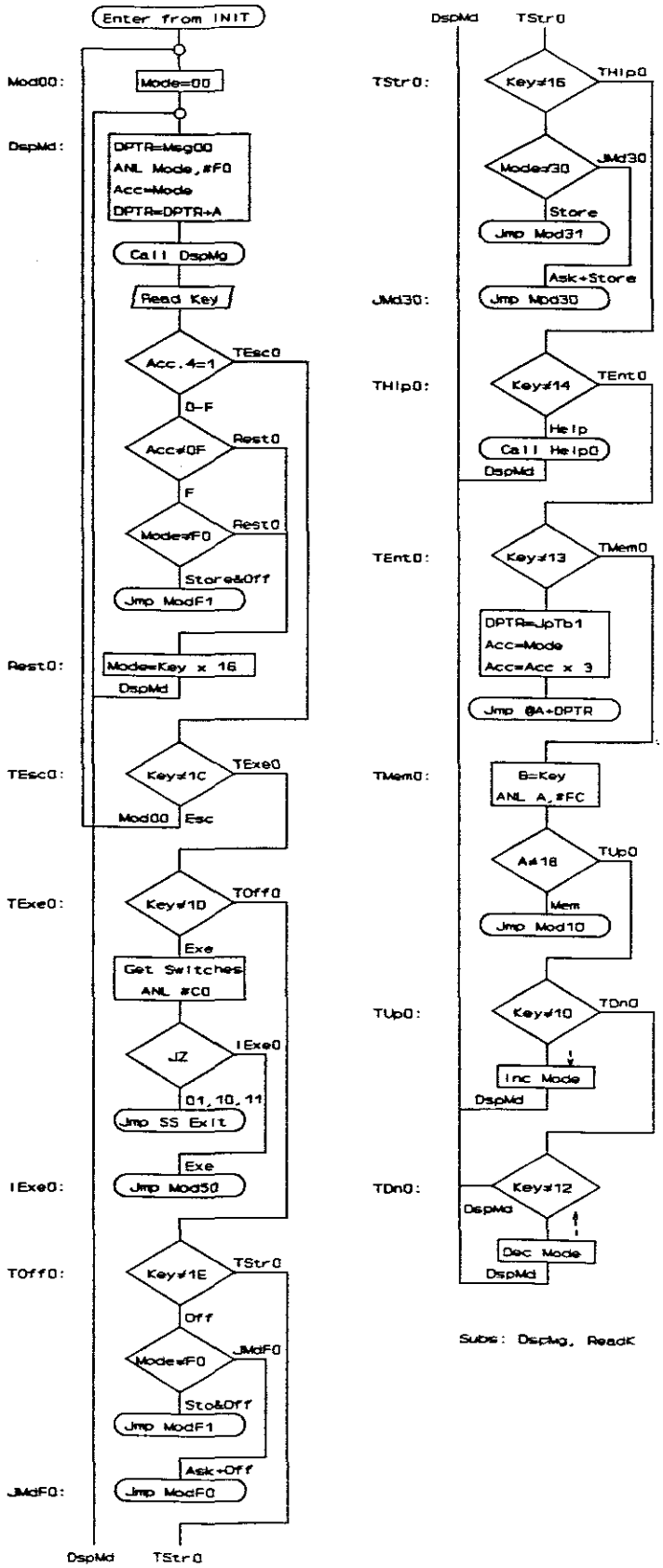



**Fig.10.4. The Main Menu Flow Diagram**

## Assembler Listing

```
;*************************************************************
;       Main Menu - Mode 0
;*************************************************************

Mod00:  MOV Mode,#00H
DspMd:  ANL 00H,#0F0H               ;Cancel any Sub Modes
        MOV DPTR,#Msg00
        MOV A,Mode                  ;Add mode to Msg00
        ADD A,DPL                   ; to display message
        MOV DPL,A
        MOV A,DPH
        ADDC A,#00H
        MOV DPH,A
        CALL DspMg

        CALL ReadK

        JB Acc.4,TEsc0              ;Jmp if not a Number Key
        CJNE A,#0FH,Rest0           ;Test if it is OFF Key
        CJNE Mode,#0F0H,Rest0       ;Is Mode F selected as well
        JMP ModF1                   ;Direct Save & Off

Rest0:  SWAP A
        MOV Mode,A                  ;Set Mode=Key
        JMP DspMd

TEsc0:  CJNE A,#1CH,TExe0           ;Test if Esc Key
        JMP Mod00

TExe0:  CJNE A,#1DH,TOff0           ;Test if Exe Key
        MOV DPTR,#KyBrd             ;Get Switches
        MOVX A,@DPTR
        ANL A,#0C0H                 ;Mask in Switches
        JZ IExe0                    ;To Mode 5 for No Switches
        JMP SSExt                   ;To SS & BP Exit

IExe0:  JMP Mod50                   ;Long jump to Mode 5

TOff0:  CJNE A,#1EH,TStr0           ;Test if OFF Key
        CJNE Mode,#0F0H,JMdF0       ;Is Mode F selected as well
        JMP ModF1                   ;Direct Save & Off

JMdF0:  JMP ModF0                   ;Confirm Save & Off

TStr0:  CJNE A,#16H,THlp0           ;Test if STORE Key
        CJNE Mode,#30H,JMd30        ;Test if Mode = Store also
        JMP Mod31                   ;Store directly

JMd30:  JMP Mod30                   ;Confirm Store

THlp0:  CJNE A,#14H,TEnt0           ;Test if Help Key
        MOV DPTR,#Help0             ;Main Menu Help
        CALL Help
        JMP DspMd

TEnt0:  CJNE A,#13H,TMem0           ;Test if Enter Key
        MOV DPTR,#JmpTb             ;DPTR = Jump Table
        MOV A,Mode
        SWAP A                      ;Mode in LSN
        MOV B,#03H
        MUL AB                      ;Mode x 3
        JMP @A+DPTR                 ;Jump to current selected mode
```

```
TMem0:    MOV B,A                       ;Store Key in B
          ANL A,#0FCH                   ;All Mem keys to 18
          CJNE A,#18H,TUp0              ;Test if Mem key
          JMP Mod10                     ;To Mode 1 with B=Mem Segm

TUp0:     MOV A,B                       ;Get Key back in Acc
          CJNE A,#15H,TDn0              ;Test if Up Key
          MOV A,#0F0H
InDcM:    ADD A,Mode                    ;Subtract 10H from mode
          MOV Mode,A
IDspM:    JMP DspMd

TDn0:     CJNE A,#11H,IDspM             ;Test if Down Key
          MOV A,#10H
          JMP InDcM

;************************************************************
;         Mode 0 Jump Table
;************************************************************

JmpTb:    JMP Mod00
          JMP Mod10
          JMP Mod20
          JMP Mod30
          JMP Mod40
          JMP Mod50
          JMP Mod60
          JMP Mod70
          JMP Mod80
          JMP Mod90
          JMP ModA0
          JMP ModB0
          JMP ModC0
          JMP ModD0
          JMP ModE0
          JMP ModF0

;************************************************************
```

## 10.8. Mode 1: Edit Memory Segment

This routine allows the user to inspect and alter any internal or external memory byte of the system. The main entry point is from Mode 0, where the Code segment will be selected automatically. Other entry points are form the main menu when one of the memory segment select keys 'Code', 'Data', 'SFR' or 'Xdata' has been selected.

It is not normally possible to have write access the Code memory in an 8051 system. This was bypassed by the mapping of the Code memory on the upper half of the external data memory as well, where reading or writing access is done through the "MOVX" instruction. The address of the current byte being edited, is stored in the two system variables 'LoAdd' and 'HiAdd' in the internal data memory. The memory segment being accessed depends on the type of instruction used. The upper address byte for the Internal Data and the SFRs does not have any significance, and for writing or reading to or from the SFRs, a special system specific manipulation had to be applied:

### Reading and Writing to a "Variable" SFR Address

To enable access to any of the specified memory segment bytes on the external or internal memory maps, indirect memory access could be used with the 'LoAdd' and 'HiAdd' bytes containing the address of the byte. For the SFRs, however, indirect addressing does not exit. Indirect addressing to one of these addresses between 80h and FFh, will result in access to the hidden 128 bytes behind the SFRs, which only exists for the 8052 series of micro-controllers.

To access an SFR register specified in the 'LoAdd', a software modification to the Operating System had to be devised to change the direct addressing instructions to contain the specified SFR address as the operand. Because the Operating System is stored in RAM and therefore "soft", this operation was achieved with relative ease: Before a read or write instruction to an SFR register is executed, the software protection bit in the LCA Control Register was disabled. The address value in the operating instruction byte is then changed to reflect the selected SFR address value, before the modified "direct addressing" instruction is executed. The write protection bit was then enabled again before continuing.

The aim of the Edit Memory Segment mode is to enable the user to inspect and change the contents of any byte in any of the segments. Some bytes, however, do not exist in this system, and others may be write protected. To confirm that a specified byte in any of the memory segments was changed by the user, the contents are re-read and displayed from the actual address after the change was written to the byte. If the value does not reflect the value keyed in by the user, it will mean that the address was protected, or does not exist.

The Operating Diagram, the Flow Diagram, and the Assembler Listing of the software routines for Mode 1 follow on the next few pages:

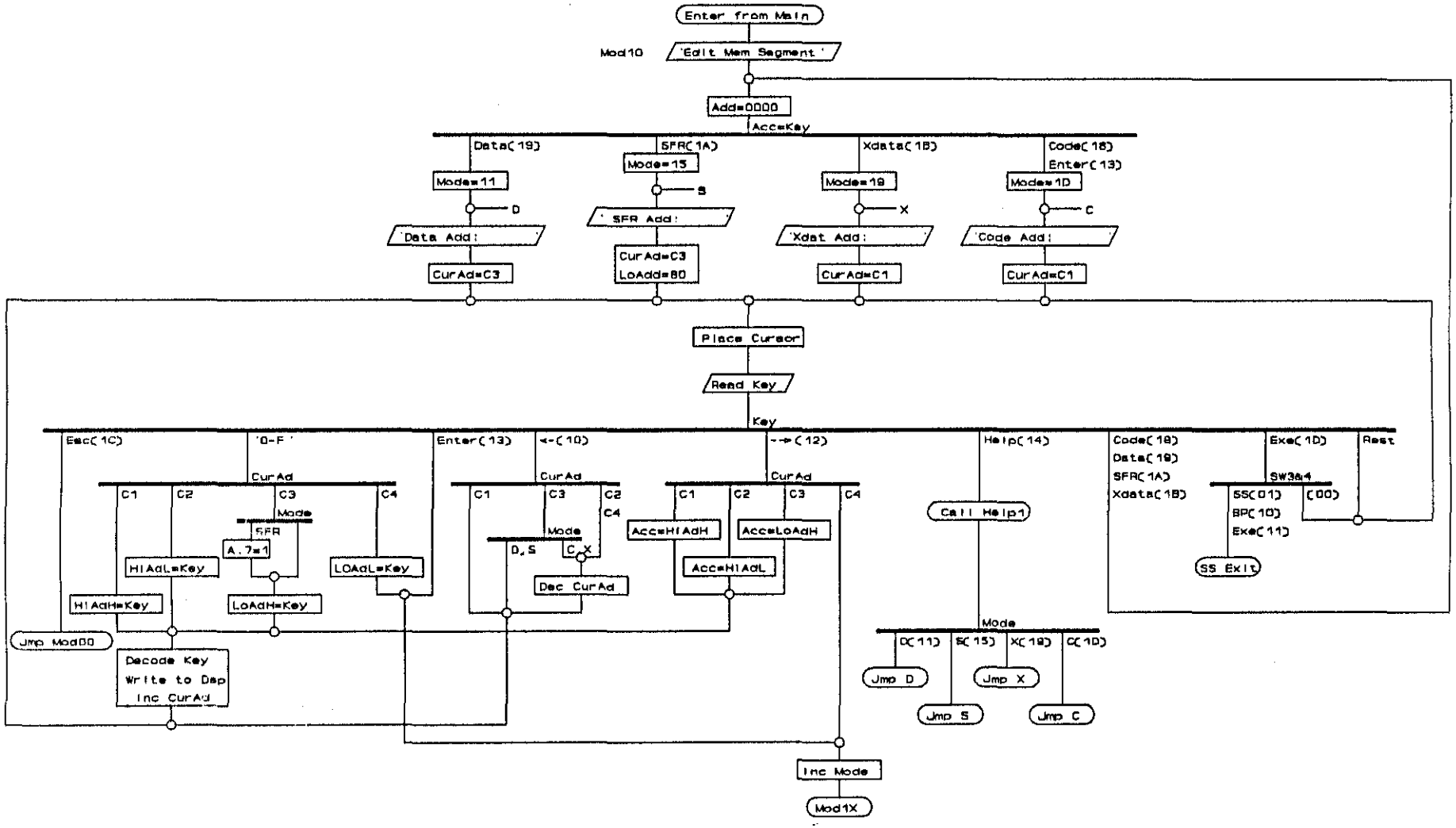



**Fig.10.5. Edit Memory Segment Operating Diagram** (page 1 of 2)

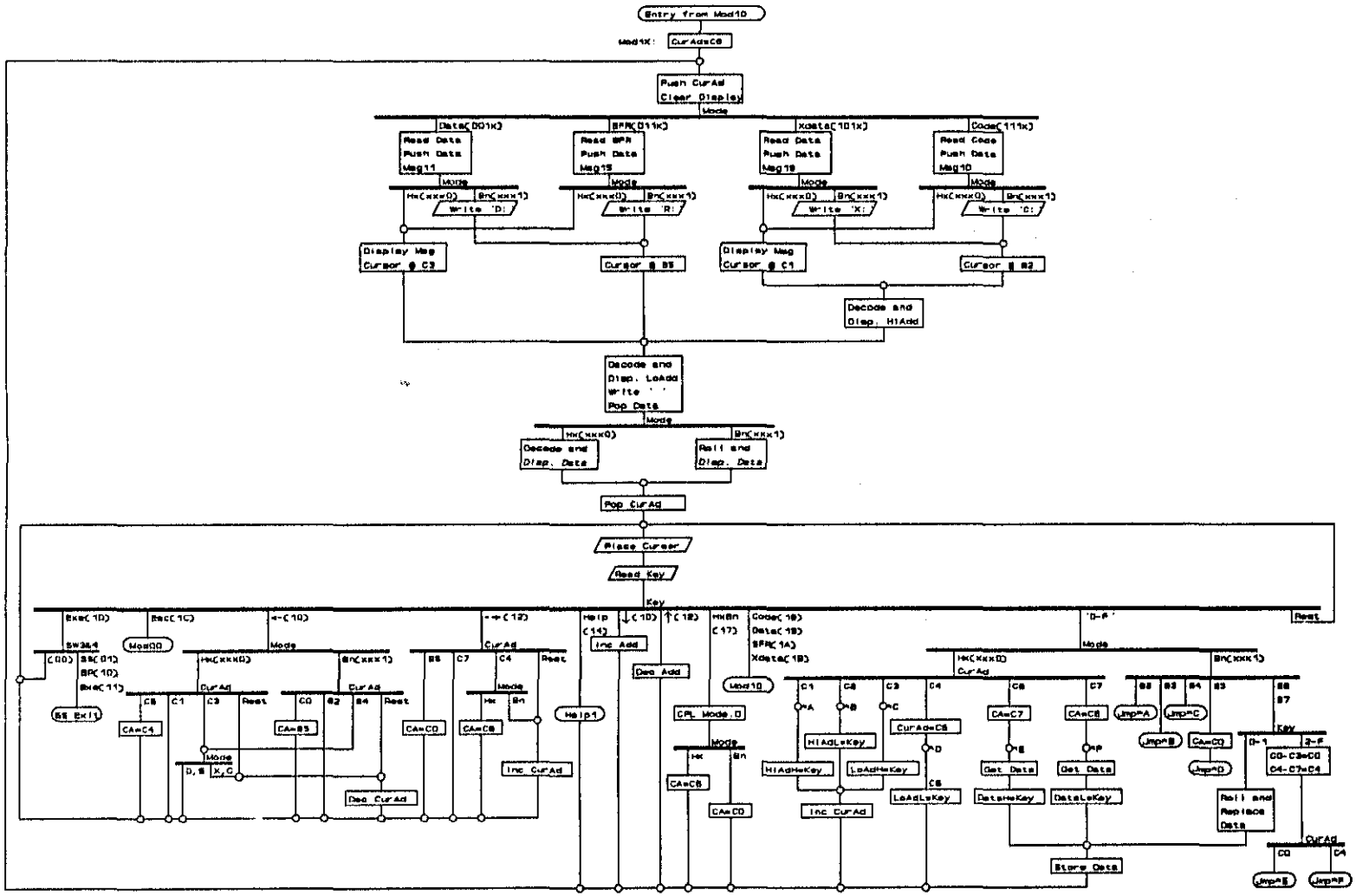



**Fig.10.6. Edit Memory Segment Operating Diagram (page 2 of 2)**

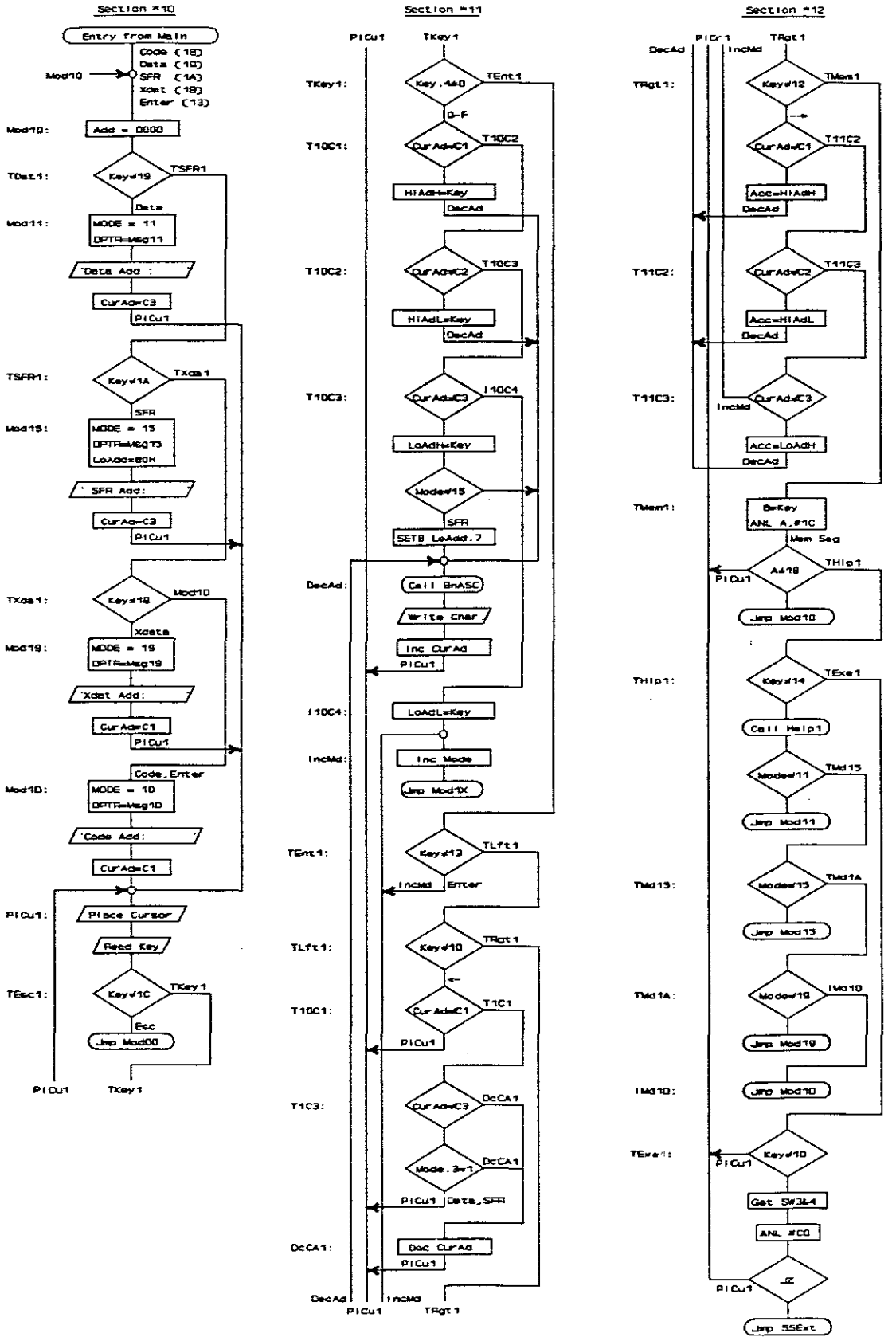



**Fig.10.7. Edit Memory Segment Flow Diagram** (page 1 of 4)

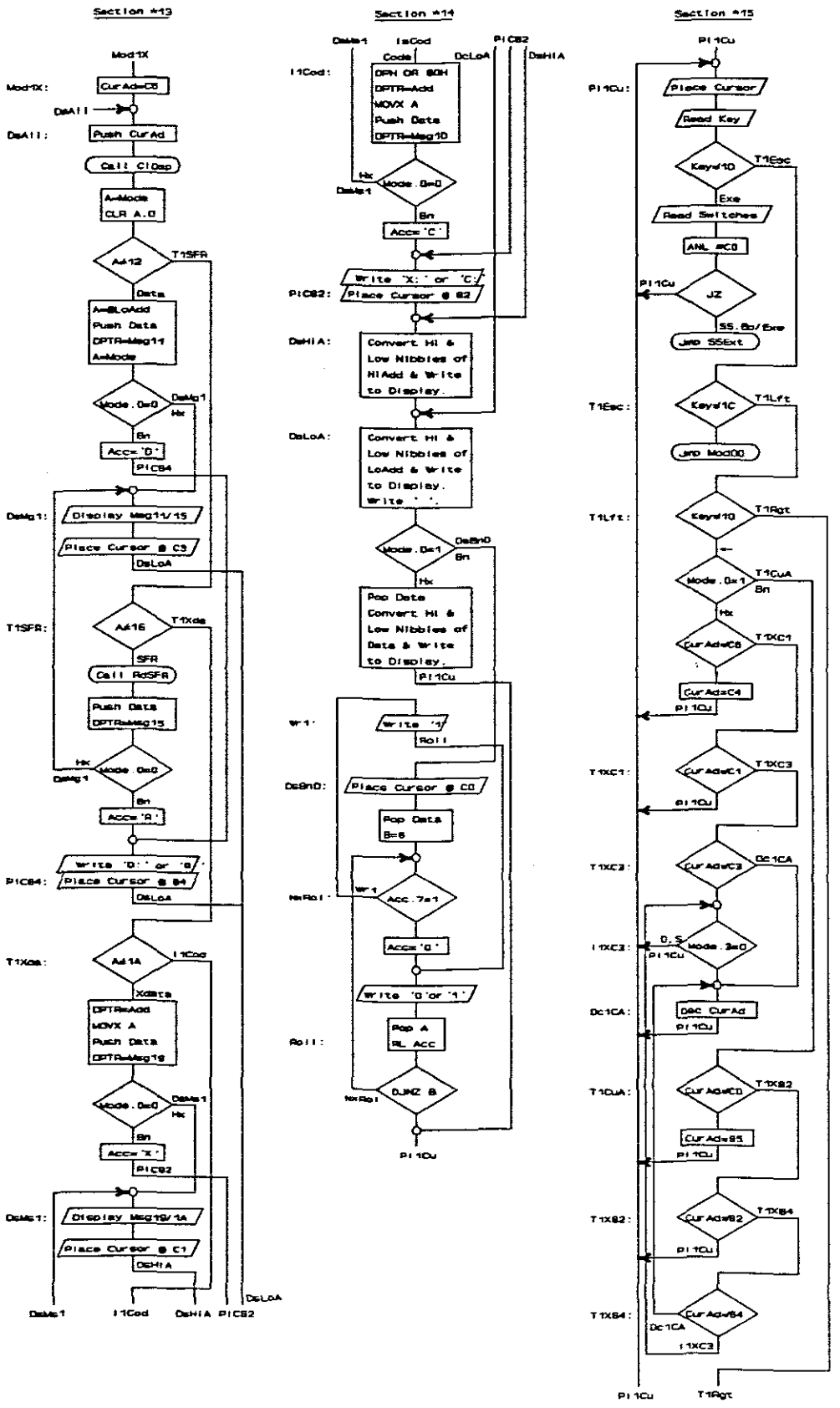



**Fig.10.8. Edit Memory Segment Flow Diagram** (page 2 of 4)

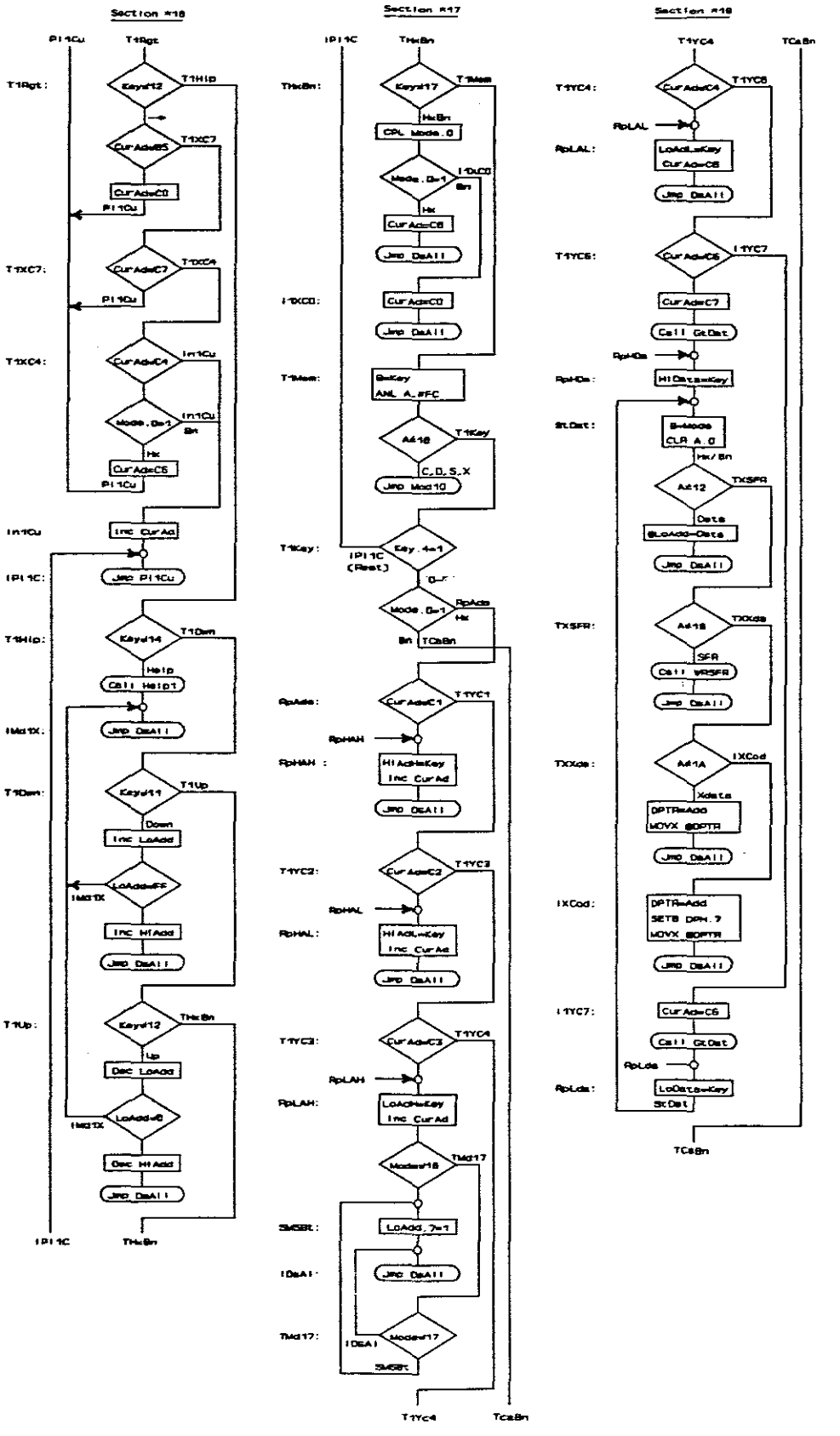


**Fig.10.9. Edit Memory Segment Flow Diagram** (page 3 of 4)

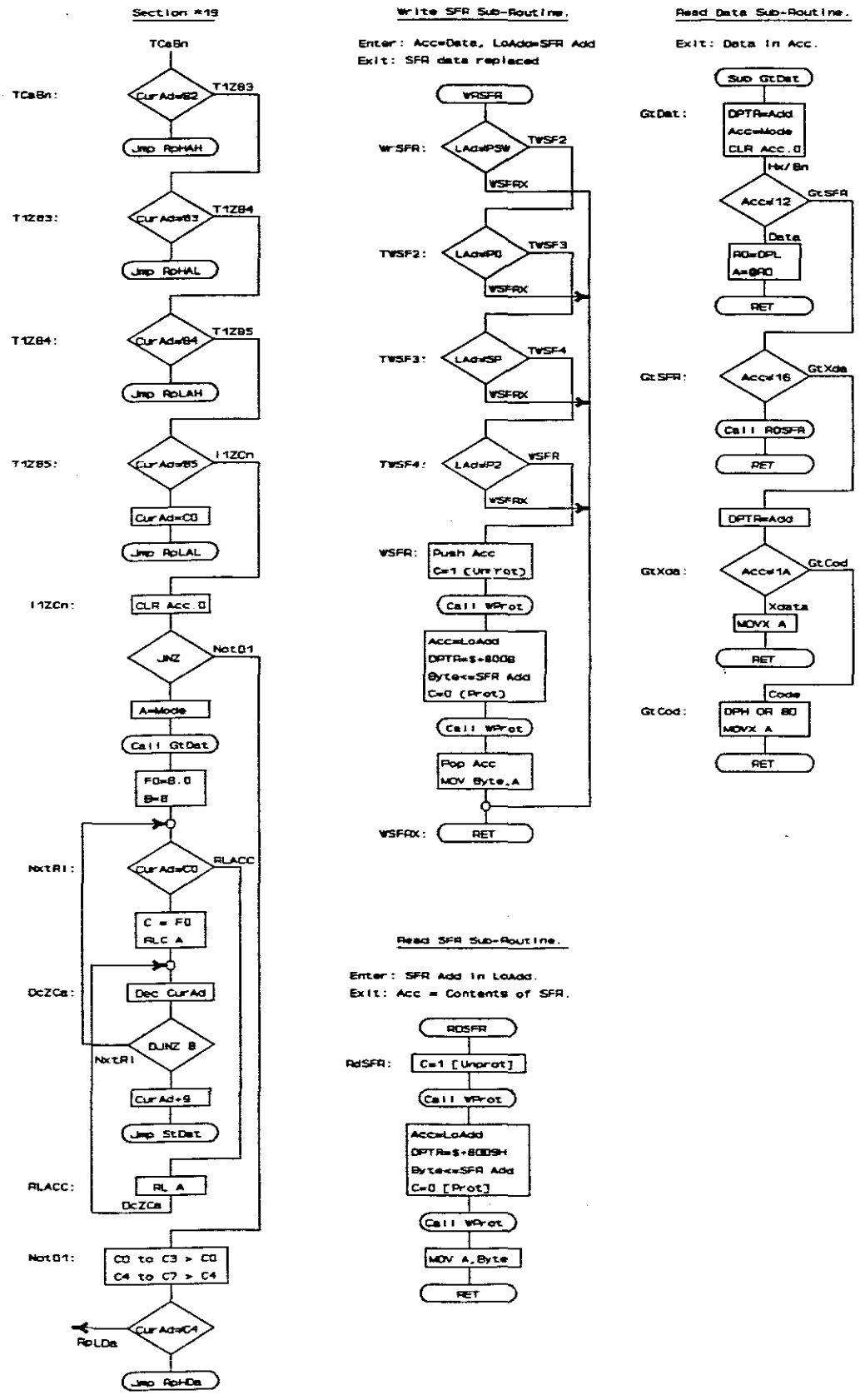



**Fig.10.10. Edit Memory Segment Flow Diagram** (page 4 of 4)

## Assembler Listing

```
;*************************************************************
;       Edit Memory - Mode 1
;*************************************************************

;       Select Memory Segment for Editing

Mod10:  MOV LoAdd,#0                ;B=Mem Key: 18=Code 19=Data
        MOV HiAdd,#0                ;           1A=SFR  1B=Xdata
TDat1:  MOV A,B
        CJNE A,#19H,TSFR1           ;Test if Data Memory
Mod11:  MOV Mode,#11H
        MOV DPTR,#Msg11             ;'Data Add:       '
        CALL DspMg
        MOV CurAd,#0C3H
        JMP PlCu1

TSFR1:  CJNE A,#1AH,TXda1           ;Test if SFR Memory
Mod15:  MOV Mode,#15H
        MOV DPTR,#Msg15             ;' SRF Add:       '
        CALL DspMg
        MOV LoAdd,#80H              ;SFR Add for direct start
        MOV CurAd,#0C3H
        JMP PlCu1

TXda1:  CJNE A,#1BH,Mod1D           ;Test if Xdata Memory Mod19:
        MOV Mode,#19H
        MOV DPTR,#Msg19             ;'Xdat Add:       '
        CALL DspMg
        MOV CurAd,#0C1H
        JMP PlCu1

Mod1D:  MOV Mode,#1DH
        MOV DPTR,#Msg1D             ;'Code Add:       '
        CALL DspMg
        MOV CurAd,#0C1H

PlCU1:  MOV DPTR,#DspCm
        MOV A,CurAd                 ;Place cursor at C1 or C3
        CALL WrDsp

        CALL ReadK

TEsc1:  CJNE A,#1CH,TKey1           ;Test if Escape Key
        JMP Mod00

TKey1:  JB Acc.4,TEnt1              ;Jmp if not Key 0-F

;       Replace digit under cursor with Key value.

T10C1:  CJNE CurAd,#0C1H,T10C2      ;Test if CurAd=C1 (1st Digit)
        SWAP A                      ;Key in Acc MSN
        ANL 02H,#0FH                ;Clear Hiadd Hi
        ORL 02H,A                   ;Write key to Hi HiAdd
        SWAP A                      ;Key back to Lo nibble
        JMP DecAd                   ; for displaying digit

T10C2:  CJNE CurAd,#0C2H,T10C3      ;Test if CurAdd=C2 (2nd Digit)
        ANL 02H,#0F0H               ;Clear Hiadd Lo
        ORL 02H,A                   ;Write key to Lo HiAdd
        JMP DecAd
```

```
T10C3:  CJNE CurAd,#0C3H,I10C4      ;Test if CurAdd=C3 (3rd Digit)
        SWAP A
        ANL 01H,#0FH                ;Clear LoAdd Hi
        CJNE Mode,#15H,STMSB        ;Jump if not SFR
        ORL 01H,#80H                ;Set MSB for SFR Adds 80 - FF
STMSB:  ORL 01H,A                   ;Write key to Hi LoAdd
        SWAP A

DecAd:  CALL BnASC                  ;Key still in Acc
        MOV DPTR,#DspDa
        CALL WrDsp                  ;Write Character to Display
        INC CurAd                   ;Cursor on next character
        JMP P1Cu1

I10C4:  ANL 01H,#0F0H               ;Clear Lo LoAdd
        ORL 01H,A                   ;Place key in LoAdd
IncMd:  INC Mode
        JMP Mod1X

;       Test Other Keys

TEnt1:  CJNE A,#13H,TLft1           ;Test if Enter Key
        JMP IncMd

TLft1:  CJNE A,#10H,TRgt1           ;Test if Left Key
        CJNE CurAd,#0C1H,T1C3       ;Test if CurAd is on digit 1
        JMP P1Cu1                   ; then don't Dec CurAd

T1C3:   CJNE CurAd,#0C3H,DoCA1      ;Test if CurAd is on digit 3
        MOV A,Mode
        JB Acc.3,DoCA1             ;Jmp if not SFR(11) or Data(15)
        JMP P1Cu1

DoCA1:  DEC CurAd
IPCu1:  JMP P1Cu1                   ;Intermediate Long Jump

TRgt1:  CJNE A,#12H,TMem1           ;Test if Right key
        CJNE CurAd,#0C1H,T11C2      ;Test if on 1st digit
        MOV A,HiAdd                 ;Get existing value to display
        SWAP A                      ;HiAdd Hi in Acc LSN
        JMP DecAd

T11C2:  CJNE CurAd,#0C2H,T11C3      ;Test if on 2nd digit
        MOV A,HiAdd                 ;Get existing value to display
        JMP DecAd

T11C3:  CJNE CurAd,#0C3H,IncMd      ;Test if on 3rd digit
        MOV A,LoAdd                 ;Get existing value to display
        SWAP A                      ;LoAdd Hi in Acc LSN
        JMP DecAd

;       New Memory Segment Selected Keys

TMem1:  MOV B,A                     ;Store Key in B
        ANL A,#1CH                  ;Clear 2 LSBs, all ->18
        CJNE A,#18H,TH1p1           ;Test if a Mem Key
        JMP Mod10                   ;Key in B

TH1p1:  MOV A,B                     ;Get Key back in Acc
        CJNE A,#14H,TExe1           ;Test if Help Key
        MOV DPTR,#Help1
        CALL Help
        CJNE Mode,#11H,TMd15        ;Test if Data Mode
        JMP Mod11
```

```
TMd15:  CJNE Mode,#15H,TMd1A        ;Test if SFR Mode
        JMP Mod15

TMd1A:  CJNE Mode,#19H,IMd1D        ;Test if Xdata Mode
        JMP Mod19

IMd1D:  JMP Mod1D                   ;Else Code Mode

;       Exe Key: Only active while in SS, BP or Exe running modes.

TExe1:  CJNE A,#1DH,IPCu1           ;Rest of keys
        MOV DPTR,#KyBrd
        MOVX A,@DPTR                ;Read Switches
        ANL A,#0C0H                 ;Mask in switches
        JZ IPCu1                    ;Ignore if no switches
        JMP SSExt                   ; else to SS & BP Exit

;*************************************************************
;       Edit Memory - Mode 1X
;*************************************************************

Mod1X:  MOV CurAd,#0C6H             ;Entry from Mod10, 4th Add key

;       Decode and Display SFR or Data Addresses

DsAll:  CALL ClDsp
        MOV A,Mode
        CLR Acc.0                   ;Both Bin & Hex mode
        CJNE A,#12H,T1SFR           ;Test if Data (both Bn & Hex)
        MOV A,@LoAdd                ;Get LoAdd Data
        PUSH Acc                    ;Save Data in stack
        MOV DPTR,#Msg11             ;'Data Add:      '
        MOV A,Mode
        JNB Acc.0,DMsg1             ;Jump if Hex Mode
        MOV A,#'D'                  ;Prepare to write 'D:'
        JMP P1C84

DMsg1:  CALL DspMg                  ;Display Msg11/15 for Hex
        MOV DPTR,#DspCm
        MOV A,#0C3H
        CALL WrDSP                  ;Place Cursor at C3 for Hex
        JMP DsLoA

T1SFR:  CJNE A,#16H,T1Xda           ;Test if SFR (both Bn & Hex)
        CALL RDSFR                  ;Read SFR Data
        PUSH Acc                    ;  and store in stack
        MOV DPTR,#Msg15             ;' SFR Add:      '
        MOV A,Mode
        JNB Acc.0,DMsg1             ;Jump if Hex Mode
        MOV A,#'R'                  ;Write 'R:'

P1C84:  MOV DPTR,#DspDa             ;Write 'D' or 'R' on 80,
        CALL WrDsp
        MOV A,#':'                  ; then ':'
        CALL WrDsp
        MOV DPTR,#DspCm
        MOV A,#84H
        CALL WrDsp                  ;Place Cursor at Dsp Add 84
        JMP DsLoA

;       Decode and Display Xdata or Code Addresses

T1Xda:  MOV DPL,LoAdd               ;DPTR=Hi & LoAdd for X: & C:
        MOV DPH,HiAdd
        CJNE A,#1AH,I1Cod           ;Test if Xdata (both Bn & Hx)
        MOVX A,@DPTR                ;Read Xdata
```

```
          PUSH Acc                  ; and store
          MOV DPTR,#Msg19           ;'Xdat Add:       '
          MOV A,Mode
          JNB Acc.0,DsMs1           ;Jump if Hex Mode
          MOV A,#'X'                ;Prepare to write 'X:'
          JMP P1C82

DsMs1:    CALL DspMg                ;Display Msg19/1D for Hex
          MOV DPTR,#DspCm
          MOV A,#0C1H
          CALL WrDsp                ;Place cursor at Add C1
          JMP DsHiA

I1Cod:    ORL DPH,#80H              ;Select Code Xdata Addres
          MOVX A,@DPTR              ;Read Code Data
          PUSH Acc
          MOV DPTR,#Msg1D
          MOV A,Mode
          JNB Acc.0,DsMs1           ;Jump if Hex Mode
          MOV A,#'C'                ;Write 'C:' to Display

P1C82:    MOV DPTR,#DspDa
          CALL WrDsp
          MOV A,#':'
          CALL WrDsp
          MOV DPTR,#DspCm
          MOV A,#82H
          CALL WrDsp                ;Place Cursor at Dsp Add 82

          ;Display Hi Address

DsHiA:    MOV DPTR,#DspDa
          MOV A,HiAdd
          SWAP A                    ;Hi HiAdd in Acc LSN
          CALL BnASC
          CALL WrDsp                ;Write High Nibble Character
          MOV A,HiAdd
          CALL BnASC
          CALL WrDsp                ;Write Low Nibble Character

;         Display Lo Address

DsLoA:    MOV DPTR,#DspDa
          MOV A,LoAdd
          SWAP A                    ;Hi LoAdd in Acc LSN
          CALL BnASC
          CALL WrDsp                ;Write High Nibble Character
          MOV A,LoAdd
          CALL BnASC
          CALL WrDsp                ;Write Low Nibble Character
          MOV A,#' '
          CALL WrDsp                ;Write a Space
          MOV A,Mode
          JB Acc.0,DsBnD            ;Jump if Binary Mode

;         Display Data in Hex mode

          POP Acc                   ;Get Data
          PUSH Acc                  ;Resave for Lo Nibble
          SWAP A
          CALL BnASC
          CALL WrDsp                ;Write Hi Data Nibble
          POP Acc
          CALL BnASC
          CALL WrDsp                ;Write Lo Data Nibble
          JMP P11Cu                 ;To Place Cursor
```

```
;       Display Data in Binary Mode

Wr1:    PUSH Acc
        MOV A,#'1'                  ;Character '1'
        JMP Roll

DsBnD:  MOV DPTR,#DspCm
        MOV A,#0C0H
        CALL WrDsp                  ;Cursor at Dsp Add C0 for Bin
        MOV DPTR,#DspDa             ;Set up DPTR for Data
        POP Acc
        MOV B,#08H                  ;Set up B as a Bit Counter
NxRol:  JB Acc.7,Wr1                ;Test if MSB is "1"
        PUSH Acc
        MOV A,#'0'                  ;Character '0'

Roll:   CALL WrDsp
        POP Acc
        RL A                        ;Next bit to Acc MSB
        DJNZ B,NxRol

;       Return point for cursor movement keys

P11Cu:  MOV DPTR,#DspCm
        MOV A,CurAd                 ;Place Cursor
        CALL WrDsp

        CALL ReadK

        CJNE A,#1DH,T1Esc           ;Test if Exe Key
        MOVX A,@DPTR                ;DPTR still set from ReadK
        ANL A,#0C0H                 ;Mask in switches only
        JZ P11Cu                    ;Ignore for no Switches
        JMP SSExt                   ;Else SS & BP Exit

T1Esc:  CJNE A,#1CH,T1Lft           ;Test if Esc Key
        JMP Mod00

;       Cursor Left in Hex Mode

T1Lft:  CJNE A,#10H,T1Rgt           ;Test if Left Key
        MOV A,Mode
        JB Acc.0,T1CuA              ;Jump if Binary Mode
        CJNE CurAd,#0C6H,T1XC1      ;Test if on left Data digit
        MOV CurAd,#0C4H             ; to right Add digit if it is
        JMP P11Cu

T1XC1:  CJNE CurAd,#0C1H,T1XC3      ;Test if on left Add digit
        JMP P11Cu                   ; ignore if it is

T1XC3:  CJNE CurAd,#0C3H,Dc1CA      ;Test if on 3rd Add digit

I1XC3:  JNB Acc.3,P11Cu             ;Acc still Mode. Jmp Dat/SFR

;       Cursor Left in Binary Mode

Dc1CA:  DEC CurAd                   ;Dec for CurAd=C7,C4,C2,(C3)
        JMP P11Cu

T1CuA:  CJNE CurAd,#0C0H,T1X82      ;Left Data digit in Bin Mode
        MOV CurAd,#85H              ;To last of Add digits
        JMP P11Cu

T1X82:  CJNE CurAd,#82H,T1X84       ;On Left of Add digits
        JMP P11Cu
```

```
T1X84:  CJNE CurAd,#84H,Dc1CA       ;On 3rd Add digit
        JMP I1XC3

;       Cursor Right key

T1Rgt:  CJNE A,#12H,T1Hlp           ;Test if Right Key
        CJNE CurAd,#85H,T1XC7       ;Test if on right Add digit
        MOV CurAd,#0C0H             ; To 1st Data digit (Bn Mode)
        JMP P11Cu

T1XC7:  CJNE CurAd,#0C7H,T1XC4      ;Test if on right Data digit
        JMP P11Cu

T1XC4:  CJNE CurAd,#0C4H,In1Cu      ;Test if on Rgt Add digit(Hex)
        MOV A,Mode
        JB Acc.0,In1Cu              ;Jmp if Bin Mode
        MOV CurAd,#0C6H             ;Else to 1 Data in Hex Mode
        JMP P11Cu

In1Cu:  INC CurAd                   ;Inc for CA=83,84,85,C1-C6
IP11C:  JMP P11Cu                   ;Intermediate Long Jump

;       Other Keys:

T1Hlp:  CJNE A,#14H,T1Dwn           ;Test if Help Key
        MOV DPTR,#Help1
        CALL Help
IMd1X:  JMP DsAll

T1Dwn:  CJNE A,#11H,T1Up            ;Test if Down Key
        INC 01H                     ;Inc LoAdd
        CJNE LoAdd,#00H,IMd1X       ;SFR Add corrected in Mod1X
        INC 02H                     ;Inc HiAdd on roll over
        JMP DsAll

T1Up:   CJNE A,#15H,THxBn           ;Test if Up Key
        DEC 01H                     ;Dec LoAdd
        CJNE LoAdd,#0FFH,IMd1X      ;SFR Add corrected in Mod1X
        DEC 02H                     ;Dec HiAdd on roll over
        JMP DsAll

THxBn:  CJNE A,#17H,T1Mem           ;Test if HxBn Key
        XRL 00H,#01H                ;Compliment Mode.0
        MOV A,Mode
        JB Acc.0,I1XC0              ;Jmp if now Bin Mode
        MOV CurAd,#0C6H             ;CurAd=C6 for Hex Mode
        JMP DsAll

I1XC0:  MOV CurAd,#0C0H             ;CurAd=C0 for Bn Mode
        JMP DsAll

;       New Memory Segment key.

T1Mem:  MOV B,A                     ;Store Key
        ANL A,#0FCH                 ;18,19,1A,1B -> 18
        CJNE A,#18H,T1Key           ;Test if a Mem Key
        JMP Mod10

;       '0-F' Keys

T1Key:  JB B.4,IP11C                ;Return for rest of keys
        MOV A,Mode                  ; B still holds key
        JNB Acc.0,RpAds
        JMP TCaBn                   ;Jmp if Bin Mode
```

```
;       Replace Add digits in Hex Mode

RpAds:  MOV A,B                     ;Key to Acc
        CJNE CurAd,#0C1H,T1YC2      ;Test if CurAd on Add 1

RpHAH:  SWAP A                      ;Key in Acc MSN
        ANL 02H,#0FH                ;Clear HiAdd Hi
        ORL 02H,A                   ;Write Key
        INC CurAd
        JMP DsAll

T1YC2:  CJNE CurAd,#0C2H,T1YC3      ;Test if CurAd on Add 2

RpHAL:  ANL 02H,#0F0H               ;Clear HiAdd Lo
        ORL 02H,A                   ;Write Key
        INC CurAd
        JMP DsAll

T1YC3:  CJNE CurAd,#0C3H,T1YC4      ;Test if CurAd on Add 3

RpLAH:  SWAP A                      ;Key in Acc MSN
        ANL 01H,#0FH                ;Clear LoAdd Hi
        ORL 01H,A                   ;Write Key
        INC CurAd
        CJNE Mode,#16H,TMd17        ;Test if SFR Hx Mode
SMSBT:  ORL 01H,#80H                 ;Set MSB for Add 80-FF

IDsAl:  JMP DsAll

TMd17:  CJNE Mode,#17,IDsAl         ;Test if SFR Bn Mode
        JMP SMSBT

T1YC4:  CJNE CurAd,#0C4H,T1YC6      ;Test if CurAd on Add 4

RpLAL:  ANL 01H,#0F0H               ;Clear LoAdd Lo
        ORL 01H,A                   ;Write Key
        INC CurAd
        INC CurAd
        JMP DsAll

;       Replace Data Digits in Hex Mode

T1YC6:  CJNE CurAd,#0C6H,I1YC7      ;Test if CurAd on Data 1
        MOV CurAd,#0C7H             ;Key = B
        MOV A,Mode                  ;Mode in Acc for GtDat RpHDa:
        CALL GtDat                  ;Old Data in A
        ANL A,#0FH                  ;Clear Hi Nibble
        SWAP A
        ORL A,B                     ;Write Key
        SWAP A                      ;New Data in Hi Nibble

StDat:  MOV B,A                     ;Data in B
        MOV A,Mode
        CLR Acc.0                   ;Same for Bn or Hex
        CJNE A,#12H,TXSFR           ;Test if Data Segment
        MOV @R1,B                   ;Store Data
        JMP DsAll

TXSFR:  CJNE A,#16H,TXXda           ;Test if SFR Segment
        MOV A,B                     ;New data in Acc for WRSFR
        CALL WRSFR
        JMP DsAll

TXXda:  CJNE A,#1AH,IXCod           ;Test if Xdata Segment
        MOV A,B
        MOV DPL,LoAdd
```

```
          MOV DPH,HiAdd
          MOVX @DPTR,A              ;Write data
          JMP DsAll

IXCod:    MOV A,B
          MOV DPL,LoAdd
          MOV DPH,HiAdd
          ORL DPH,#80H              ;Set Code Write Add
          MOVX @DPTR,A              ;Write data
          JMP DsAll

I1YC7:    MOV CurAd,#0C6H           ;Set cursor. B=Key
          MOV A,Mode                ;Mode in Acc for GtDat RpLDa:
          CALL GtDat
          ANL A,#0F0H               ;Clear Low Nibble
          ORL A,B                   ;New Data in Acc
          JMP StDat

;         Replace Address digit in Binary Mode

TCaBn:    MOV A,B                   ;Key in Acc and in B
          CJNE CurAd,#82H,T1Z83     ;Test if Add 1
          JMP RpHAH                 ;Cursor Inc in RpHAH

T1Z83:    CJNE CurAd,#83H,T1Z84     ;Test if Add 2
          JMP RpHAL

T1Z84:    CJNE CurAd,#84H,T1Z85     ;Test if Add 3
          JMP RpLAH

T1Z85:    CJNE CurAd,#85H,T1ZCn     ;Test if Add 4
          MOV CurAd,#0C0H           ;Cursor to Data digit 1
          JMP RpLAL

;         Replace Data bit in Binary Mode

T1ZCn:    CLR Acc.0
          JNZ Not01                 ;Jump if Key not "0" or "1"
          MOV A,Mode                ;Mode in Acc for GtDat
          CALL GtDat                ;Return with Acc = old data
          MOV C,B.0                 ;Key "1" or "0" to Flag 0
          MOV F0,C
          MOV B,#08H                ;Set up B as bit counter
NxtRl:    CJNE CurAd,#0C0H,RLACC    ;Test if CurAd is on MSB
          MOV C,F0                  ;Flag to Carry
          RLC A                     ;Replace MSB with Key
DcZCa:    Dec CurAd
          DJNZ B,NxtRl
          MOV B,A                   ;New data in B
          MOV A,CurAd
          ADD A,#09H                ;Restore CurAd + 1
          CLR Acc.3                 ;Make CurAd C8 -> C0
          MOV CurAd,A
          MOV A,B                   ;Data in Acc for StDat
          JMP StDat

RLACC:    RL A
          JMP DcZCa

;         Replace data digit if key not '0' or '1'

Not01:    ANL 03H,#0FCH             ;CuAdd: C0-C3>C0 C4-C7>C4
          XRL 03H,#00000100B        ;C4->C0 and C0->C4
          CJNE CurAd,#0C4H,RpLDa    ;Lo Dat if cursor not on C4
          JMP RpHDa                 ; else Hi if cursor was on C4
;*************************************************************
```

```
;*************************************************************
;       Some Relavent Subroutines
;*************************************************************
;       Write Data to Variable SFR Subroutine
;*************************************************************

;       Enter: Acc = Data, LoAdd(R1) = SFR Add
;       Exit: SFR data replaced
;       Corrupt: DPTR, Acc, C
;       Subs & Stack: 5 = 3 (WProt 2)

WrSFR:  CJNE LoAdd,#0D0H,TWSF2      ;Test PSW
        JMP WSFRX

TWSF2:  CJNE LoAdd,#80H,TWSF3       ;Test P0
        JMP WSFRX

TWSF3:  CJNE LoAdd,#81H,TWSF4       ;Test SP
        JMP WSFRX

TWSF4:  CJNE LoAdd,#0A0H,WSFR       ;Test P2
        JMP WSFRX

WSFR:   PUSH Acc
        SETB C
        CALL WProt                  ;Unprotect Code Mem
        MOV A,LoAdd
        MOV DPTR,#$+800BH           ;Code Write Add
        MOVX @DPTR,A                ;Write New Code ipo 04H
        CLR C
        CALL WProt                  ;Write Protect Code Mem
        POP Acc
        MOV 04H,A                   ;04H was replaced by SFR Add
WSFRX:  RET

;*************************************************************
;       Read Data from Variable SFR Subroutine
;*************************************************************

;       Enter: LoAdd(R1) = SFR Add
;       Exit: Acc=Data
;       Corrupt: DPTR, Acc, C
;       Subs & Stack: 4 = 2 (WProt 2)

RdSFR:  SETB C
        CALL WProt                  ;Unprotect Code Mem
        MOV A,LoAdd
        MOV DPTR,#$+8009H           ;Code Write Add
        MOVX @DPTR,A
        CLR C
        CALL WProt                  ;Write Protect Code Mem
        MOV A,04H
        RET

;*************************************************************
```

```
;*************************************************************
;       Read Data from Selected Memory Segment Subroutine
;*************************************************************

;       Enter: Segment in Acc: 12=D, 16=SFR, 1A=X, 1E=Code
;              Address in (HiAdd) & LoAdd [(R2) & R1]
;       Exit: Data in Acc
;       Corrupt: DPTR, Acc, C
;       Subs & Stack: 6 = 2 (RdSFR 2 (WProt 2))

GtDat:  MOV DPH,HiAdd
        MOV DPL,LoAdd
        MOV A,Mode
        CLR Acc.0                   ;For Hex OR Bin Modes
        CJNE A,#12H,GtSFR           ;Test if Data
        MOV A,@R1                   ;Read Data
        RET

GtSFR:  CJNE A,#16H,GtXda           ;Test SFR
        CALL RdSFR
        RET

GtXda:  CJNE A,#1AH,GtCod           ;Test if Xdata
        MOVX A,@DPTR
        RET

GtCod:  ORL DPH,#80H                ;Code Xdata Address
        MOVX A,@DPTR                ;Read Code
        RET

;*************************************************************
```

## 10.9. Mode 2: Edit Pre-Interrupt Registers

This mode will be entered after the execution of every user instruction executed in the *Single Step Mode*, when a Break Point Match has been found in the *Break Point Mode*, or when the 'Escape' key has been pressed in the real time *Execute Mode*. It allows the user to inspect and alter the user program values of the SFR registers and the internal data memory at the point of the break. Only those registers and the data bytes that would be corrupted by the Operating System will be stored in the external memory storage area. They are directly accessible in this mode. The other SFRs and internal data memory can be accessed by the normal "Memory Edit Mode". Should any value be changed in the "Pre-Interrupt Registers", the new values will be loaded into the registers before the next user instruction is executed. Even the program counter may be changed to allow the user to start executing at a new point in his program.

The current Pre-interrupt register values and all the other system variables may be stored by the "Save Data & System Variables" of Mode 3. A user may then start again at the same point and with exactly the same system variables and settings after, for example, a system crash or a power-off break. The stored values can be re-loaded by using the "Restore Data & System Variables" of Mode 4, after a reset or power up.

The sequence of the Pre-Interrupt registers as they would appear on the screen of the display, and the external addresses where they are stored on the external memory map, are shown on the next page:

| Registers: Digits | Pre-Int.Storage | Bulk Storage |
|---|---|---|
| "Prog.Cntr: dd dd" | F043 F042 | F0C3 F0C2 |
| " PSW ACC: dd dd" | F045 F044 | F0C5 F0C4 |
| " DPH DPL: dd dd" | F047 F046 | F0C7 F0C6 |
| " SP BReg: dd dd" | F049 F048 | F0C9 F0C8 |
| " IP IE : dd dd" | F04B F04A | F0CB F0CA |
| "TCON TMOD: dd dd" | F04D F04C | F0CD F0CC |
| " TH0 TL0: dd dd" | F04F F04E | F0CF F0CE |
| " TH1 TL1: dd dd" | F051 F050 | F0D1 F0D0 |
| "D:00=dd dd dd dd" | F052 - F055 | F0D2 - F0D5 |
| "D:04=dd dd dd dd" | F056 - F059 | F0D6 - F0D9 |
| "D:08=dd dd dd dd" | F05A - F05D | F0DA - F0DD |
| "D:0C=dd dd dd dd" | F05E - F061 | F0DE - F0E1 |
| "D:10=dd dd dd dd" | F062 - F065 | F0E2 - F0E5 |
| "D:14=dd dd dd dd" | F066 - F069 | F0E6 - F0E9 |

("dd" = hexadecimal digits)

When the 'Exe' key is pressed from this or any other menu option, these values will be restored in the registers and the next user instruction will be executed. If there is no immediate reaction from the 'Exe' key, the 'Esc' key must be pressed first, then the 'Exe' key. The Running Mode Selection Switches must be set to select any one of the Running Modes.

The Operating Diagram, the Flow Diagram and the Assembler Listing of the software routines for Mode 2 are shown on the next few pages:

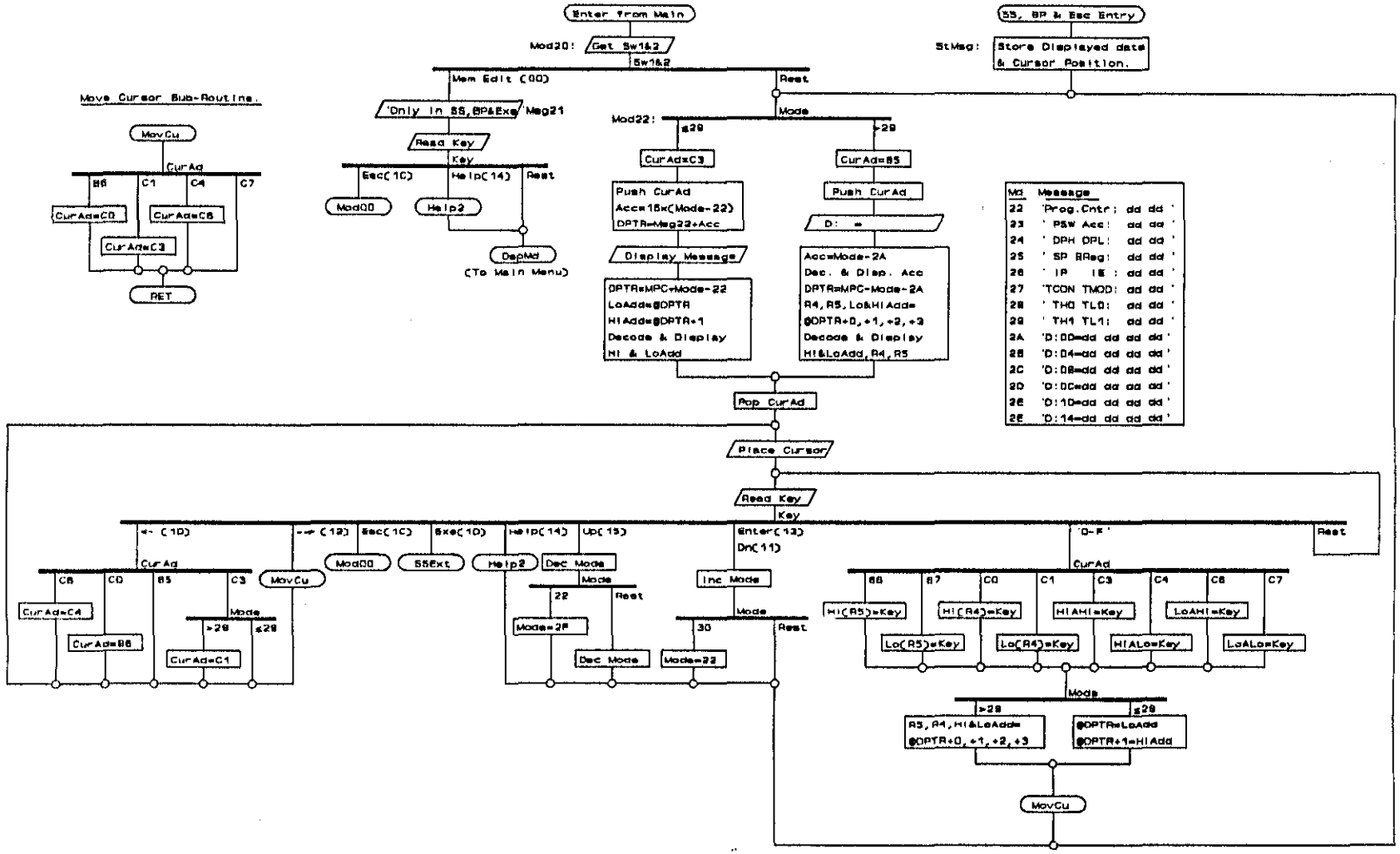



**Fig.10.11. Edit Pre-Interrupt Registers Operating Diagram**

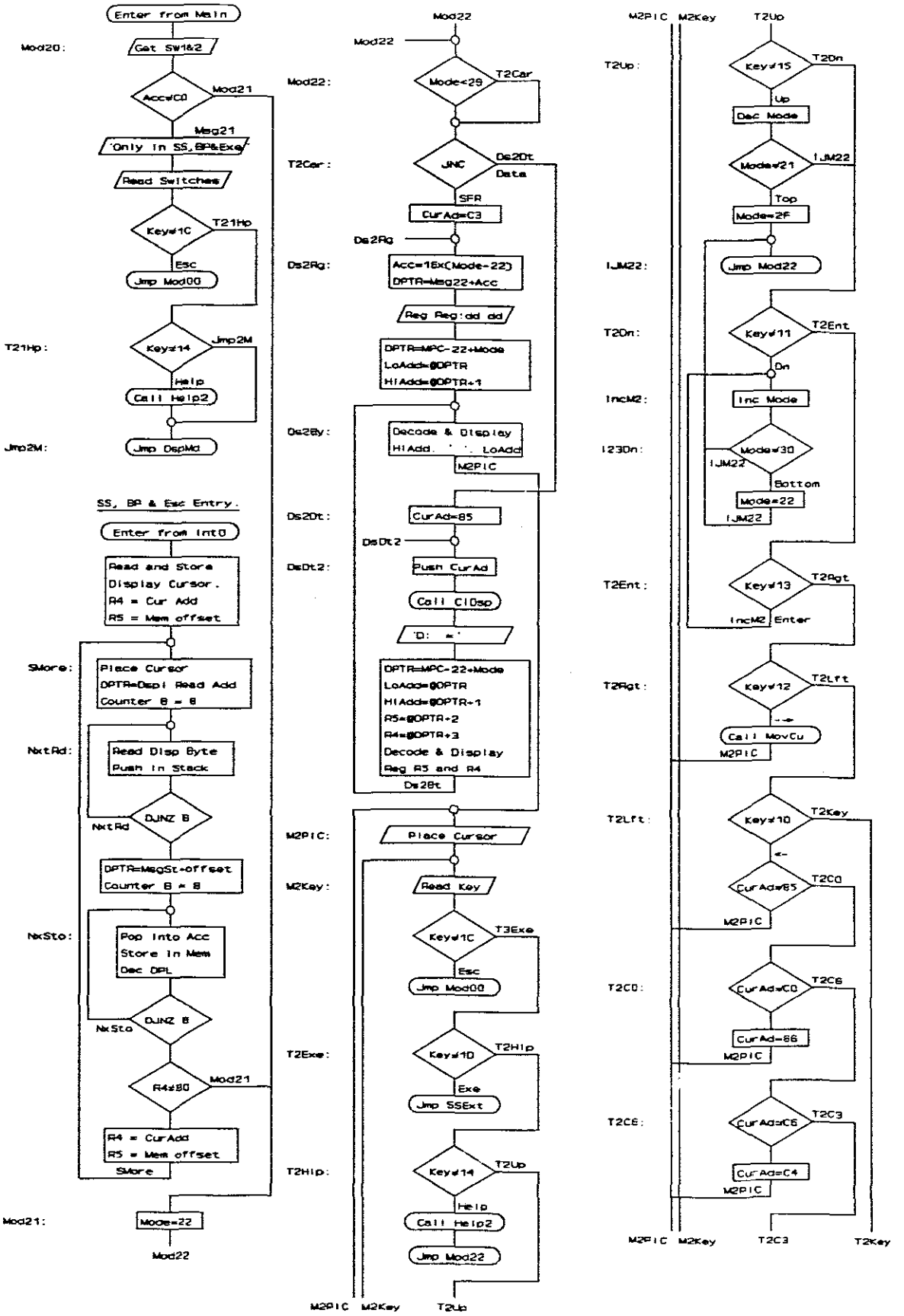



**Fig.10.12. Edit Pre-Interrupt Registers Flow Diagram** (page 1 of 2)

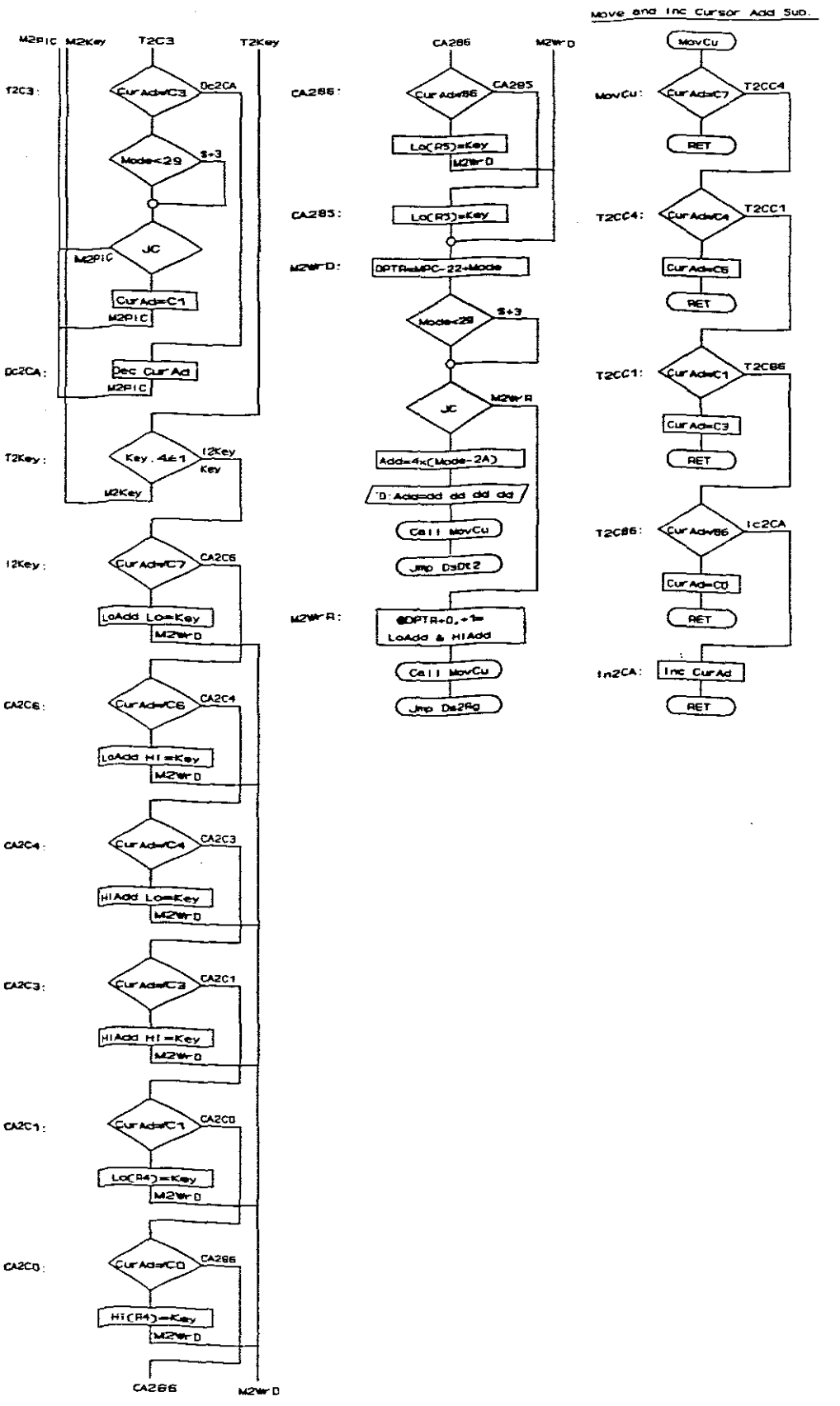



**Fig. 10.13. Edit Pre-Interrupt Registers Flow Diagram** (page 2 of 2)

## Assembler Listing

```
;*************************************************************
;       Edit Pre-Interrupt Registers & Data - Mode 20
;*************************************************************

Mod20:  MOV DPTR,#KyBrd
        MOVX A,@DPTR                ;Read Switches
        ANL A,#0C0H                 ;Mask in Switches
        JNZ Mod21                   ;Jmp if none set
        MOV DPTR,#Msg21             ;'Only in SS&BP Md'
        CALL DspMg
        CALL ReadK
        CJNE A,#1CH,T21Hp           ;Test if Esc Key
        Jmp Mod00

T21Hp:  CJNE A,#14H,Jmp2M           ;Test if Help Key
        MOV DPTR,#Help2
        CALL Help
Jmp2M:  JMP DspMd                   ;Rest, to Main Menu

;       Store Display Data and Cursor Address

StMsg:  MOV DPTR,#DspAd             ;Entry from SS, BP or Exe Esc
        MOVX A,@DPTR                ;Read Address from Display
        CALL WSHRT
        SETB Acc.7                  ;Ready for restore
        MOV DPTR,#DsCSt             ;Display Cursor Store
        MOVX @DPTR,A
        MOV R4,#80H                 ;Setup for 1st 8 bytes
        MOV R5,#07H                 ;Setup for MsgSt offset

SMore:  MOV DPTR,#DspCm
        MOV A,R4
        CALL WrDsp
        MOV DPTR,#DspRd             ;Ready to read data
        MOV B,#08H                  ;Counter = 8

NxtRd:  MOVX A,@DPTR                ;Read Display Byte
        CALL WShrt
        PUSH Acc                    ; and store in stack
        DJNZ B,NxtRd

        MOV DPTR,#MsgSt             ;Message Store in Memory
        MOV A,R5
        ADD A,DPL                   ;Add offset
        MOV DPL,A
        MOV B,#08H

NxSto:  POP Acc
        MOVX @DPTR,A                ;Store bytes in reverse order
        DEC DPL
        DJNZ B,NxSto
        CJNE R4,#80H,StDsp          ;Exit after second read
        MOV R4,#0C0H                ;2nd line Cursor Add
        MOV R5,#0FH                 ;2nd offset
        JMP SMore

StDsp:  MOV DPTR,#DspCm
        MOV A,#06H                  ;Inc Cursor, display steady
        CALL WrDsp
        MOV A,#0DH                  ;Display ON, cursor char blink
        CALL WrDsp
```

```
Mod21:  MOV Mode,#22H               ;Point to PC  22-2A
Mod22:  CJNE Mode,#2AH,$+3          ;C=0 if Mode>29H = data
        JNC Ds2Dt                   ;C=1 for Registers

;       Display Registers, Hi & Lo bytes

        MOV CurAD,#0C3H             ;Left on Left digit
Ds2Rg:  MOV A,Mode                  ;Calculate Message Ref
         ADD A,#0DEH                 ;Subtract 22H
        SWAP A                      ;x16
        MOV DPTR,#Msg22             ;DPTR=Msg22+16x(Mode-22)
        ADD A,DPL
        MOV DPL,A
        CLR A
        ADDC A,DPH
        MOV DPH,A
        CALL DspMg

        MOV A,Mode                  ;Set up to Get Registers
        ADD A,#0DEH                 ;Acc=Mode-22H
        RL A                        ;Acc=2x(Mode-22H)
        MOV DPTR,#MPC               ;Register store base add
        ADD A,DPL
        MOV DPL,A
        CLR A
        ADDC A,DPH
        MOV DPH,A

        MOVX A,@DPTR
        MOV LoAdd,A
        INC DPTR
        MOVX A,@DPTR
        MOV HiAdd,A

Ds2By:  MOV DPTR,#DspCm             ;Decode & Disp Hi&LoAdd
        MOV A,#0C3H
        CALL WrDsp                  ;Place cursor on C3
        MOV DPTR,#DspDa
        MOV A,HiAdd
        SWAP A                      ;Hi Nibble
        CALL BnASC
        CALL WrDsp
        MOV A,HiAdd                 ;Lo Nibble
        CALL BnASC
        CALL WrDsp
        MOV A,#' '                  ;space
        CALL WrDsp

        MOV A,LoAdd
        SWAP A                      ;Hi Nibble
        CALL BnASC
        CALL WrDsp
        MOV A,LoAdd                 ;Lo Nibble
        CALL BnASC
        CALL WrDsp
        JMP M2PlC                   ;Jmp to Place Cursor

;       Display Internal Data as: 'D:aa=dd dd dd dd'
;                          Stored in: R0 R5 R4 R2 R1
Ds2Dt:  MOV CurAd,#85H
DsDt2:  CALL ClDsp                  ;Clear display
        MOV A,Mode                  ;Translate Mode to D:Add
        ADD A,#0D6H                 ;Mode-2AH
        RL A                        ;(Mode-2A)x4
        RL A
        PUSH Acc
```

```
Mod21:  MOV Mode,#22H               ;Point to PC  22-2A
Mod22:  CJNE Mode,#2AH,$+3          ;C=0 if Mode>29H = data
        JNC Ds2Dt                   ;C=1 for Registers

;       Display Registers, Hi & Lo bytes

        MOV CurAD,#0C3H             ;Left on Left digit
Ds2Rg:  MOV A,Mode                  ;Calculate Message Ref
         ADD A,#0DEH                 ;Subtract 22H
        SWAP A                      ;x16
        MOV DPTR,#Msg22             ;DPTR=Msg22+16x(Mode-22)
        ADD A,DPL
        MOV DPL,A
        CLR A
        ADDC A,DPH
        MOV DPH,A
        CALL DspMg

        MOV A,Mode                  ;Set up to Get Registers
        ADD A,#0DEH                 ;Acc=Mode-22H
        RL A                        ;Acc=2x(Mode-22H)
        MOV DPTR,#MPC               ;Register store base add
        ADD A,DPL
        MOV DPL,A
        CLR A
        ADDC A,DPH
        MOV DPH,A

        MOVX A,@DPTR
        MOV LoAdd,A
        INC DPTR
        MOVX A,@DPTR
        MOV HiAdd,A

Ds2By:  MOV DPTR,#DspCm             ;Decode & Disp Hi&LoAdd
        MOV A,#0C3H
        CALL WrDsp                  ;Place cursor on C3
        MOV DPTR,#DspDa
        MOV A,HiAdd
        SWAP A                      ;Hi Nibble
        CALL BnASC
        CALL WrDsp
        MOV A,HiAdd                 ;Lo Nibble
        CALL BnASC
        CALL WrDsp
        MOV A,#' '                  ;space
        CALL WrDsp

        MOV A,LoAdd
        SWAP A                      ;Hi Nibble
        CALL BnASC
        CALL WrDsp
        MOV A,LoAdd                 ;Lo Nibble
        CALL BnASC
        CALL WrDsp
        JMP M2P1C                   ;Jmp to Place Cursor

;       Display Internal Data as: 'D:aa=dd dd dd dd'
;                          Stored in: R0 R5 R4 R2 R1
Ds2Dt:  MOV CurAd,#85H
DsDt2:  CALL ClDsp                  ;Clear display
        MOV A,Mode                  ;Translate Mode to D:Add
        ADD A,#0D6H                 ;Mode-2AH
        RL A                        ;(Mode-2A)x4
        RL A
        PUSH Acc
```

```
        MOV DPTR,#DspDa         ;Data store base add
        MOV A,#'D'
        CALL WrDsp
        MOV A,#':'
        CALL WrDsp
        POP Acc                 ;Data store offset
        PUSH Acc
        SWAP A
        CALL BnASC              ;Hi Nibble Add
        CALL WrDsp
        POP Acc
        PUSH Acc                ;To use again
        CALL BnAsc
        CALL WrDsp              ;Lo Nibble Add
        MOV A,#'='
        CALL WrDsp

        POP Acc                 ;Acc=4x(Mode-2AH)
        MOV DPTR,#MIDat         ;Mem Data Reference
        ADD A,DPL               ;Add offset in Acc
        MOV DPL,A
        CLR A
        ADDC A,DPH
        MOV DPH,A

        MOVX A,@DPTR
        MOV R5,A                ;1st Data to R5
        INC DPTR
        MOVX A,@DPTR
        MOV R4,A                ;2nd Data to R4
        INC DPTR
        MOVX A,@DPTR
        MOV HiAdd,A             ;3rd Data to R2
        INC DPTR
        MOVX A,@DPTR
        MOV LoAdd,A             ;4th Data to R1

        MOV DPTR,#DspDa
        MOV A,R5
        SWAP A                  ;Hi Nibble, 1st data byte
        CALL BnAsc
        CALL WrDsp
        MOV A,R5                ;Lo Nibble
        CALL BnAsc
        CALL WrDsp
        MOV DPTR,#DspCm         ;Set cursor for 2nd data byte
        MOV A,#0C0H
        CALL WrDsp
        MOV DPTR,#DspDa
        MOV A,R4
        SWAP A                  ;Hi Nibble, 2nd data byte
        CALL BnAsc
        CALL WrDsp
        MOV A,R4                ;Lo Nibble
        CALL BnAsc
        CALL WrDsp
        JMP Ds2By               ;To display other 2 bytes

;       Return point for key functions

M2PlC:  MOV DPTR,#DspCm         ;Place cursor at CurAd
        MOV A,CurAd
        CALL WrDsp
```

```
M2Key:   CALL ReadK

         CJNE A,#1CH,T2Exe          ;Test if Esc Key
         JMP Mod00

T2Exe:   CJNE A,#1DH,T2Hlp          ;Test if Exe Key
         JMP SSExt

T2Hlp:   CJNE A,#14H,T2Up           ;Test if Help Key
         MOV DPTR,#Help2
         CALL Help
         JMP Mod22                  ;Redisplay all

T2Up:    CJNE A,#15H,T2Dn           ;Test if Up Key
         DEC Mode
         CJNE Mode,#21H,IJM22       ;Test if on top of list
         MOV Mode,#2FH

IJM22:   JMP Mod22                  ;Intermediate jump

T2Dn:    CJNE A,#11H,T2Ent          ;Test if Down Key
IncM2:   INC Mode
         CJNE Mode,#30H,IJM22       ;Test if at bottom of list
         MOV Mode,#22H
         JMP Mod22

T2Ent:   CJNE A,#13H,T2Rgt          ;Test if Enter Key
         JMP IncM2                  ;Same result as Down Key

T2Rgt:   CJNE A,#12H,T2Lft          ;Test if Right Key
         CALL MovCu
         JMP M2PlC

T2Lft:   CJNE A,#10H,T2Key          ;Test if Left Key
         CJNE CurAd,#85H,T2C0
         JMP M2PlC
T2C0:    CJNE CurAd,#0C0H,T2C6
         MOV CurAd,#86H
         JMP M2PlC

T2C6:    CJNE CurAd,#0C6H,T2C3
         MOV CurAd,#0C4H
         JMP M2PlC

T2C3:    CJNE CurAd,#0C3H,Dc2CA     ;Dec for other values
         CJNE Mode,#2AH,$+3         ;C=0 if Mode>29H = data
         JC M2PlC                   ;C=1, Reg, no change
         MOV CurAd,#0C1H            ; else to C1 for data
         JMP M2PlC

Dc2CA:   DEC CurAd                  ;Dec for rest of Cursor adds
         JMP M2PlC

T2Key:   JNB Acc.4,I2Key            ;Test if '0-F' Key
         JMP M2Key                  ;Too far for JB M2Key

I2Key:   CJNE CurAd,#0C7H,CA2C6
         ANL 01H,#0F0H              ;Clear LoAdd Lo
         ORL 01H,A                  ;Write Key
         JMP M2WrD

CA2C6:   CJNE CurAd,#0C6H,CA2C4
         ANL 01H,#0FH               ;Clear LoAdd Hi
         SWAP A
         ORL 01H,A                  ;Write Key
         JMP M2WrD
```

```
CA2C4:  CJNE CurAd,#0C4H,CA2C3
        ANL 02H,#0F0H           ;Clear HiAdd Lo
        ORL 02H,A               ;Write Key
        JMP M2WrD

CA2C3:  CJNE CurAd,#0C3H,CA2C1
        ANL 02H,#0FH            ;Clear HiAdd Hi
        SWAP A
        ORL 02H,A               ;Write Key
        JMP M2WrD

CA2C1:  CJNE CurAd,#0C1H,CA2C0
        ANL 04H,#0F0H           ;Clear R4 Lo
        ORL 04H,A               ;Write Key
        JMP M2WrD

CA2C0:  CJNE CurAd,#0C0H,CA286
        ANL 04H,#0FH            ;Clear R4 Hi
        SWAP A
        ORL 04H,A               ;Write Key
        JMP M2WrD
CA286:  CJNE CurAd,#86H,CA285
        ANL 05H,#0F0H           ;Clear R5 Lo
        ORL 05H,A               ;Write Key
        JMP M2WrD

CA285:  ANL 05H,#0FH            ;Clear R5 Hi
        SWAP A
        ORL 05H,A               ;Write Key

M2WrD:  CJNE Mode,#2AH,$+3      ;C=0 if Mode>29H = data
        JC M2WrR                ;C=1, Registers


;       Write new Internal Data back to store

        MOV A,Mode              ;Calculate offset
        SUBB A,#2AH             ;Mode-2A = zero ref
        RL A                    ;x4
        RL A
        MOV DPTR,#MIDat         ;Mem Data Reference
        ADD A,DPL
        MOV DPL,A
        CLR A
        ADDC A,DPH
        MOV DPH,A
        MOV A,R5
        MOVX @DPTR,A            ;Write R5
        INC DPTR
        MOV A,R4
        MOVX @DPTR,A            ;Write R4
        INC DPTR
        MOV A,HiAdd
        MOVX @DPTR,A            ;Write HiAdd
        INC DPTR
        MOV A,LoAdd
        MOVX @DPTR,A            ;Write LoAdd
        CALL MovCu              ;Cursor one digit right
        JMP DsDt2
```

```
;       Write new Register values back to store

M2WrR:  MOV A,Mode                  ;Calculate the Reg offset
        ADD A,#0DEH                 ;Mode-22
        RL A                        ;x2
        MOV DPTR,#MPC               ;Mem PC Reference
        ADD A,DPL
        MOV DPL,A
        CLR A
        ADDC A,DPH
        MOV DPH,A

        MOV A,LoAdd
        MOVX @DPTR,A                ;Write LoAdd
        INC DPTR
        MOV A,HiAdd
        MOVX @DPTR,A                ;Write HiAdd
        CALL MovCu
        JMP Ds2Rg

;*************************************************************
;       Move or Increment Cursor Add for Mode 2 Subroutine
;*************************************************************

;       Corrupt: CurAd, 2 stack Bytes

MovCu:  CJNE CurAd,#0C7H,T2CC4
        RET

T2CC4:  CJNE CurAd,#0C4H,T2CC1
        MOV CurAd,#0C6H
        RET

T2CC1:  CJNE CurAd,#0C1H,T2C86
        MOV CurAd,#0C3H
        RET

T2C86:  CJNE CurAd,#86H,In2CA
        MOV CurAd,#0C0H
        RET

In2CA:  Inc CurAd                   ;Rest of positions
        RET

;*************************************************************
```

## 10.10. Mode 3 & 4: Store and Retrieve Internal Data & System Variables

When the power to the board is switched off, all the internal data and SFR values of the micro-processor will be destroyed. The "Smart Socket" battery backed up Code and Data RAM ICs, however, will preserve their contents during a power break. To conserve the power dissipation from the onboard battery, a three minute auto-off timer was built into the software: If no button is pressed for three minutes while in the Operating System, the internal data and SFRs of the controller will be stored in the Code RAM IC, and the power will be switched off automatically. This was included so that if a user is testing a program and some interference, such as a front door or telephone call, cause the user to leave the system for too long, it will switch the system off automatically. To prevent a few hours work being destroyed, the current settings, before the switch off, will be stored and can be retrieved at any later stage.

A small extension to the power off storage allows a user to also store all the current settings of the system at any point. He can then restart again at a specific point in a user program, without having to re-run the program right from the beginning. Say one discovers there is an error somewhere in the program that creates a system crash and the system can only be recovered by a reset: A break point can be inserted just before the suspect routine. When the control is over to the Operating System after the break point, all the current settings are stored using Mode 3. They can even be dumped to a PC for disk storage and later reloaded back into the system. Now the user can Single Step or run through the suspect routine to try to find the error. If the system crashes, the settings at the

last storage point can be retrieved with Mode 4. Then the user can try another strategy to find the error. He does not have to go through the entire program again from the beginning.

With this storage facility, a user can also interrupt his program at any point, store the current settings on disk, and test an entire new program. At a later stage, the first program and settings can be reloaded, and the user can continue from exactly the same point where the break occurred.

The bulk storage of the internal data and system variables is only initiated by one of the following: When selected by the user to do so in Mode 3; when the three minute auto-off period has expired; or when the user selects the "OFF" option of Mode F. The normal system variables occupy a different block of memory than the bulk storage area. The normal usage of the Operating System will therefore not corrupt these values.

The Operating Diagrams, the Flow Diagrams and the Assembler listings of the software routines for Modes 3 and 4 are included in the next few pages:

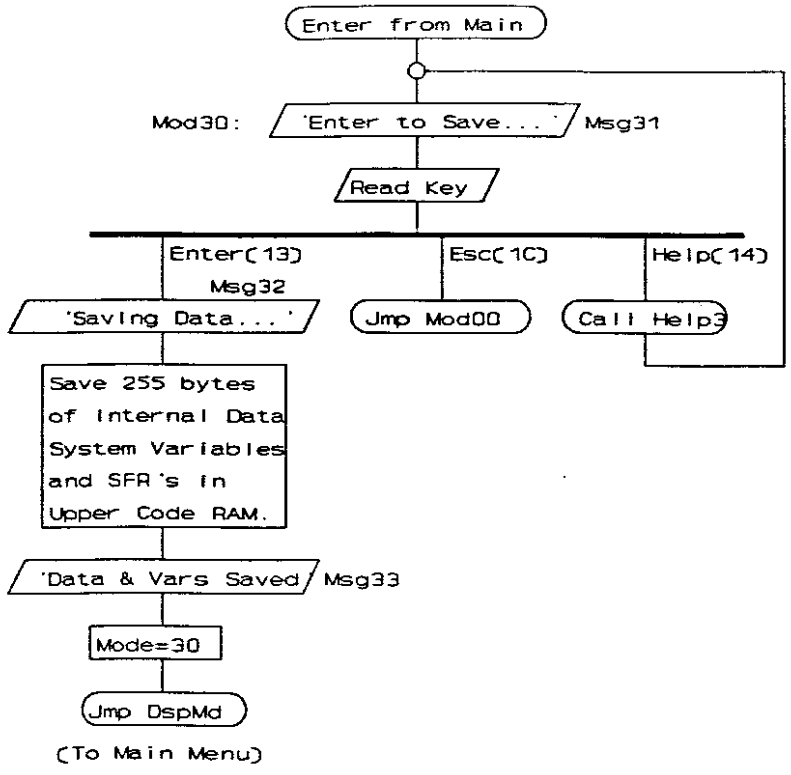



**Fig. 10.14. Save Data and System Variables Operating Diagram**

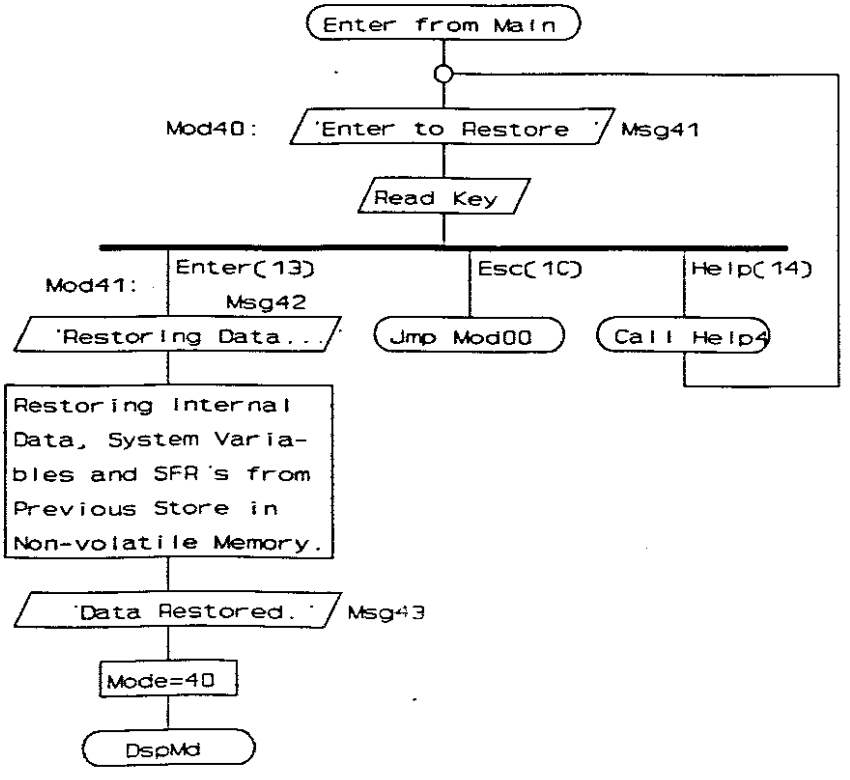



**Fig. 10.15. Retrieve Data and System Variables Operating Diagram**

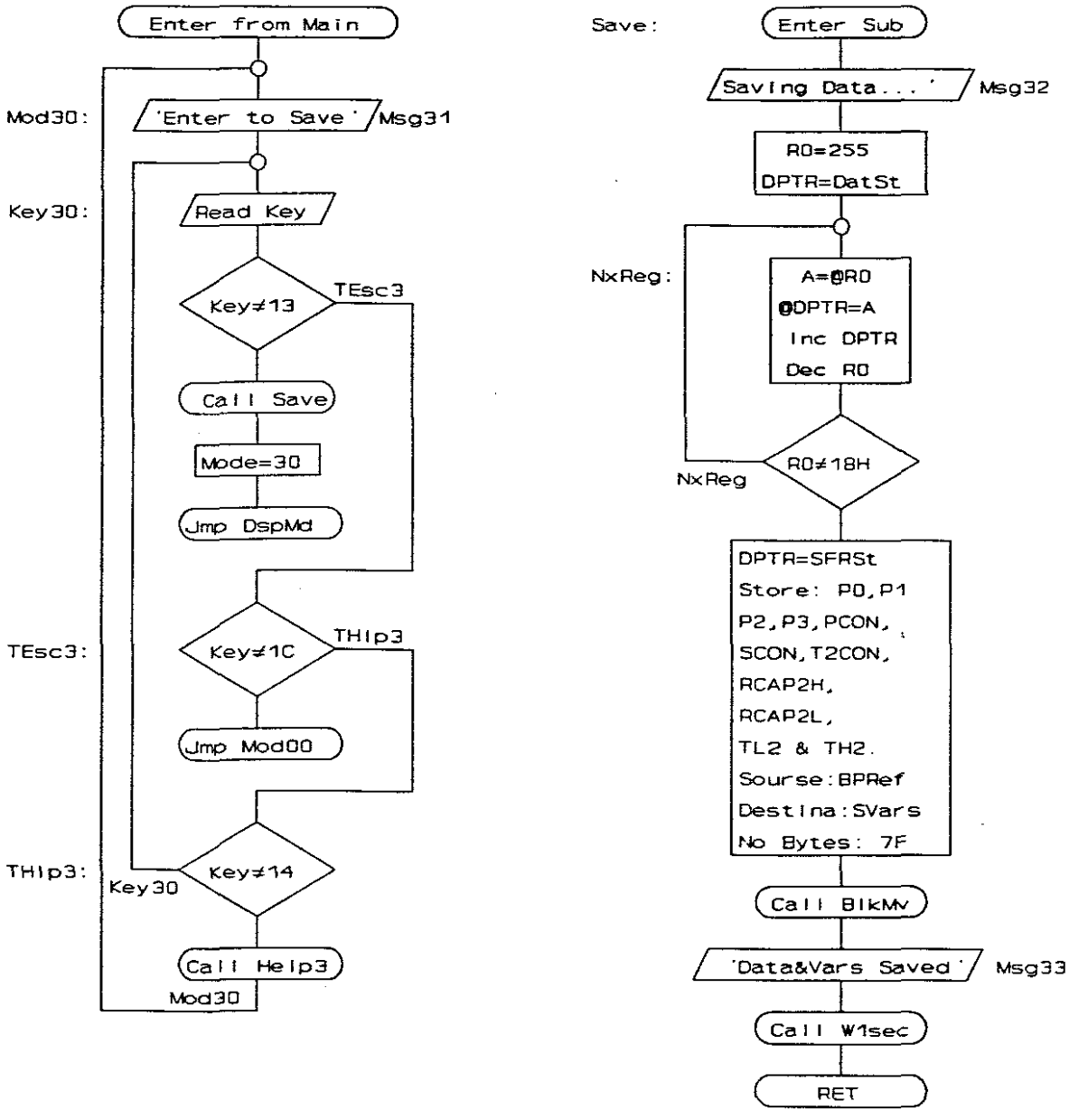



**Fig. 10.16. Save Data and System Variables Flow Diagram**

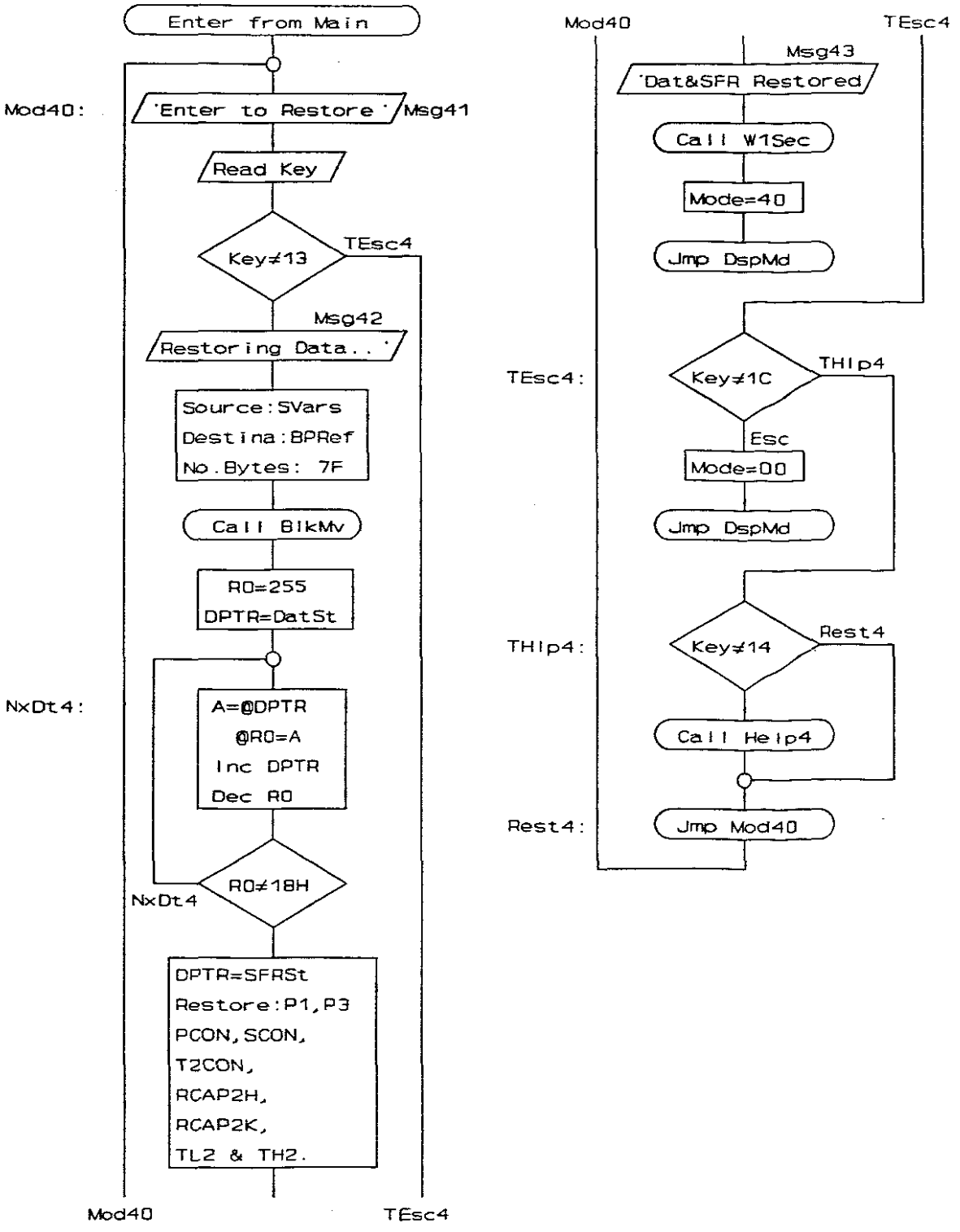



**Fig. 10.17. Retrieve Data and System Variables Flow Diagram**

```
;*************************************************************
;       Save Internal Data & System Variables - Mode 30
;*************************************************************

Mod30:  MOV DPTR,#Msg31             ;'Enter to Save'
        CALL DspMg
Key30:  CALL ReadK
        CJNE A,#13H,TEsc3           ;Test if Enter Key
Mod31:  CALL SAVE
        MOV Mode,#30H               ;Restore Mode
        JMP DspMd                   ;To Main Menu

TEsc3:  CJNE A,#1CH,THlp3           ;Test if Esc Key
        JMP Mod00

THlp3:  CJNE A,#14H,Key30           ;Test if Help Key
        MOV DPTR,#Help3
        CALL Help
        JMP Mod30

;*************************************************************
;       Save Internal Data and SFR Subroutine
;*************************************************************

;       Corrupt: DPTR, Acc, C, Tm0, R0, EX0=1
;       Subs & Stack: 10 = 2 (DspMg 4 (WrDsp 2 (T0Int 2))),
;                      6 = 2 (W1Sec 2 (T0Int 2)),
;                      8 = 4 (BlkMv 4)

SAVE:   MOV DPTR,#Msg32             ;'Saving Data....'
        CALL DspMg
        CALL W1Sec

        MOV R0,#0FFH                ;Counter = FFH
        MOV DPTR,#DatSt             ;Start of Data Storage

NxReg:  MOV A,@R0                   ;Store Internal Data
        MOVX @DPTR,A
        INC DPTR
        DEC R0
        CJNE R0,#18H,NxReg          ;Down to 18H

        MOV DPTR,#SFRSt             ;Start of SFR Storage
        MOV A,P0                    ;Port 0
        MOVX @DPTR,A
        INC DPTR

        MOV A,P1                    ;Port 1
        MOVX @DPTR,A
        INC DPTR

        MOV A,P2                    ;Port 2
        MOVX @DPTR,A
        INC DPTR

        MOV A,P3                    ;Port 3
        MOVX @DPTR,A
        INC DPTR

        MOV A,PCON                  ;PCON
        MOVX @DPTR,A
        INC DPTR
```

```
          MOV A,SCON                    ;SCON
          MOVX @DPTR,A

          INC DPTR
          MOV A,0C8H                    ;T2CON
          MOVX @DPTR,A

          INC DPTR
          MOV A,0CAH                    ;RCAP2H
          MOVX @DPTR,A
          INC DPTR

          MOV A,0CBH                    ;RCAP2L
          MOVX @DPTR,A
          INC DPTR

          MOV A,0CCH                    ;TL2
          MOVX @DPTR,A
          INC DPTR

          MOV A,0CDH                    ;TH2
          MOVX @DPTR,A

;         Store Current System Variables

          PUSH 00H                      ;Store LoAdd still in R0
          PUSH 02H                      ; and HiAdd
          MOV DPTR,#SVars               ;Destination Add
          MOV R0,DPL                    ; in R3,R0
          MOV R3,DPH
          MOV R1,#07FH                  ;Byte Counter - 1
          MOV R2,#00H
          MOV DPTR,#BPRef               ;Source Add for Sys Vars

          CALL BlkMv

          POP 02H                       ;Recover Lo & HiAdd
          POP 01H

          MOV DPTR,#Msg33               ;'Data&Vars Saved'
          CALL DspMg

          CALL W1Sec

          RET

;*************************************************************
;         Restoring Data & System Variables - Mode 40
;*************************************************************

Mod40:    MOV DPTR,#Msg41               ;'Enter to Restore'
          CALL DspMg
          CALL ReadK
          CJNE A,#13H,TEsc4             ;Test if Enter Key
          MOV DPTR,#Msg42               ;'Restoring Data..'
          CALL DspMg

;         Restore System Variables and Pre-Interrupt Registers

          MOV DPTR,#BPRef               ;Destination Address
          MOV R0,DPL                    ; in R3,R0
          MOV R3,DPH
          MOV R1,#07FH                  ;Counter (128 Bytes - 1)
          MOV R2,#00H                   ; in R2,R1
          MOV DPTR,#SVarS               ;DPTR = Source Address
          CALL BlkMv                    ;To Block Move (Up)
```

```
;       Restoring save Internal Data (231 bytes)

        MOV R0,#0FFH            ;Counter and Data Add
        MOV DPTR,#DatSt

NxDt4:  MOVX A,@DPTR            ;Get byte from Mem
        MOV @R0,A               ;Restore Internal Data
        INC DPTR
        DEC R0
        CJNE R0,#18H,NxDt4      ;Test if done (FF to 18)

;       Restoring saved SFR (11 bytes)

        MOV DPTR,#SFRSt+1       ;P1 in memory
        MOVX A,@DPTR
        MOV P1,A
        INC DPTR
        INC DPTR
        MOVX A,@DPTR
        MOV P3,A                ;Port 3

        INC DPTR
        MOVX A,@DPTR
        MOV PCON,A              ;PCON

        INC DPTR
        MOVX A,@DPTR
        MOV SCON,A              ;SCON

        INC DPTR
        MOVX A,@DPTR
        MOV 0C8H,A              ;T2CON

        INC DPTR
        MOVX A,@DPTR
        MOV 0CAH,A              ;RCAP2L

        INC DPTR
        MOVX A,@DPTR
        MOV 0CBH,A              ;RCAP2L

        INC DPTR
        MOVX A,@DPTR
        MOV 0CCH,A              ;TL2

        INC DPTR
        MOVX A,@DPTR
        MOV 0CDH,A              ;TH2

        MOV DPTR,#Msg43         ;'Data Restored!  '
        CALL DspMg
        MOV DPTR,#0E2B4H
        CALL WPSec              ;Wait 1.5 seconds
        MOV Mode,#40H           ;Restore Mode

DpMdI:  JMP DspMd               ;Intermediate Jump

TEsc4:  CJNE A,#1CH,TH1p4       ;Test if Esc Key
        JMP Mod00

TH1p4:  CJNE A,#14H,Rest4
        MOV DPTR,#Help4
        CALL Help
Rest4:  JMP Mod40

;*************************************************************
```

## 10.11. Mode 5: Execute a User Program in Real Time

This mode is one of the three options where a user program can be run after it has been loaded in form a PC, or keyed in through the "Edit Memory Segment" Mode. The other two modes are running a user program in the "Single Step" mode (Mode 6), or the "Break Point" mode (Mode 7). All three modes are essentially the same and they all exit into the execution of the user program through this mode.

The Execution Mode enables a user to execute a user program in real time to test hardware configurations, such as switch bounce and multiplexed displays. These cannot be tested successfully on a simulator or in one of the other slower modes. It is, however, possible to interrupt the normal execution of a user program by pressing the 'Escape' key. Such a break is considered to be a temporary suspension of the operation, and all the relevant SFR, internal data and system variables are stored before control is handed over to the Operating System. The user can then inspect the Pre-Interrupt Registers through Mode 2, or use any of the other Operating System options. When the 'Exe' key is pressed, all the stored SFR and internal data values will be reloaded back into the micro-controller, and the execution will continue with exactly the same settings it had before the break.

The default starting address of the user program at 0000h may also be changed before the execution of the user program is initiated, so that a user can start running any of his other modules or subroutines immediately. It must be pointed

out that when a user program is initiated to start running using Mode 5, it is assumed that the contents of the micro-controller registers can be any value. The essential register settings required must be included at the beginning of the user program. The exit routine into the user program for this mode is not the same routine that is used *after* a "Single Step", "Break Point" or 'Esc' key break return, that would reload previous SFR and internal Data values.

The method selected to exit the Operating System into the execution of a user program, is done with an unconventional "Return from Interrupt", instruction. This was chosen to enable the *Single Step* and *Break Point* running modes as well: The data book specifies that no interrupt will happen directly after any instruction accessing any of the interrupt control registers, or a "Return from Interrupt" instruction. At least one further instruction will be executed before the interrupt occurs. This principle is used by pushing the return address into the stack and by setting the Interrupt 0 flag by software just before an "Enable Interrupt 0" and the "Return from Interrupt" instructions. The execution will "return" to the address in the stack, execute one user instruction, and only then react on the interrupt request by jumping to the Interrupt 0 subroutine. When the *Execution* mode is selected, however, the interrupt flag will ***not*** be set before jumping to the user program.

The Operating Diagram, the Flow Diagram, and the Assembler listing of the software routines for Mode 5 are included on the next few pages:

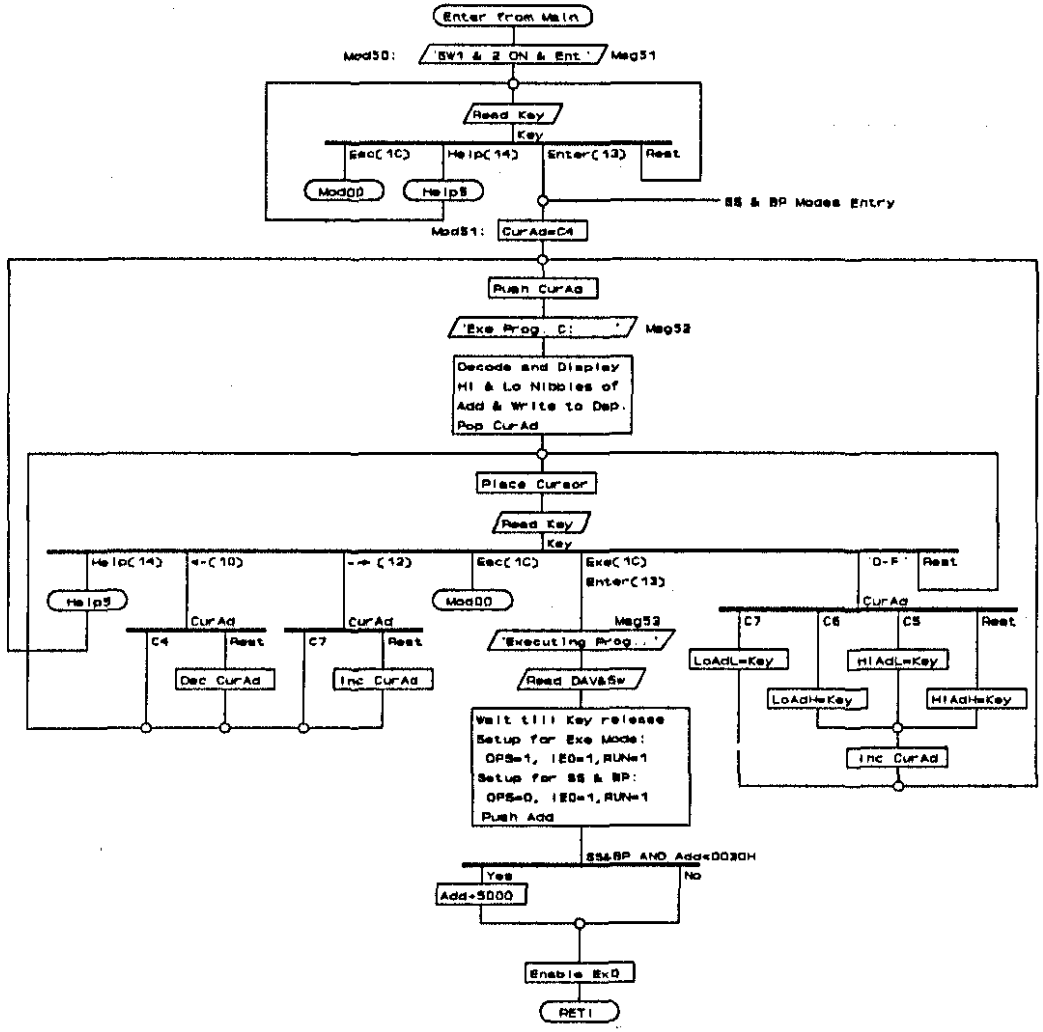



**Fig.10.18. Execute a User Program Mode Operating Diagram**

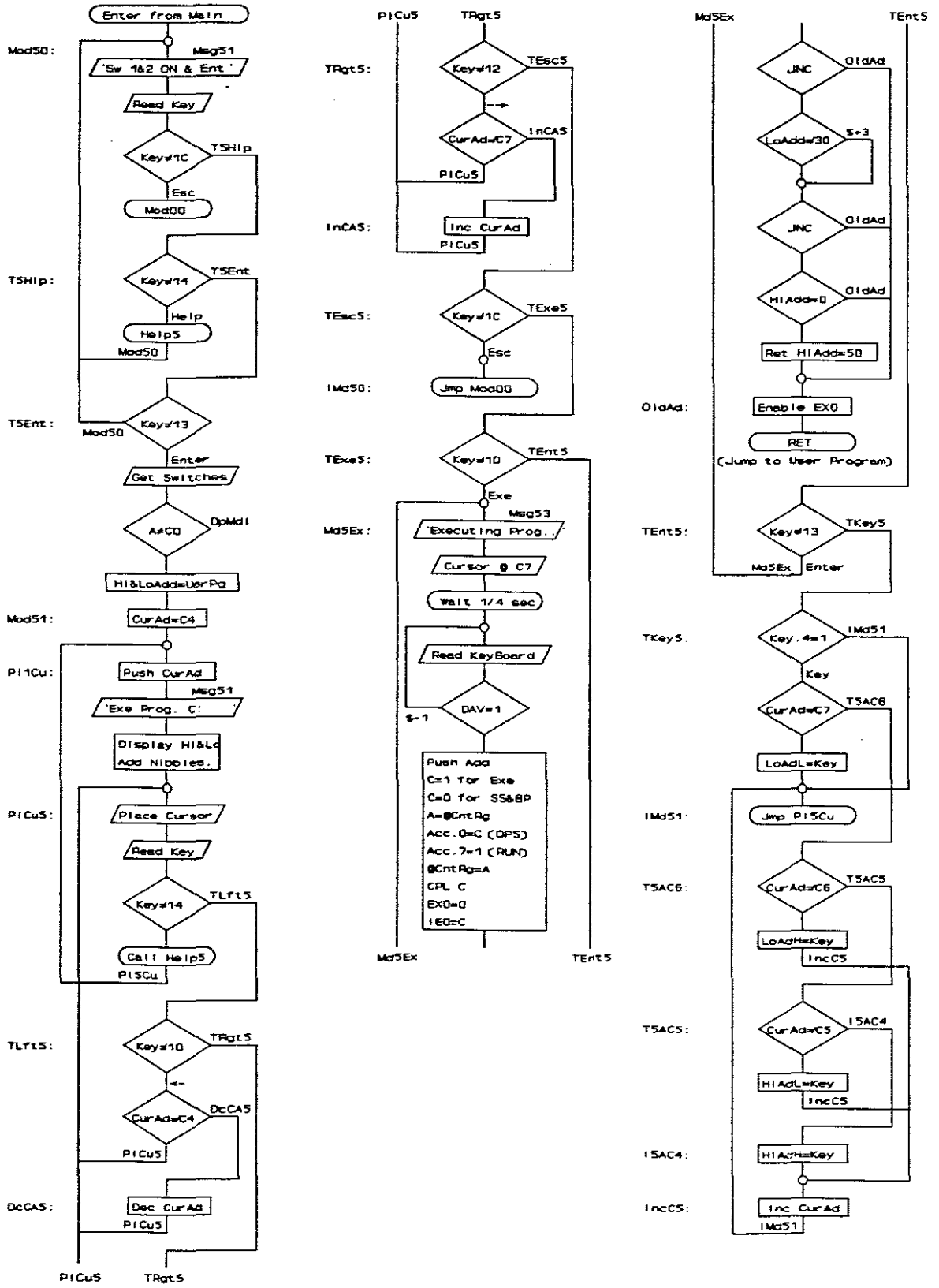



**Fig.10.19. Execute a User Program Mode Flow Diagram**

## Assembler Listing

```
;*************************************************************
;       Execute an User Program - Mode 50
;*************************************************************

Mod50:  MOV DPTR,#Msg51             ;'Sw 1 & 2 ON & Ent'
        CALL DspMg

        CALL ReadK

        CJNE A,#1CH,T5Hlp           ;Test if Esc Key
        JMP Mod00

T5Hlp:  CJNE A,#14H,T5Ent           ;Test if Help Key
        MOV DPTR,#Help5

        CALL Help

        JMP Mod50

T5Ent:  CJNE A,#13H,Mod50           ;Test if Enter Key
        MOV DPTR,#KyBrd
        MOVX A,@DPTR
        ANL A,#0C0H
        CJNE A,#0C0H,DpMdI          ;Test if switches set
        MOV DPTR,#UsrPg
        MOV HiAdd,DPH
        MOV LoAdd,DPL

;       Entry from SS & BP Modes

;       Display/Edit User Program Start Address

Mod51:  MOV DPTR,#Msg52             ;'Exe Prog. C:     '

        CALL Ent2B

        CJNE A,#1CH,Md5Ex           ;Test if Esc Key
        Jmp Mod00

Md5Ex:  MOV DPTR,#Msg53             ;'Executing Prog..'
        CALL DspMg

        MOV DPTR,#DspCm
        MOV A,#0C7H                 ;Place cursor on Disp Add C7
        CALL WrDsp

        MOV DPTR,#0FB1EH            ;¼ second
        CALL WPSec

        MOV DPTR,#KyBrd
        MOVX A,@DPTR
        JB Acc.5,$-1                ;Wait for key release (DAV=1)


        ;Prepare for User Program Execution

        PUSH 01H                    ;Push LoAdd &
        PUSH 02H                    ;HiAdd in stack for RET jmp
        MOV C,Acc.6
        ANL C,Acc.7                 ;C=1 for EXE mode
        MOV DPTR,#CntRg
        MOVX A,@DPTR
```

```
        MOV Acc.0,C             ;OPS=1 for EXE, 0 for SS&BP
        SETB Acc.7              ;EDIT=1 for all modes
        CLR Acc.1               ;Set Write Protect OPSYS
        MOVX @DPTR,A            ;Store and Reset IRQ Latch
        CPL C                   ;C=0 for EXE, 1 for SS&BP
        CLR EX0                 ;Disable Interrupt
        MOV IE0,C               ;IE0=1 to create SS interrupt

        JNC OldAd               ;Address stay as is for EXE
        CJNE LoAdd,#30H,$+3     ;Test if LoAdd < 30H
        JNC OldAd               ; and jump if not
        CJNE HiAdd,#00H,OldAd   ; or jump if HiAdd not 00H
        POP Acc                 ;Hi Return Address
        MOV A,#50H              ;Set Add for swapped vectors
        PUSH Acc                ;Restore in stack

OldAd:  SETB EX0                ;Enable SS, BP or Esc Int

        RETI                    ;Exe Program at stack ADD

;*************************************************************
```

## 10.12. Modes 6 and 7: Run a User Program in the "Single Step" or "Break Point" Modes

These two modes operate approximately in the same way, in that the user program is only allowed to execute one instruction step before it is interrupted. In the Single Step mode, the current SFR and internal data will be stored before the control is handed over to the Operating System. In the Break Point mode, the current program counter will be compared with up to thirty-two user definable addresses. Only if a match is found, will the current data be stored and control handed over to the Operating System. If *no* match is found, the next user program step will be executed directly.

To run a user program through the *Single Step Mode*, the screen will prompt the switching of the Running Mode Select Slide switches, before the Start Address can be changed and the execution initiated. Here, the Interrupt 0 flag will be set by the software before the "Enable Interrupt 0" and the "Return from Interrupt" instructions. One user instruction will be executed before the interrupt request will take effect. The Interrupt 0 subroutine of the Operating System will be discussed in detail later, but what it does in principle, is to test if another interrupt is pending and enabled. If so, it will store the interrupt vector as an additional return address in the stack, so that the next step will be the first instruction of the user interrupt subroutine. This enables a user to do single stepping through his own interrupt subroutines as well, without appreciable pre-requisites or interference from the Operating System.

The Break Point Table is not part of the user program. It is therefore not necessary to insert special instructions in the user program to enable a Break Point interrupt. The table is stored in the system variable area in the Code RAM. The Break Point Addresses need not be in numerical order. The system will compare all the Break Point values with the current user program counter value until either a match or a 0000h value is found.

When the *Break Point Mode* is selected, the user may inspect and alter the existing Break Point Table, or add new break points as required. To eliminate a Break Point halfway through the table, the user must alter the address to some impossible address that will never be reached, such as FFFFh. Placing a 0000h in that position will create an "End of Break Point Table", which is not desirable. Mode A will clear the entire Break Point Table and replace all the values with 0000h.

The Operating Diagram, the Flow Diagrams, and the Assembler Listings of the software routines for Modes 6 and 7 are included on the next few pages:

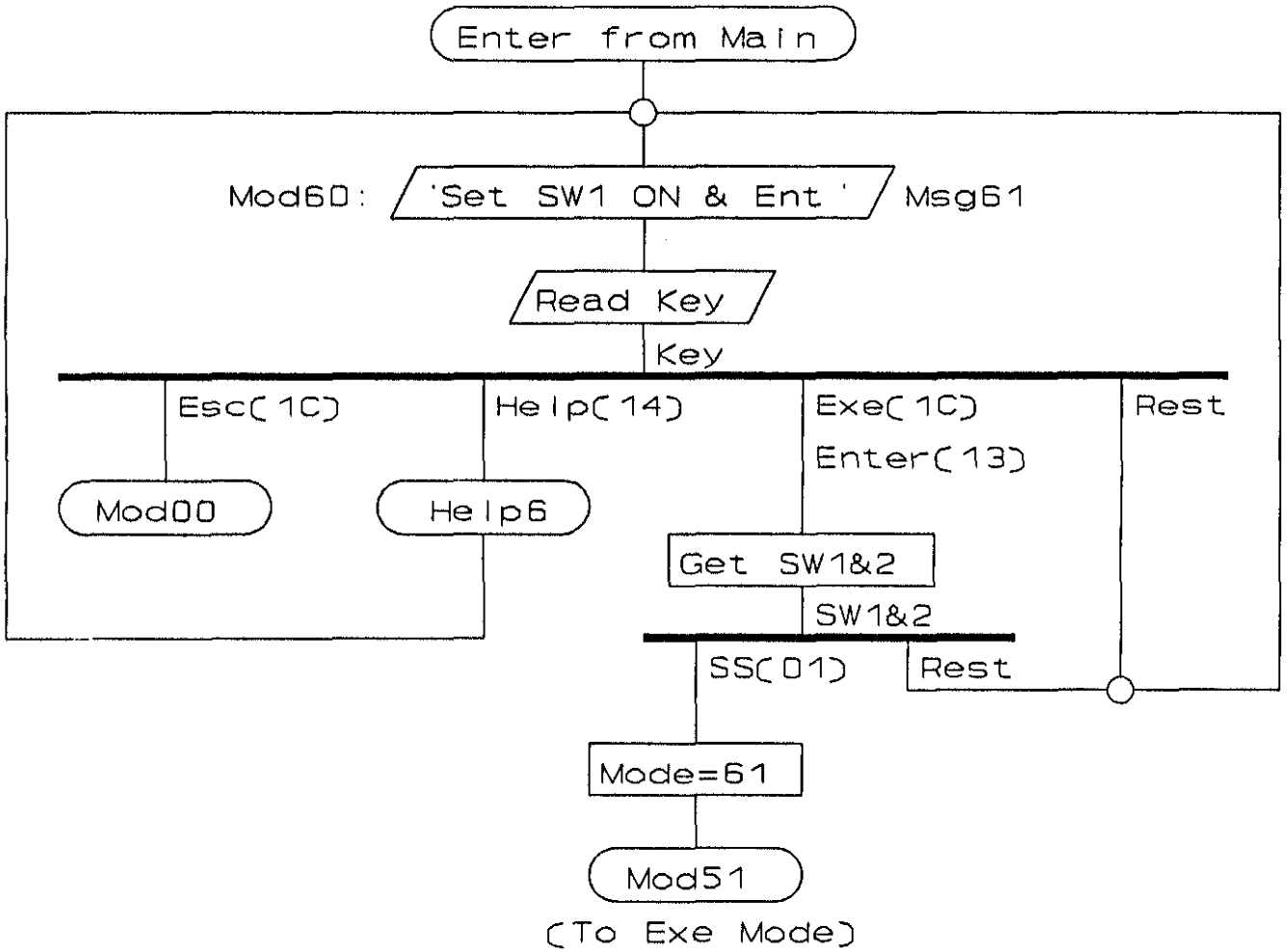



**Fig.10.20. Single Step Selection Operating Diagram**

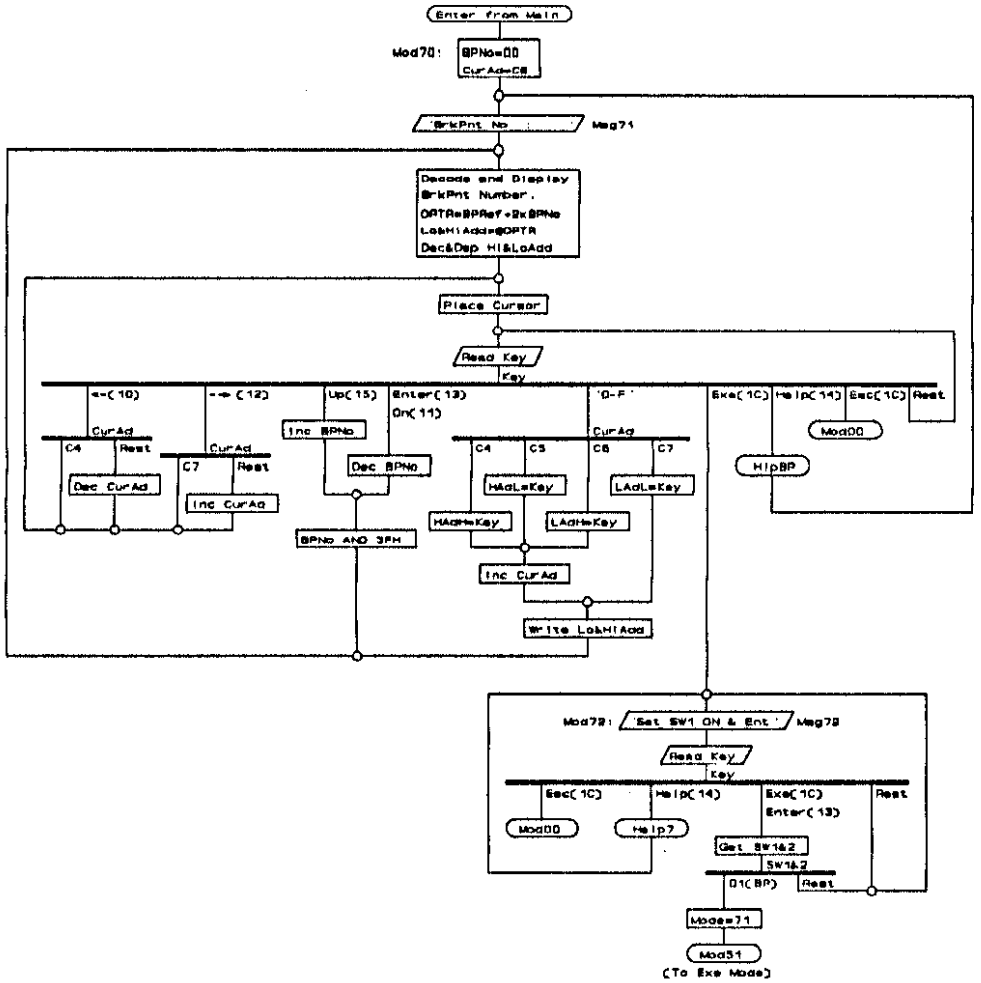



**Fig.10.21. Break Point Selection Operating Diagram**

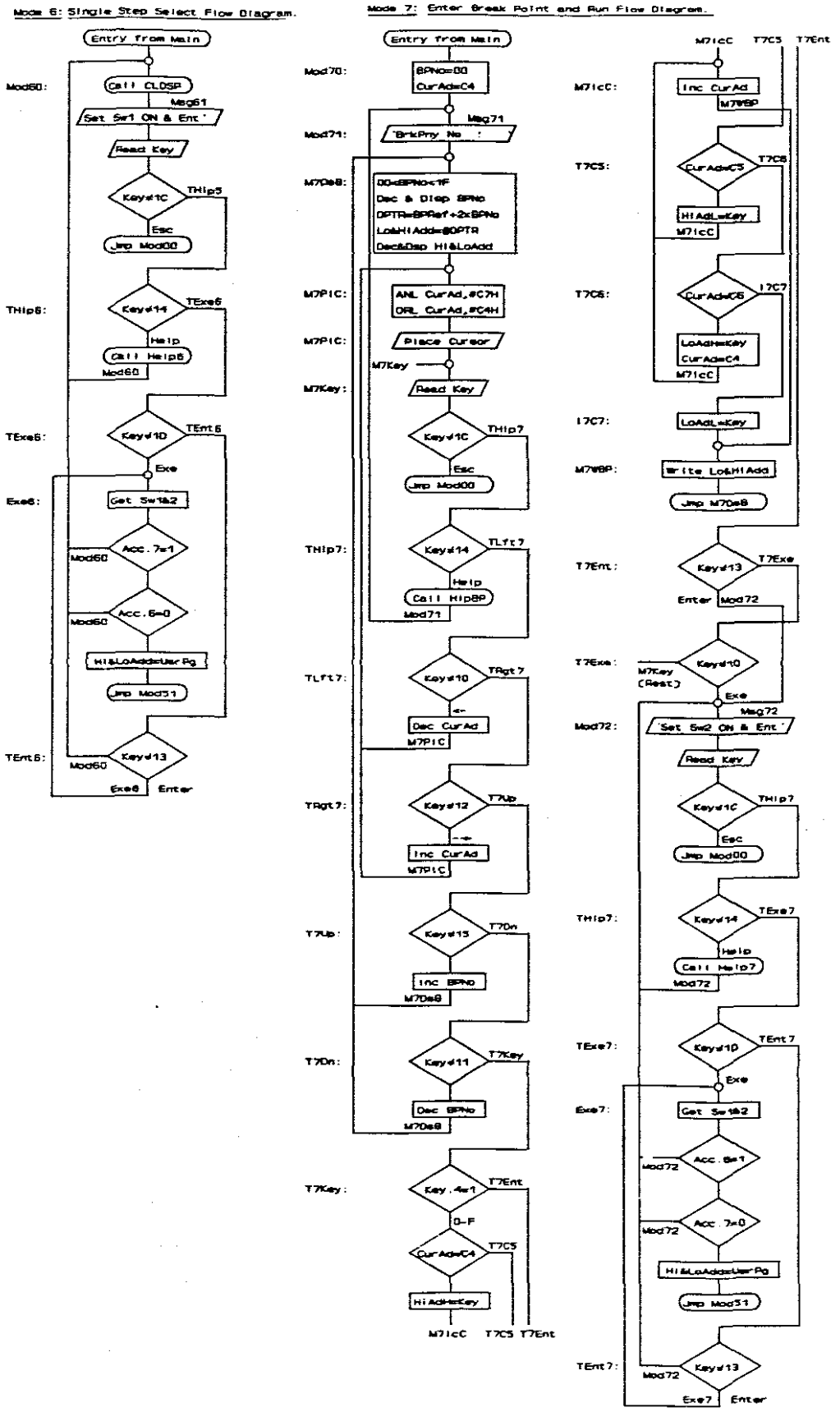



**Fig.10.22. Single Step & Break Point Selection Flow Diagrams**

## Assembler Listings

```
;*************************************************************
;        Select Single Step Execution - Mode 60
;*************************************************************

Mod60:   MOV DPTR,#Msg61            ;'Set Sw 1 ON & Ent'
         CALL DspMg
         CALL ReadK
         CJNE A,#1CH,THlp6          ;Test if Escape Key
         JMP Mod00

THlp6:   CJNE A,#14H,TExe6          ;Test if Help Key
         MOV DPTR,#Help6
         CALL Help
         JMP Mod60

TExe6:   CJNE A,#1DH,TEnt6          ;Test if Exe Key

Exe6:    MOV DPTR,#KyBrd
         MOVX A,@DPTR
         ANL A,#0C0H                ;Mask in Switches
         JNB Acc.6,IDpMd            ;Repeat if SS not set
         JB Acc.7,IDpMd             ;Repeat if BP set
         MOV DPTR,#UsrPg            ;Normal User Program Start
         MOV R1,DPL                 ; in Hi and LoAdd
         MOV R2,DPH
         JMP Mod51                  ;Go to Exe Mode to start

IDpMd:   JMP DspMd                  ;Intermediate jump

TEnt6:   CJNE A,#13H,Mod60          ;Test if Enter Key
         JMP Exe6                   ;Same action as Exe key

;*****************************************************************
;       Edit Break Points and Start Execution - Mode 70
;*****************************************************************

Mod70:   MOV Mode,#00H              ;R0=Mode=Break Point number
         MOV CurAd,#0C4H
Mod71:   MOV DPTR,#Msg71            ;'BrkPnt No  :    '
         CALL DspMg
M7DsB:   ANL 00H,#1FH               ;Round off after Inc or Dec
         MOV DPTR,#DspCm            ; 32 BPs  are between 00-1F
         MOV A,#0C1H                ; each BP is two bytes
         CALL WrDsp
         MOV DPTR,#DspDa
         MOV A,Mode                 ;BP number
         SWAP A
         CALL BnASC
         CALL WrDsp                 ;Write Hi Nibble
         MOV A,Mode
         CALL BnASC
         CALL WrDsp                 ;Write Lo Nibble
         MOV A,#':'                 ;Rewrite ':', and
         CALL WrDsp                 ; place Cursor on C4
         MOV DPTR,#BPRef            ;BP Reference in Mem
         MOV A,Mode
         RL A                       ;BP Number offset
         ADD A,DPL
         MOV DPL,A
         CLR A
         ADD A,DPH
         MOV DPH,A
         MOVX A,@DPTR               ;Get Lo BP
```

```
        MOV LoAdd,A
        INC DPTR
        MOVX A,@DPTR            ;Get Hi BP
        MOV HiADD,A
        MOV DPTR,#DspDa
        SWAP A
        CALL BnASC
        CALL WrDsp              ;HiBP Hi Nibble
        MOV A,HiAdd
        CALL BnASC
        CALL WrDsp              ;HiBP Lo Nibble
        MOV A,LoAdd
        SWAP A
        CALL BnASC
        CALL WrDsp              ;LoBP Hi Nibble
        MOV A,LoAdd
        CALL BnASC
        CALL WrDsp              ;LoBP Lo Nibble

M7PlC:  ANL 03H,#0C7H           ;Reset C8->C4 after left
        ORL 03H,#0C4H           ;Reset C3->C7 after right
        MOV DPTR,#DspCm
        MOV A,CurAd
        CALL WrDsp

;       Return point after cursor move functions

M7Key:  CALL Readk

        CJNE A,#1CH,T7Hlp       ;Test if Esc Key
        JMP Mod00

T7Hlp:  CJNE A,#14H,T7Lft       ;Test if Help Key
        MOV DPTR,#HlpBP
        CALL Help
        JMP Mod71

T7Lft:  CJNE A,#10H,T7Rgt       ;Test if Left Key
        DEC CurAd
        JMP M7PlC

T7Rgt:  CJNE A,#12H,T7Up        ;Test if Right Key
        INC CurAd
        JMP M7PlC

T7Up:   CJNE A,#15H,T7Dn        ;Test if Up Key
        DEC Mode                ;BP Number
        JMP M7DsB

T7Dn:   CJNE A,#11H,T7Key       ;Test if Down Key
        INC Mode                ;BP Number
        JMP M7DsB

T7Key:  JB Acc.4,T7Ent          ;Test if 0-F Key

;       Edit Break Points

        CJNE CurAd,#0C4H,T7C5   ;Test if was on 1st digit
        ANL 02H,#0FH            ;Blank Hi Nibble
        SWAP A
        ORL 02H,A               ;Replace Hi Nibble

M7IcC:  INC CurAd
        JMP M7WBP

T7C5:   CJNE CurAd,#0C5H,T7C6
```

```
        ANL 02H,#0F0H           ;Blank Lo Nibble
        ORL 02H,A               ;Replace Lo Nibble
        JMP M7IcC

T7C6:   CJNE CurAd,#0C6H,I7C7
        ANL 01H,#0FH            ;Blank Hi Nibble
        SWAP A
        ORL 01H,A               ;Replace Hi Nibble
        JMP M7IcC

I7C7:   ANL 01H,#0F0H           ;Blank Lo Nibble
        ORL 01H,A               ;Replace Lo Nibble
        MOV CurAd,#0C4H

M7WBP:  MOV DPTR,#BPRef
        MOV A,Mode
        RL A
        ADD A,DPL
        MOV DPL,A
        CLR A
        ADD A,DPH
        MOV DPH,A
        MOV A,LoAdd
        MOVX @DPTR,A            ;Write Lo BP
        INC DPTR
        MOV A,HiADD
        MOVX @DPTR,A            ;Write Hi BP
        JMP M7DsB

T7Ent:  CJNE A,#13H,T7Exe       ;Test if Enter Key
        JMP Mod72               ;Same action as Exe key

M7Iky:  JMP M7Key               ;No action from rest of keys
T7Exe:  CJNE A,#1DH,M7IKy       ;Test if Exe Key
Mod72:  MOV DPTR,#KyBrd
        MOVX A,@DPTR
        ANL A,#0C0H             ;Mask in Switches
        JZ Mod73                ;Go to SSExt if
        JMP SSExt               ; switches already set

Mod73:  MOV DPTR,#Msg72         ;'Set SW2 ON & Ent'
        CALL DspMg
        CALL ReadK
        CJNE A,#1CH,THlp7       ;Test if Esc Key
        JMP Mod00

THlp7:  CJNE A,#14H,TExe7       ;Test if Help Key
        MOV DPTR,#Help7
        CALL Help
        JMP Mod73

TExe7:  CJNE A,#1DH,TEnt7       ;Test if Exe Key

Exe7:   MOV DPTR,#KyBrd
        MOVX A,@DPTR            ;Read Switches
        JNB Acc.7,Mod73         ;Repeat if BP not set
        JB Acc.6,Mod73          ;Repeat if SS set
        MOV Mode,#71H           ;Set Mode.0 for BP
        MOV DPTR,#UsrPg         ;Normal User Program start
        MOV R1,DPL
        MOV R2,DPH
        JMP Mod51               ;Go to Exe Mode to start
TEnt7:  CJNE A,#13H,Mod72       ;Test if Enter Key
        JMP Exe7                ;Same action as Exe key

;*************************************************************
```

## 10.13. Modes 8, 9 and A: Bulk clearing of Internal Data, the User Program Area and the Break Point Table

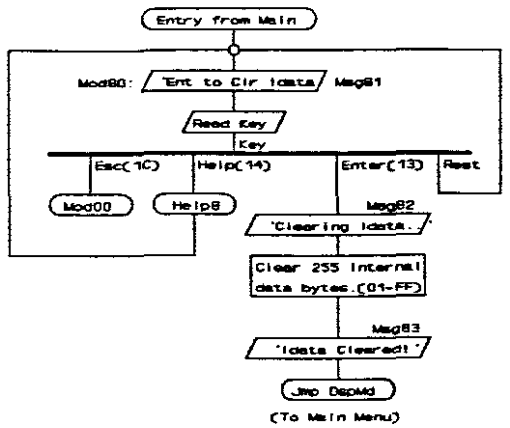



**Fig.10.23. Mode 8: Clear Internal Data Operating Diagram**

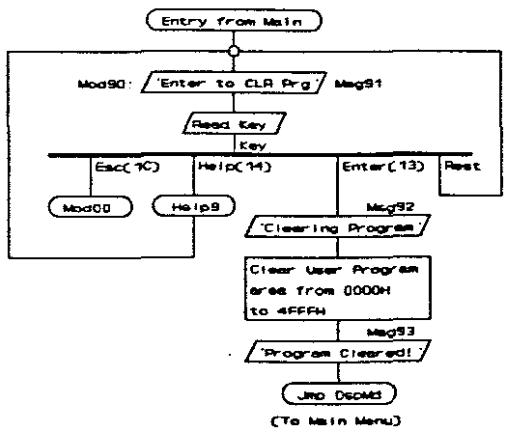



**Fig.10.24. Mode 9: Clear Program Area Operating Diagram**

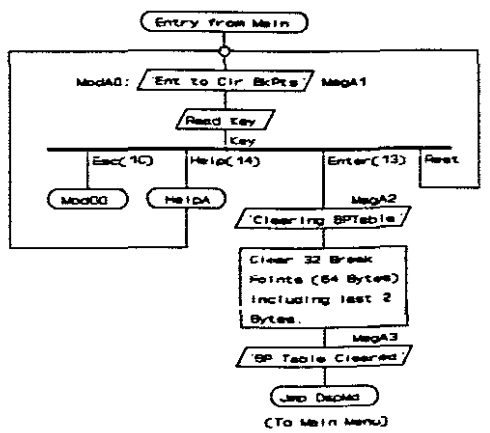



**Fig.10.25. Mode A: Clear Break Point Table Operating Diagram**

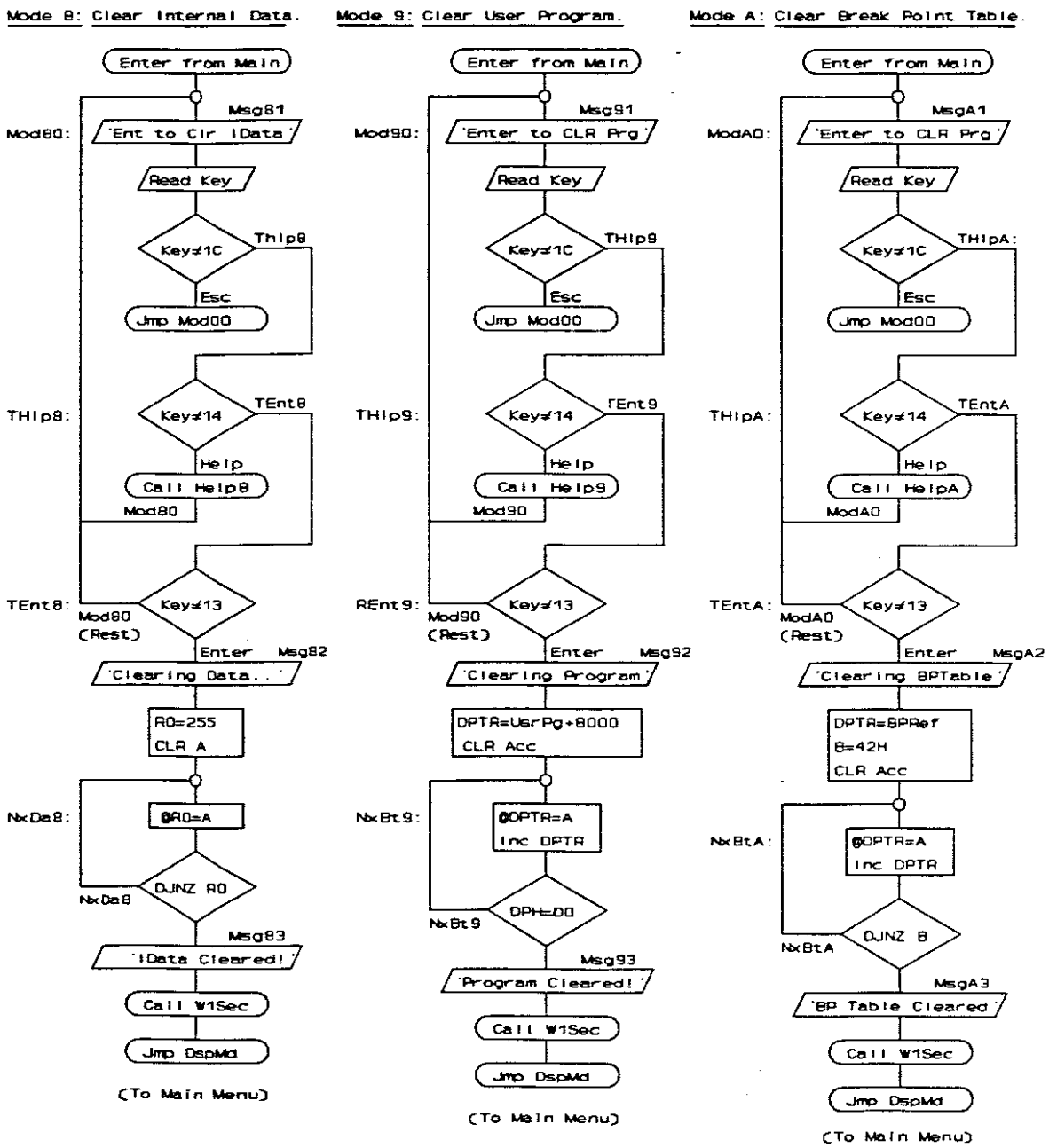



**Fig.10.26. Mode 8, 9 & A: Clear Memory Areas Flow Diagrams**

```
;*************************************************************
;       Clear Internal Data - Mode 80
;*************************************************************

Mod80:  MOV DPTR,#Msg81             ;'Ent to Clr IData'
        CALL DspMg
        CALL ReadK
        CJNE A,#1CH,THlp8           ;Test if Esc Key
        JMP Mod00

THlp8:  CJNE A,#14H,TEnt8           ;Test if Help Key
        MOV DPTR,#Help8
        CALL Help
        JMP Mod80

TEnt8:  CJNE A,#13H,Mod80           ;Test if Enter Key
        MOV DPTR,#Msg82             ;'Clearing IData..'
        CALL DspMg
        MOV R0,#0FFH                ;Counter and Data Add
        CLR A

NxDa8:  MOV @R0,A
        DJNZ R0,NxDa8               ;Test if done (FF - 01)

        MOV DPTR,#Msg83             ;'IData Cleared!  '
        CALL DspMg
        CALL W1Sec                  ;Wait 1 sec
        MOV Mode,#80H               ;Restore Mode
        JMP DspMd

;*************************************************************
;       Clear User Program Area - Mode 90
;*************************************************************

Mod90:  MOV DPTR,#Msg91             ;'Ent to Clr UsrPg'
        CALL DspMg
        CALL ReadK
        CJNE A,#1CH,THlp9           ;Test if Esc Key
        JMP Mod00

THlp9:  CJNE A,#14H,TEnt9           ;Test if Help Key
        MOV DPTR,#Help9
        CALL Help
        JMP Mod90

TEnt9:  CJNE A,#13H,Mod90           ;Test if Enter Key
        MOV DPTR,#Msg92             ;'Clearing Program'
        CALL DspMg
        MOV DPTR,#UsrPg             ;Start of user program
        ORL DPH,#80H                ;Code Write Add is Code+8000H

NxBt9:  CLR A
        MOVX @DPTR,A                ;Write zero to Xdata
        INC DPTR
        MOV A,DPH
        CJNE A,#0D0H,NxBt9          ;Upper end of User Program + 1

        MOV DPTR,#Msg93             ;'Program Cleared!'
        CALL DspMg
        CALL W1Sec                  ;Wait 1 sec
        MOV Mode,#90H               ;Restore Mode
        JMP DspMd
```

```
;************************************************************
;       Clear Break Point Table - Mode A0
;************************************************************

ModA0:   MOV DPTR,#MsgA1            ;'Ent to Clr BkPts'
         CALL DspMg
         CALL ReadK
         CJNE A,#1CH,THlpA          ;Test if Esc Key
         JMP Mod00

THlpA:   CJNE A,#14H,TEntA          ;Test if Help Key
         MOV DPTR,#HelpA
         CALL Help
         JMP ModA0

TEntA:   CJNE A,#13H,ModA0          ;Test if Enter Key
         MOV DPTR,#MsgA2            ;'Clearing BPTable'
         CALL DspMg
         MOV DPTR,#BPRef            ;Start of Break Points
         MOV B,#42H                 ;Clear 64 + 2 Bytes
         CLR A

NxBtA:   MOVX @DPTR,A               ;Write zero to BP Table
         INC DPTR
         DJNZ B,NxBtA

         MOV DPTR,#MsgA3            ;'BP Table Cleared'
         CALL DspMg
         CALL W1Sec                 ;Wait 1 sec
         MOV Mode,#0A0H             ;Restore Mode
         JMP DspMd

;************************************************************
```

```
;*************************************************************
;       Clear Break Point Table - Mode A0
;*************************************************************

ModA0:  MOV DPTR,#MsgA1             ;'Ent to Clr BkPts'
        CALL DspMg
        CALL ReadK
        CJNE A,#1CH,THlpA           ;Test if Esc Key
        JMP Mod00

THlpA:  CJNE A,#14H,TEntA           ;Test if Help Key
        MOV DPTR,#HelpA
        CALL Help
        JMP ModA0

TEntA:  CJNE A,#13H,ModA0           ;Test if Enter Key
        MOV DPTR,#MsgA2             ;'Clearing BPTable'
        CALL DspMg
        MOV DPTR,#BPRef             ;Start of Break Points
        MOV B,#42H                  ;Clear 64 + 2 Bytes
        CLR A

NxBtA:  MOVX @DPTR,A                ;Write zero to BP Table
        INC DPTR
        DJNZ B,NxBtA

        MOV DPTR,#MsgA3             ;'BP Table Cleared'
        CALL DspMg
        CALL W1Sec                  ;Wait 1 sec
        MOV Mode,#0A0H              ;Restore Mode
        JMP DspMd

;*************************************************************
```

## 10.14. Mode B: RS232 Serial Load from a PC

The 11.059 MHz crystal used on the system allows one to select any of the standard asynchronous serial baud rates between 300 and 9600 baud accurately. The data format protocol is 8 data bits, No Parity and 1 Stop Bit.

The serial data expected must be in the INTEL Hex format. In each line the number of bytes and the address of the first byte in the line are specified, followed by the data bytes and a checkbyte. A complete description of the INTEL Hex format is included in the appendix, but a brief summary of the contents of one line of data is as follows:

```
:10123400112233445566778899AABBCCDDEEFF00CC
```

* A line always starts with a colon (:)

* Two hexadecimal digits specify the number of data bits included in that line with a maximum of sixteen.

* Four hexadecimal digits specify the address of the first data bit, with ascending addresses for the following bytes.

* Two Control digits follow (00 = Normal Data, and 01 = the End Line).

* From one to sixteen data bytes, made up of two hexadecimal digits each.

* A two hexadecimal digit checkbyte.

After selecting the appropriate baud rate in this mode, the system will monitor the RS232 link until the colon of an INTEL Hex format line is received. Each line will be interpreted and stored until the "End of Transmission" line is received. If a checkbyte error was detected, or any key was pressed by the user, the reception will be aborted. The loading process from the PC is initiated by the normal DOS **COPY** command to, for example, COM1.

The only place in the Operating System where the 3-minute auto-off feature is not working, is when the system waits for a next INTEL Hex line. This feature can be used to keep the system active and not switching off within the prescribed three minutes. The system does not have to be connected to a PC and the pressing of any key will return the control to the main menu.

The main aim of the serial load feature is to load user programs into the system. It is also possible to dump and reload any of the other memory segments to or from a PC. The contents of all the memory segments are accessible for reading and writing on the 64K byte External Data Map of the micro-controller. The table below shows the external address locations of the various segments:

| **Segment** | **Normal Address** | **Memory** | **External Address** |
|---|---|---|---|
| Xdata (32K) | 0000h - 7FFFh | Data RAM | 0000h - 8FFFh |
| User Code (22K) | 0000h - 4FFFh | Code RAM | 8000h - CFFFh |
| Op.System (8K) | 5000h - 6FFFh | Code RAM | D000h - EFFFh |

| Segment | Normal Address | Memory | External Address |
|---|---|---|---|
| SFR (128b) | 08h - FFh | Internal | F0C2h - F0D1h |
| | | Code RAM | F100h - F10Ah |
| Idata (24b) | 00h - 17h | Internal | F0D2h - F0E9h |
| Idata(rest)(232) | 18h - 7Fh | Internal | F118h - F1FFh |
| Configuration | F800h - FFFFh | Code RAM | F800h - FFFFh |

The code addresses of an assembled user program on the PC are normally from address 0000h upwards, and stored as such in the INTEL Hex format. To enable one to load such a file and write it directly to the correct code addresses, the most significant *address* bit of any data byte read through the RS232 channel, will automatically be set by the software. It will therefore be written to the *top* half of the 64K byte external data memory map from address 8000h upwards.

Bulk storage of the current settings of a program is all located in the top half of the 64K bytes external memory. It can be stored to disk by using Mode C, and reloaded back without any problems created by the setting of the most significant bit of the address. These bits are set already. The only problem that can exist is when a user wants to store some of his normal external data in the lower 32K bytes to disk, for later retrieval.

Here, the most significant address bit will be zero, but when reloaded, will land in the top half of the 64K byte address map. With some care, this can also be bypassed by using the Xdata Block Move Option. It is very easy to load the user program that one can 'borrow' the user code address space temporary, and reload

the program again afterwards. To dump external user data from the lower half of the memory map, it may be dumped to the PC directly. After reloading, it needs to be block-moved downwards. Data between external addresses 5000 and 7FFFh must first be block-moved downwards before dumping, else it will be lost when trying to overwrite the protected Operating System and LCA configuration areas.

The Operating Diagram, the Flow Diagrams, and the Assembler listings of the software routines for Mode B and related subroutines are included on the following pages:

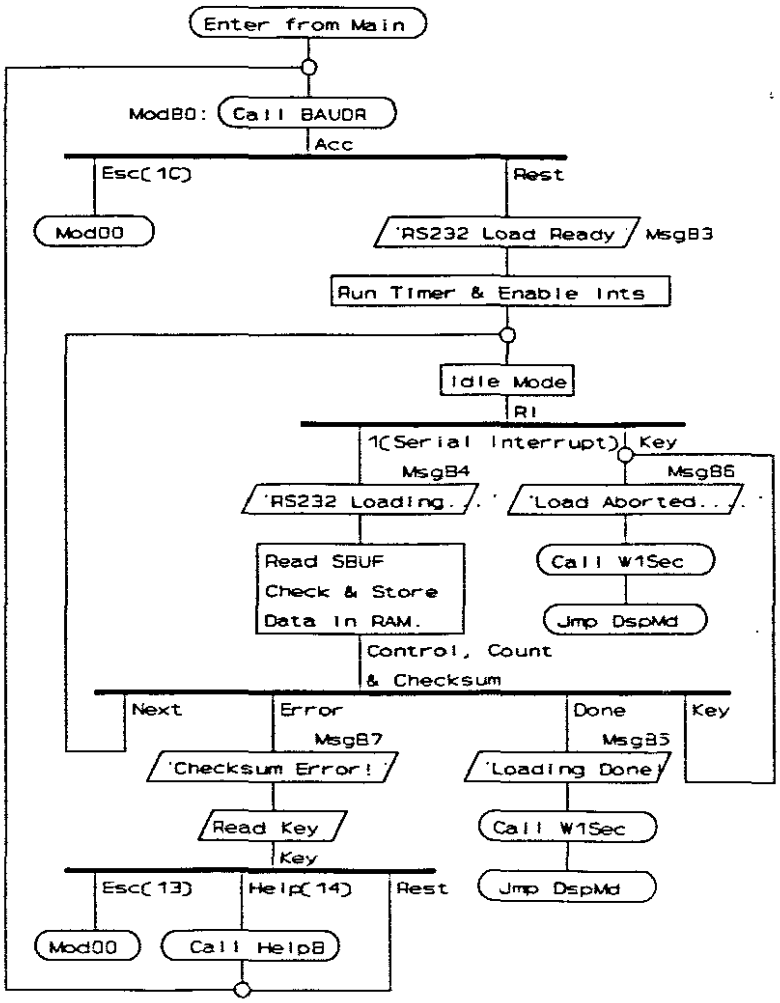



**Fig.10.27. RS232 Load Operating Diagram**

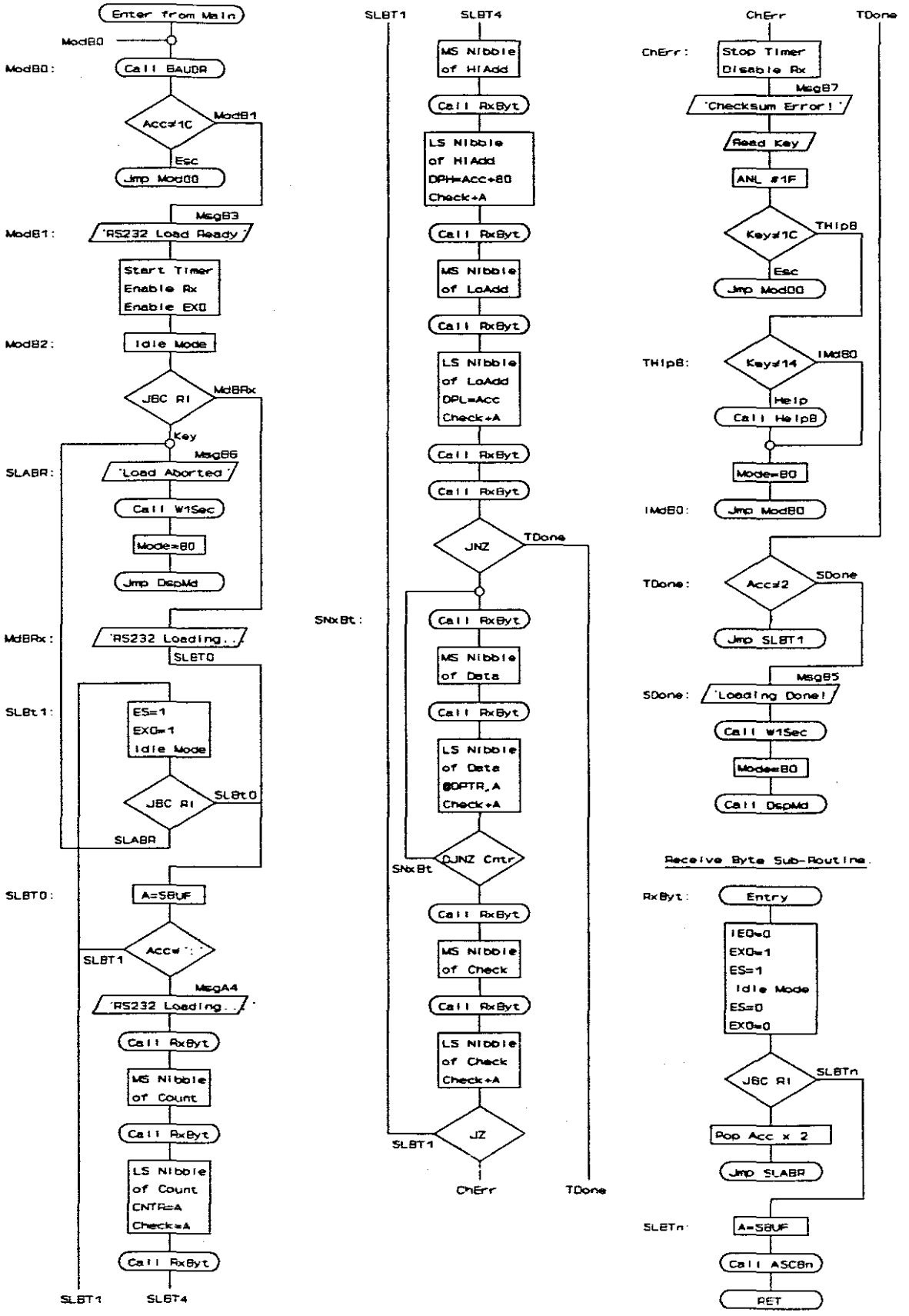



**Fig.10.28. RS232 Load Flow Diagram**

## Assembler Listing

```
;*************************************************************
;       Load Data Through RS232 - Mode B0
;*************************************************************

ModB0:  CALL BaudR                  ;To Set Baud Rate
        CJNE A,#1CH,ModB1           ;Test Esc out of BaudR sub
        JMP Mod00

ModB1:  MOV DPTR,#MsgB3             ;'RS232 Load Ready'
        CALL DspMg

        MOV SCON,#01010000B         ;Mode 1 Recieve
        SETB EX0                    ;For Keyboard Interrupt
        SETB ES                     ;Enable Rx Interrupt
        SETB TR1                    ;Start Buad Rate Generator

ModB2:  ORL PCON,#01H               ;Wait for Rx or Key
        CLR EX0                     ;Disable interrupts
        CLR ES
        JBC RI,MdBRx                ;Jmp if Serial Rx Interrupt
                                    ;else for Esc key interrupt
SLABR:  CLR TR1                     ;Stop Baud Rate
        MOV SCON,#00H               ;Stop Serial Rx
        MOV DPTR,#MsgB6             ;'Load Aborted!   '
        CALL DspMg
        CALL W1Sec                  ;Wait 1 Second

        MOV Mode,#0B0H              ;Reset Mode
        JMP DspMd                   ;Back to Main Menu

MdBRx:  MOV DPTR,#MsgB4             ;'RS232 Loading...'
        CALL DspMg                  ;Only 1st time
        JMP SLBT0

SLBT1:  ORL IE,#00010001B           ;Enable Key and Serial Ints
        ORL PCON,#01H               ;Wait for Rx or Key
        JBC RI,SLBT0                ;Jmp if Serial Rx Interrupt
        JMP SLABR

SLBT0:  MOV A,SBUF
        CJNE A,#':',SLBT1           ;Wait for ':'

;       Receive and Store Data Byte counter for line

        CALL RxByt                  ;MSN of Byte Counter in line
        SWAP A
        ORL B,A                     ;Write Hi Nibble of Count

        CALL RxByt                  ;LSN of Byte Counter
        ORL A,B                     ;Add MSN of Count
        MOV B,A                     ;Store
        MOV R3,A                    ;Start of Checksum

;       Receive and Store Address

        CALL RxByt                  ;HiAdd Hi
        SWAP A
        MOV DPH,A                   ;Write hi nibble

        CALL RxByt
        ORL A,DPH                   ;Add in Acc
        MOV DPH,A                   ;Write Lo nibble
        ADD A,R3                    ;Add Checksum
```

```
            MOV R3,A                    ;Store Checksum
            ORL DPH,#80H                ;Code Write Add = Code+8000H
            CALL RxByt                  ;LoAdd Hi
            SWAP A
            MOV DPL,A                   ;Write hi nibble

            CALL RxByt
            ORL A,DPL                   ;Add in Acc
            MOV DPL,A                   ;Write Lo nibble
            ADD A,R3                    ;Add Checksum
            MOV R3,A                    ;Store Checksum


;           Receive and Store Control Byte

            CALL RxByt                  ;Get following '0'

            CALL RxByt                  ;Get Directive
            JNZ TDone                   ;If not '0' test type


;           Receive and Store Data Bytes

SNxBt:      CALL RxByt                  ;Next data byte Hi
            SWAP A
            MOV R0,A                    ;Store Hi Data Byte

            CALL RxByt                  ;Lo Data Byte
            ORL A,R0                    ;Add Hi Nibble

            MOVX @DPTR,A                ;Write Byte to Add + 8000H
            INC DPTR                    ;Ready for next Byte
            ADD A,R3                    ;Add Checksum
            MOV R3,A                    ;Store Checksum

            DJNZ B,SNxBt                ;Test if line done

            CALL RxByt                  ;MSN of Rx Checksum
            SWAP A
            MOV R0,A                    ;Store

            CALL RxByt                  ;LSN of Rx Checksum
            ORL A,R0                    ;Add MSN

            ADD A,R3                    ;Add to make zero
            JZ SLBT1                    ;Start new line if OK

ChErr:      CLR TR1                     ;Stop Baud Rate
            MOV SCON,#00H               ;Stop Serial Rx
            MOV DPTR,#MsgB7             ;'Checksum Error'
            CALL DspMg

            CALL ReadK
            CJNE A,#1CH,THlpB           ;Test if Esc Key
            JMP Mod00

THlpB:      CJNE A,#14H,IMdB0           ;Test if Help Key
            MOV DPTR,#HelpB
            CALL Help

IMdB0:      MOV Mode,#0B0H              ;Reset Mode
            JMP DspMd                   ;Back to Main Menu

TDone:      CJNE A,#02H,SDone           ;Test if a '2'= Init line
            JMP SLBT1                   ;Go to wait for next ':'
```

```
SDone:   CLR TR1                    ;Stop Baud Rate
         MOV SCON,#00H              ;Stop Serial Rx
         MOV DPTR,#MsgB5            ;'Loading Done!'
         CALL DspMg
         CALL W1Sec                 ;Wait 1 second
         MOV Mode,#0B0H             ;Reset Mode
         JMP DspMd                  ;Back to Main Menu

;************************************************************
;        Receive HEX Format Byte Subroutine
;************************************************************

;        Exit: Converted Rx Byte in Acc (00 to 0F) or any key.
;        Conditions: Serial Receive enabled and Baud running.
;        Corrupt:DPTR, Acc, C, EX0=0
;        Subs & Stack: 4 = 2 (SXInt 2), 4 = 2 (AscBn 2),
;                      8 = 2 (X0Int 6)

RxByt:   CLR IE0                    ;In case of previous setting
         SETB EX0                   ;Possible Key interrupt
         SETB ES                    ;Enable Serial Int
         ORL PCON,#01H              ;Wait in Idle mode
         CLR ES
         CLR EX0
         JBC RI,SLBTn               ;Test if INT was Serial
         POP Acc                    ;Empty stack of return add
         POP Acc
         JMP SLABR                  ;Jump to 'Load Aborted'

SLBTn:   MOV A,SBUF
         CALL ASCBn
         RET

;************************************************************
```

# Baud Rate Selection Subroutine

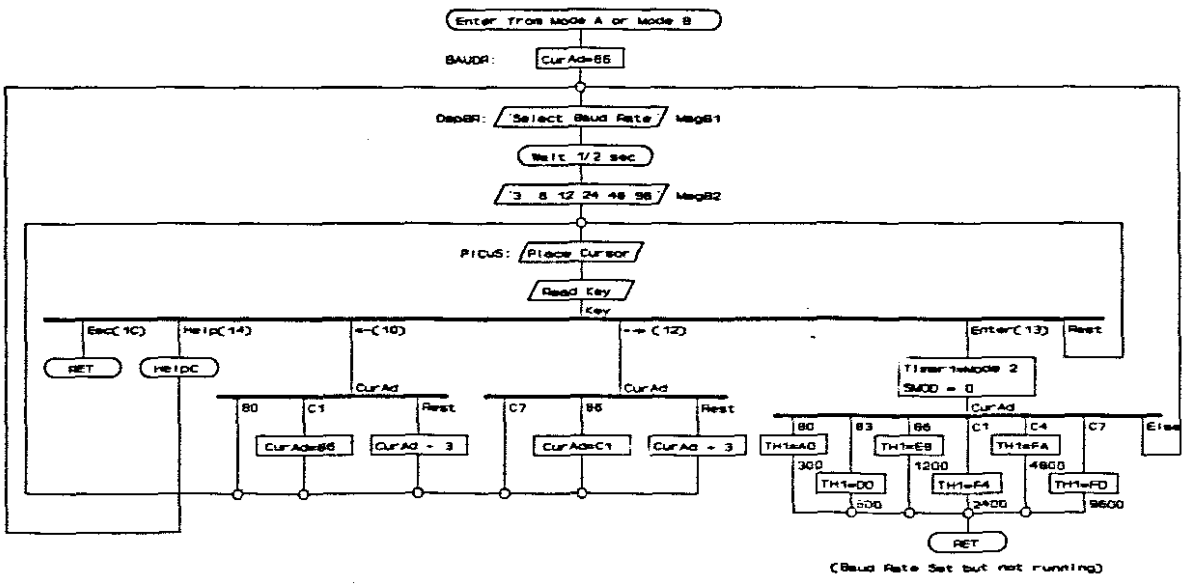



**Fig.10.29. Select Baud Rate Operating Diagram**

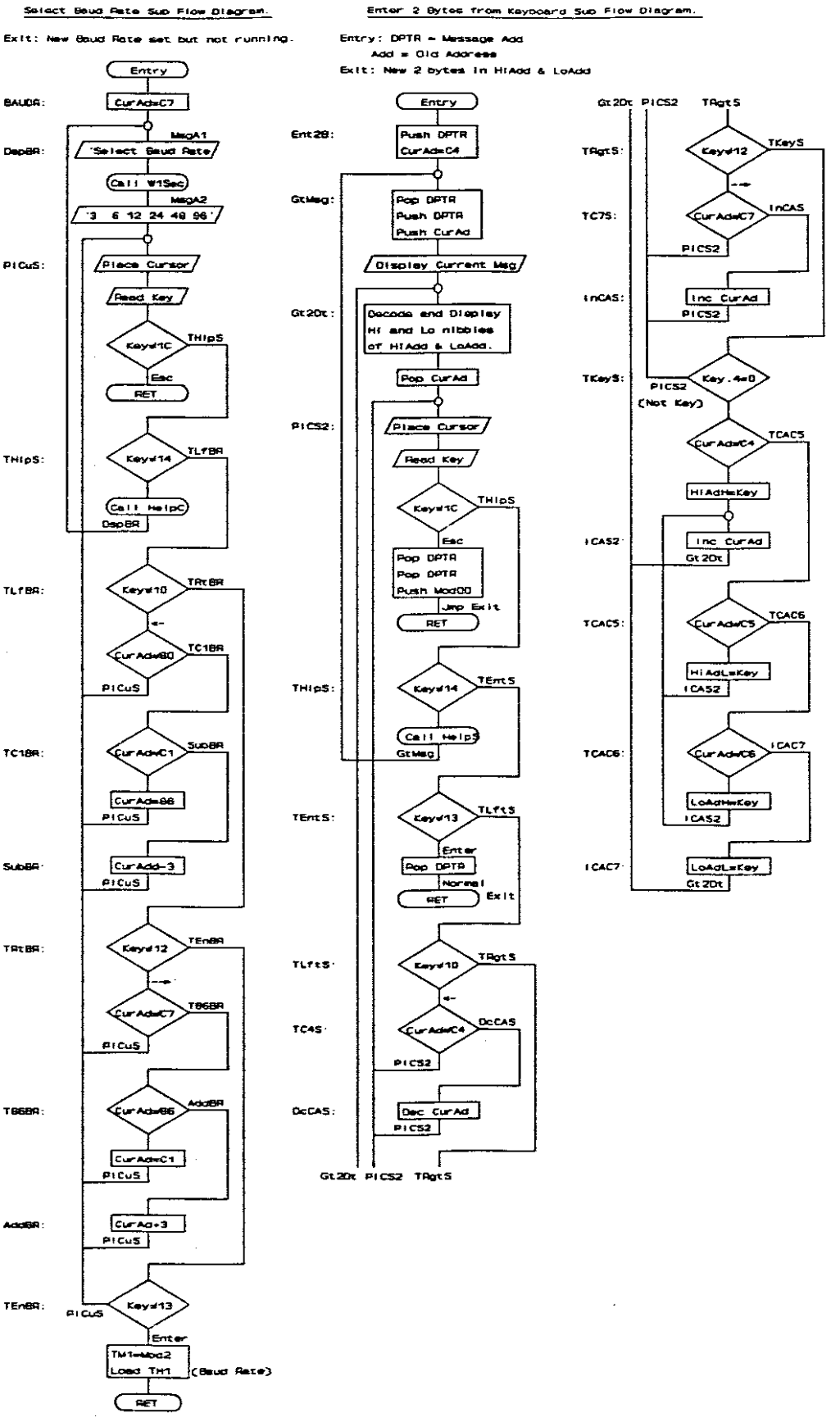



**Fig.10.30. Select Baud Rate and Enter 2 Keys Flow Diagrams**

## Assembler Listing

```
;*************************************************************
;       Select BAUD RATE Subroutine
;*************************************************************

;       Exit:Baud Rate Selected but not running
;       Corrupt: Tm0, Tm1, DPTR, Acc, B, C, R3-R7
;       Subs & Stack: 10 = 2 (DspMg 4 (WrDsp 2 (T0Int 2))),
;                     10 = 2 (ReadK 2 (X0Int 6),
;                   12 = 2 (Help 2 (DspMg 4 (WrDsp 2 (T0Int 2))))
;                    12 = 2 (Help 2 (ReadK 2 (X0Int 6)))

BaudR:  MOV CurAd,#086H             ;Place cursor at 86 (1200 bd)
DspBR:  MOV DPTR,#MsgB1             ;'Select Baud Rate'
        CALL DspMg
        MOV DPTR,#0F640H            ;Wait 1/2 second
        CALL WPSec
        MOV DPTR,#MsgB2             ;'3  6 12 24 48 96'
        CALL DspMg

P1CuS:  MOV DPTR,#DspCm
        MOV A,CurAd
        CALL WrDsp                  ;Place Cursor
        CALL READK
        CJNE A,#1CH,TH1pC           ;Test ESC key
        RET

TH1pC:  CJNE A,#14H,TLfBR           ;Test Help key
        MOV DPTR,#HelpC
        CALL Help
        JMP DspBR

TLfBR:  CJNE A,#10H,TRtBR           ;Test Left key
        CJNE CurAd,#80H,TC1BR       ;Stay at left side
        JMP P1CuS

TC1BR:  CJNE CurAd,#0C1H,SubBR      ;From C1 to 86
        MOV CurAd,#86H
        JMP P1CuS

SubBR:  DEC CurAd                   ;Else subtract 3
        DEC CurAd
        DEC CurAd
        JMP P1CuS

TRtBR:  CJNE A,#12H,TEnBR           ;Test Right
        CJNE CurAd,#0C7H,T86BR      ;Stay at right side
        JMP P1CuS

T86BR:  CJNE CurAd,#86H,AddBR       ;From 86 to C1
        MOV CurAd,#0C1H
        JMP P1CuS

AddBR:  INC CurAd                   ;Else Add 3
        INC CurAd
        INC CurAD
        JMP P1CuS

IBAUD:  JMP BaudR                   ;Intermediate Jump

TEnBR:  CJNE A,#13H,P1CuS           ;Test Enter key
        ANL TMOD,#0FH               ;Set Timer 1 = Mode 2
        ORL TMOD,#20H
        ANL PCON,#01111111B         ;SMOD = 0
```

```
        CJNE CurAd,#80H,TBR6        ;Test 300 baud
        MOV TH1,#0A0H
        RET

TBR6:   CJNE CurAd,#83H,TBR12       ;Test 600
        MOV TH1,#0D0H
        RET

TBR12:  CJNE CurAd,#86H,TBR24       ;Test 1200
        MOV TH1,#0E8H
        RET

TBR24:  CJNE CurAd,#0C1H,TBR48      ;Test 2400
        MOV TH1,#0F4H
        RET

TBR48:  CJNE CurAd,#0C4H,TBR96      ;Test 4800
        MOV TH1,#0FAH
        RET

TBR96:  CJNE CurAd,#0C7H,IBAUD      ;Test 9600
        MOV TH1,#0FDH
        RET

;*************************************************************
;       Enter 2 Bytes from Keyboard Subroutine.
;*************************************************************

;       Enter: Message Add in DPTR
;       Exit: New 2 bytes (4 Nibbles) in HiAdd & LoAdd
;       Corrupt: DPTR, Acc, B, C, Tm0, Tm1, R1-R7, EX0=1
;       Subs & Stack: 11 = 3 (DspMg 4 (WrDsp2 (T0Int 2))),
;                     10 = 3 (ReadK 2 (X0Int 5)),
;                   13 = 3 (Help 2 (DspMg 4 (WrDsp 2 (T0Int 2)))),
;                    13 = 3 (Help 2 (ReadK 2 (XoInt 6)))

Ent2B:  PUSH DPL
        MOV CurAd,#0C4H

;       Display Message and Hi & LoAdd

GtMsg:  POP DPL                     ;Get Message Ref in DPTR
        PUSH DPL                    ;Store for re-use

        MOV DPH,#68H
        CALL DspMg                  ;Display Current Message
Gt2Dt:  MOV DPTR,#DspCm             ;Place Cursor on C4
        MOV A,#0C4H
        CALL WrDsp
        MOV DPTR,#DspDa             ;Display Data Add
        MOV A,HiAdd                 ;Get HiAdd Hi Nibble
        SWAP A
        CALL BnASC
        CALL WrDsp                  ;Write Character to disp
        MOV A,HiAdd                 ;Get HiAdd Lo Nibble
        CALL BnASC
        CALL WrDsp
        MOV A,LoAdd                 ;Get LoAdd Hi Nibble
        SWAP A
        CALL BnASC
        CALL WrDsp                  ;Write Character to disp
        MOV A,LoAdd                 ;Get LoAdd Lo Nibble
        CALL BnASC
        CALL WrDsp
```

```
PlCS2:   MOV DPTR,#DspCm
         MOV A,CurAd
         CALL WrDsp                  ;Place Cursor
         CALL ReadK

         CJNE A,#1CH,THlpS           ;Test Esc Key
         Dec SP                      ;Empty Stack
         RET

THlpS:   CJNE A,#14H,TEntS           ;Test Help Key
         MOV DPTR,#HelpS
         CALL Help
         JMP GtMsg

TEntS:   CJNE A,#13H,TLftS           ;Test Enter Key
         Dec SP
         RET                         ;Normal Return, Adds as is.

TLftS:   CJNE A,#10H,TRgtS           ;Test Left Key
         CJNE CurAd,#0C4H,DcCAS      ;Test Cursor is on left digit
         MOV CurAd,#0C7H
         JMP PlCS2

DcCAS:   DEC CurAd
         JMP PlCS2

TRgtS:   CJNE A,#12H,TKeyS           ;Test Right Key
         CJNE CurAd,#0C7H,InCAS      ;Test Cursor is on right digit
         MOV CurAd,#0C4H
         JMP PlCS2

InCAS:   INC CurAd
         JMP PlCS2

TKeyS:   JB Acc.4,PlCS2              ;Jmp if not a Number Key

;        Replace Hi & LoAdd with key value

         CJNE CurAd,#0C4H,TCAC5      ;Test if Cursor Add = C4
         SWAP A                      ;Key to MSN
         ANL 02H,#0FH                ;Mask out MSN of HiAdd
         ORL 02H,A                   ;Write Key to Hi HiAdd

ICAS2:   INC CurAd                   ;Cursor on next digit
         JMP Gt2Dt

TCAC5:   CJNE CurAd,#0C5H,TCAC6      ;Test if Cursor Add = C5
         ANL 02H,#0F0H               ;Mask out LSN of HiAdd
         ORL 02H,A                   ;Write Key to Lo HiAdd
         JMP ICAS2
TCAC6:   CJNE CurAd,#0C6H,ICAC7      ;Test if Cursor Add = C6
         SWAP A                      ;Key to MSN
         ANL 01H,#0FH                ;Mask out MSN of LoAdd
         ORL 01H,A                   ;Write Key tc Hi LoAdd
         JMP ICAS2

ICAC7:   ANL 01H,#0F0H               ;Mask out LSN of LoAdd
         ORL 01H,A                   ;Write Key to Lo LoAdd
         MOV CurAd,#0C4H
         JMP Gt2Dt

;************************************************************
```

## 10.15. Mode C: Dump External Data to the PC through the RS232 Link

In this mode, a user specified number of bytes from a specified starting address will be converted to the INTEL Hex format and transmitted through the RS232 link at a selected baud rate. All the standard asynchronous serial data transmission baud rates from 300 to 9600 are available. The transmission format is set to the protocol of 8 data bits, No parity and 1 stop bit.

The default protocol setting for the RS232 communication of a PC is normally Even parity, 7 data bits, and a baud rate of 9600 baud. If a mouse driver was installed, it normally changes to No parity, 8 data bits, and a baud rate of 1200 baud. The protocol of the PC may be altered by using the external MODE command. (See page 55 for a description of the use of the MODE command.)

The selected data in the external memory will be placed in groups of sixteen or fewer data bytes per line in the standard INTEL Hex format. Each is completed with a "carriage return" character, **0Ch**. When the end of the data block has been reached, the standard terminating line will be transmitted, followed by an "End-of-File" character, **26d** or **1Ah**.

To store the data on a PC disk, the standard DOS **COPY** command must be used before the transmission is started from the board:

**C:>COPY COM1:** ***filename*.HEX**

The PC will wait for the serial transmission train, store it on disk and close the file after receiving the "End of File" character. A normal Control-C on the

keyboard of the PC will end the reception prematurely and close the file. While waiting for the data from the board, some PCs will write a "timeout" 0 to the disk every one to two seconds. This will not adversely affect the reloading of the data back the board, because the board will ignore any initial characters until a colon (:) is received.

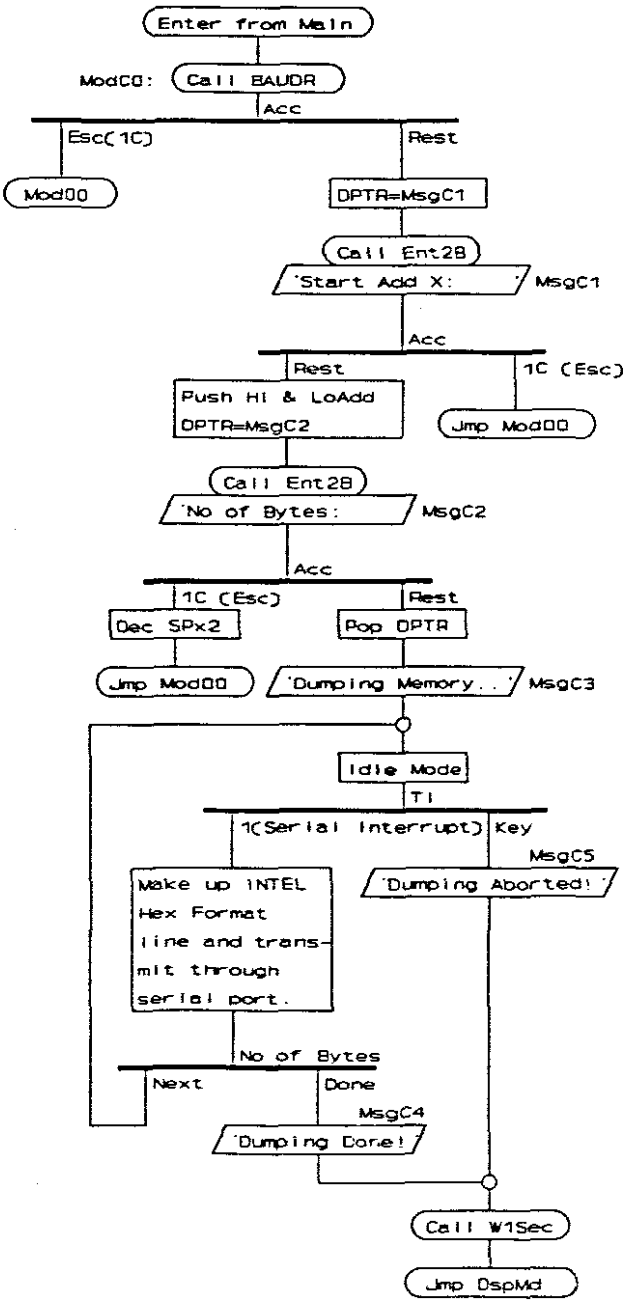



**Fig.10.31. Dump Data through the RS232 Link Operating Diagram**

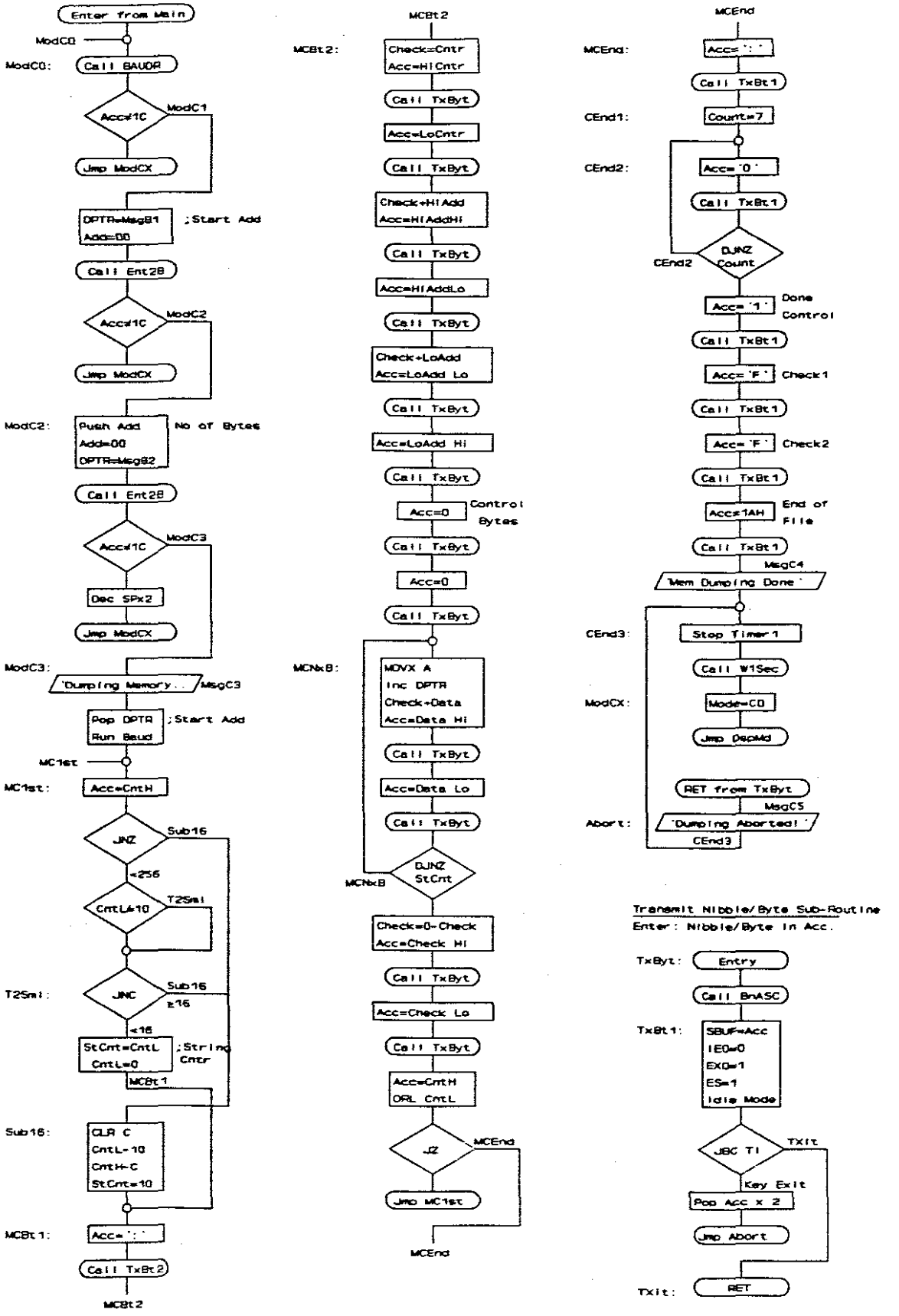



**Fig.10.32. Dump Data through the RS232 Link Flow Diagrams**

```
;*************************************************************
;       Dumping Memory to RS232 - Mode C0
;*************************************************************

ModC0:  CALL BaudR                  ;To Set Baud Rate
        MOV R0,TH1                  ;Save Baud Rate Setting
        CJNE A,#1CH,ModC1           ;Test Esc out of sub
        JMP ModCX

ModC1:  MOV LoAdd,#00H
        MOV HiAdd,#00H
        MOV DPTR,#MsgC1             ;'Start Add X:     '
        CALL Ent2B                  ;Enter 2 Bytes
        CJNE A,#1CH,ModC2           ;Exit after Esc
        JMP ModCX

ModC2:  Push 01H                    ;Store Start Add
        Push 02H
        MOV LoAdd,#00H
        MOV HiAdd,#00H
        MOV DPTR,#MsgC2             ;'No of Bytes:     '
        CALL Ent2B                  ;Enter 2 Bytes
                                    ;Byte Counter in Hi&LoAdd
        CJNE A,#1CH,ModC3           ;Exit after Esc
        Dec SP
        Dec SP
        JMP ModCX

ModC3:  MOV DPTR,#MsgC3             ;'Dumping Memory..'
        CALL DspMg
        Pop DPH                     ;Retrieve Start Add to DPTR
        Pop DPL

        MOV SCON,#01000000B         ;Mode 1 - Transmit
        MOV TH1,R0                  ;Recall Baud Rate Setting
        ANL TMOD,#0FH               ;Set Timer 1 = Mode 2
        ORL TMOD,#20H
        SETB TR1                    ;Start Baud Rate Generator

MC1st:  MOV A,HiAdd                 ;Hi Counter
        JNZ Sub16                   ;If HiCnt > 0, can subtract 16
        CJNE R1,#10H,T2Sml          ;Set C if LoCnt smaller than 16
T2Sml:  JNC Sub16                   ;Not smaller, goto -16
        MOV B,R1                    ;Line Byte Counter in B
        MOV R1,#00H                 ; and clear Lo Byte Counter
        JMP MCBt1

Sub16:  MOV A,R1
        CLR C
        SUBB A,#10H                 ;Subtract 16
        MOV R1,A                    ;Restore new Lo Count
        MOV A,R2                    ;Update Hi Count
        SUBB A,#00H                 ; if rollover carry set
        MOV R2,A                    ;Store Hi Count
        MOV B,#10H                  ; and Line counter is 16

;       Transmit ':' and Number of data bytes in line

MCBt1:  MOV A,#':'                  ;Set up for start
        CALL TxBt2                  ;Tx without BnASC conversion
```

```
MCBt2:  MOV R3,B                ;No of line bytes to checksum
        MOV A,B
        SWAP A                  ;Hi counter in Lo Acc
        CALL TxByt              ;Tx Hi Count
        MOV A,B                 ;Lo counter in A
        CALL TxByt              ;Tx Lo Count

;       Transmit Address

        MOV A,DPH               ;Hi Data Address
        ADD A,R3                ;Add HiAdd to Checksum
        MOV R3,A                ;Store checksum
        MOV A,DPH               ;Data Hi Add
        SWAP A                  ;HiAdd Hi
        CALL TxByt              ;Tx Hi Add
        MOV A,DPH
        CALL TxByt

        MOV A,DPL               ;Lo Data Address
        ADD A,R3                ;Add LoAdd to Checksum
        MOV R3,A                ;Store checksum
        MOV A,DPL               ;Data Lo Add
        SWAP A                  ;LoAdd Hi
        CALL TxByt              ;Tx Lo Add
        MOV A,DPH
        CALL TxByt

;       Transmit Control byte = 00H

        MOV A,#'0'              ;Token '0' for normal lines
        CALL TxBt2
        MOV A,#'0'              ;Valid Data = 2nd '0'
        CALL TxBt2

;       Transmit Data Bytes

MCNxB:  MOVX A,@DPTR            ;Next data byte in Acc
        ADD A,R3                ;Add Data to Checksum
        MOV R3,A                ;Store checksum
        MOVX A,@DPTR            ;Data
        SWAP A                  ;HiData in Lo Acc
        CALL TxByt              ;Tx Lo Add
        MOVX A,@DPTR
        CALL TxByt
        INC DPTR
        DJNZ B,MCNxB            ;Test if last byte of line

;       Transmit Checksum

        CLR A
        CLR C
        SUBB A,R3               ;00 - Checksum
        MOV R3,A                ;Store
        SWAP A
        CALL TxByt              ;Tx Hi Checksum
        MOV A,R3
        CALL TxByt
        MOV A,#0DH              ;Return character
        CALL TxBt2              ;Transmit without BnAsc convert

        MOV A,HiAdd
        ORL A,LoAdd
        JZ MCEnd                ;Test if there is more data
        JMP MC1st               ;Another line
```

```
;       Transmit 'End of Transmission' line

MCEnd:  MOV A,#':'              ;Start of Last line
        CALL TxBt2

CEnd1:  MOV B,#07H              ;Counter = 7 for 7 x '0'
CEnd2:  MOV A,#'0'              ; 2 for Byte Counter
        CALL TxBt2              ; + 4 for Address
        DJNZ B,CEnd2            ; + 1 for first control char

        MOV A,#'1'              ;'End of Transmission' code
        CALL TxBt2
        MOV A,#'F'              ;Checksum always FFH
        CALL TxBt2
        MOV A,#'F'
        CALL TxBt2
        MOV A,#1AH              ;End of File Character
        CALL TxBt2

        MOV DPTR,#MsgC4         ;'Mem Dumping Done'

CEnd3:  CLR TR1                 ;Stop Baud Rate
        CALL DspMg
        CALL W1Sec
ModCX:  MOV Mode,#0C0H          ;Restore Mode
        JMP DspMd               ;To Main Menu

Abort:  MOV DPTR,#MsgC5         ;'Dumping Aborted!'
        JMP CEnd3

;*************************************************************
;       Transmit Nibble or Byte Subroutine
;*************************************************************

;       Enter: Nibble (00 to 0F) or Byte in Acc
;       Conditions: Baud running
;       Corrupt: DPTR, Acc, C, EX0=0
;       Subs & Stack: 4 = 2 (BnASC 2), 4 = 2 (SXInt 2),
;                     8 = 2 (X0Int 6)

TxByt:  CALL BnASC              ;Convert Nibble to Ascii
TxBt2:  MOV SBUF,A              ;Start Tx
        CLR IE0                 ;Possible previous interrupt
        SETB EX0                ;Possible Key interrupt
        SETB ES
        ORL PCON,#01H           ;Wait to finish in Idle
        CLR ES
        CLR EX0
        JBC TI,TXit             ;Exit if SInt
        POP Acc                 ;Empty stack of return add
        POP Acc
        JMP Abort               ;Exit to Abort

TXit:   RET

;*************************************************************
```

## 10.16. Mode D: External Data Block Move

This mode is useful for moving blocks of data to other locations for temporary storage to make space for other data. For such a block move, the user must enter the lowest address for the source block, the lowest address of the destination block, and the number of bytes that must be moved. The data can be moved upwards or downwards, and the two blocks may even be overlapping, as long as both blocks fall within the boundary of the 64K byte external memory map of the controller.

For non-overlapping source and destination blocks, the move action will not create any problems. The first byte of the source is simply copied to the first byte of the destination block. The destination block will be a perfect copy of the source block. If the blocks are overlapping, however, the problem may exit that source bytes are overwritten as destination bytes, before they are moved. This is illustrated in the example below:

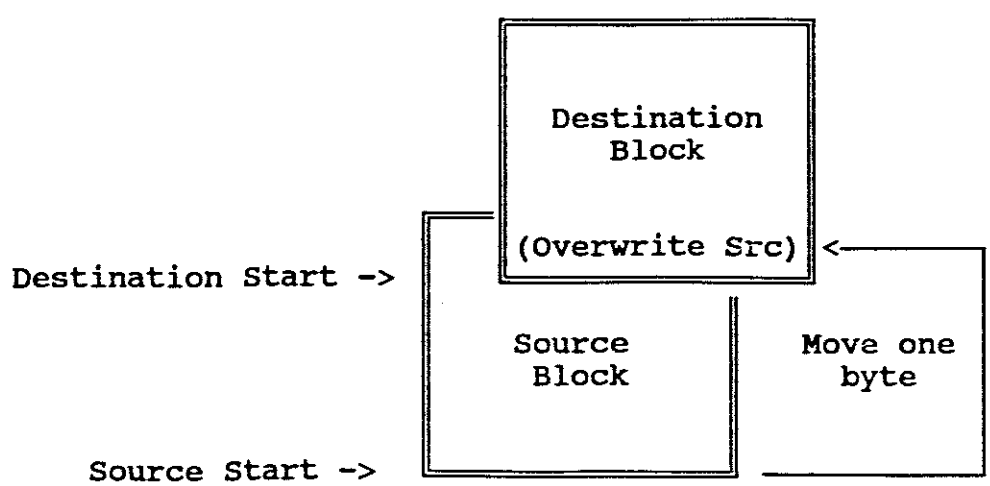



**Fig.10.33. An Overlapping Block Move**

When the lowest byte of the source block is written to the lowest byte of the destination block, it overwrites and destroys the data at the top of the source block. The solution is to start the movement of the data bytes from the highest byte of the source block to the highest byte of the destination block. When the *overwriting* occurs, the data has already been moved.

The Operating System will test the values of the source and destination addresses, as well as the number of bytes. If overlapping occurs, it will decide whether to start from the top or from the bottom. Address rollovers from FFFFh to 0000h are not allowed. The program will test if both blocks fall within the 64K bytes memory map. It not, an error message "Too many Bytes" will be displayed on the screen. The Operating Diagram, the Flow Diagrams, and the Assembler listings for the Block Move Mode D are included on the next few pages:

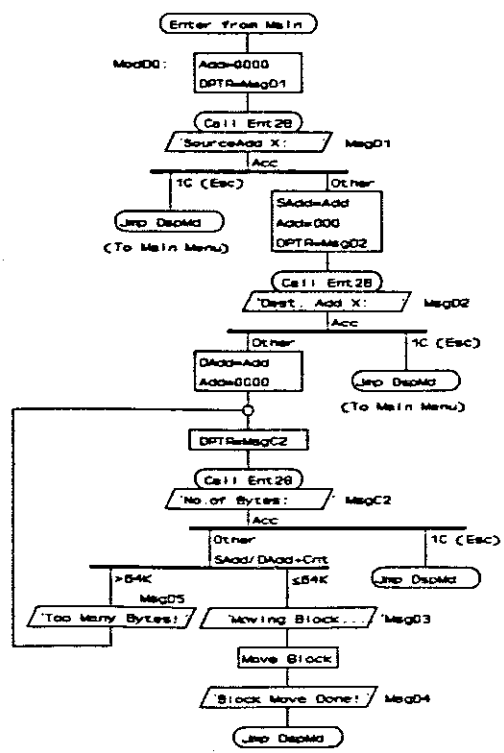



**Fig.10.34. External Data Block Move Operating Diagram**

When the lowest byte of the source block is written to the lowest byte of the destination block, it overwrites and destroys the data at the top of the source block. The solution is to start the movement of the data bytes from the highest byte of the source block to the highest byte of the destination block. When the *overwriting* occurs, the data has already been moved.

The Operating System will test the values of the source and destination addresses, as well as the number of bytes. If overlapping occurs, it will decide whether to start from the top or from the bottom. Address rollovers from FFFFh to 0000h are not allowed. The program will test if both blocks fall within the 64K bytes memory map. It not, an error message "Too many Bytes" will be displayed on the screen. The Operating Diagram, the Flow Diagrams, and the Assembler listings for the Block Move Mode D are included on the next few pages:

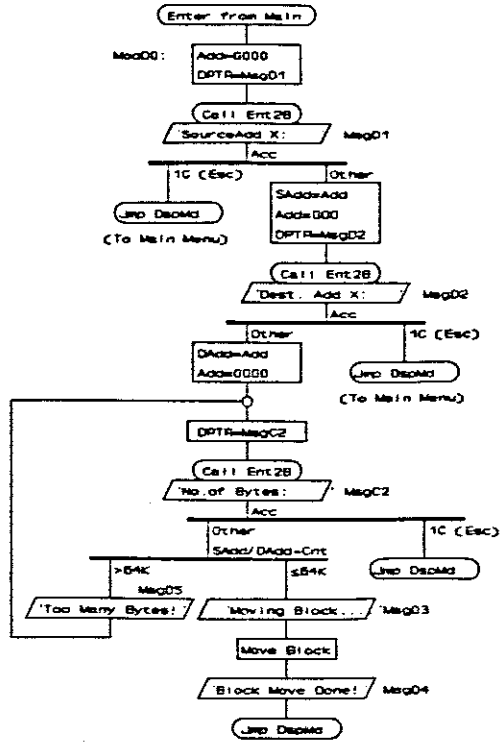



**Fig.10.34. External Data Block Move Operating Diagram**

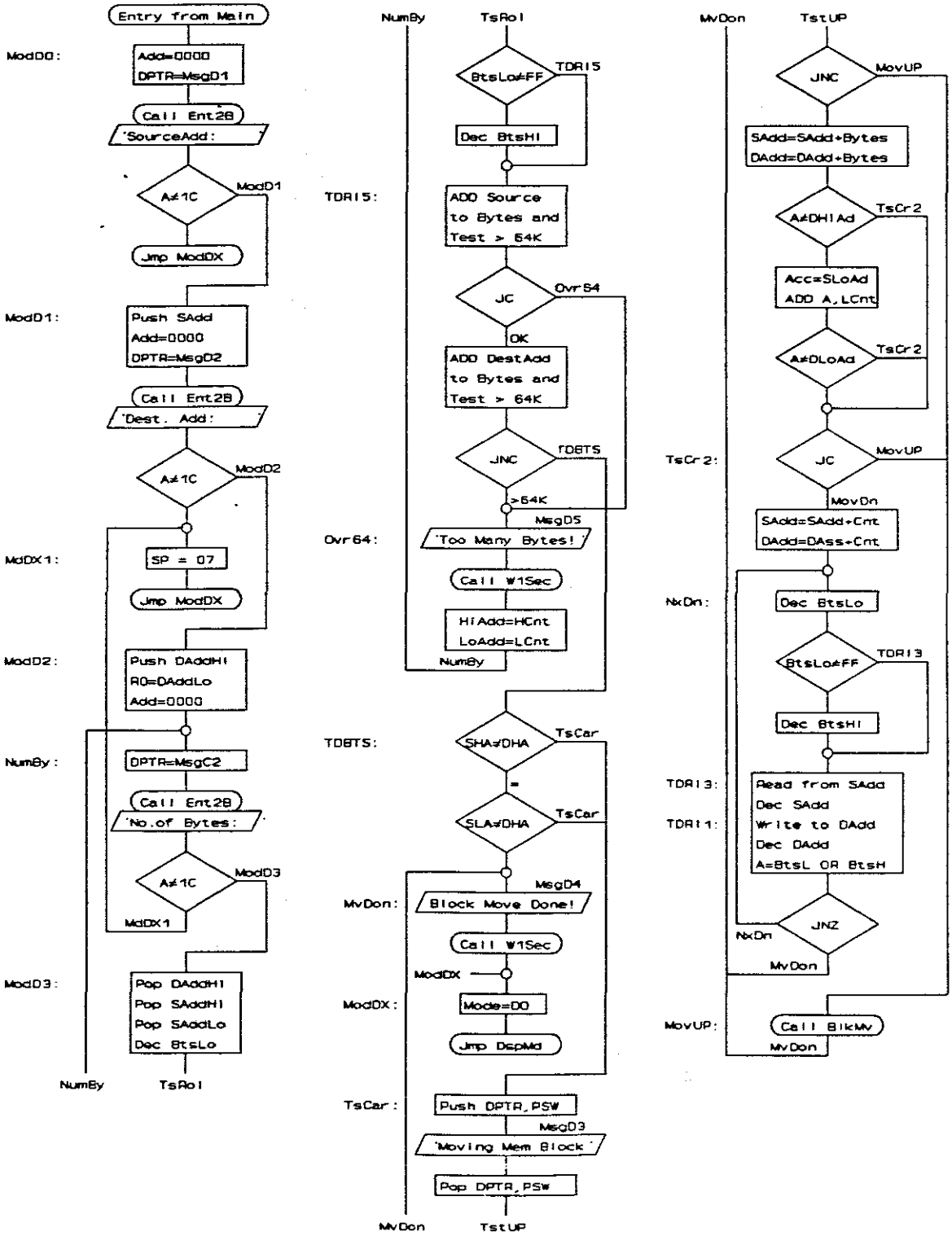



**Fig.10.35. External Data Block Move Flow Diagram**

```
;*************************************************************
;       XData Block Move - Mode D0
;*************************************************************

ModD0:  MOV LoAdd,#00H
        MOV HiAdd,#00H
        MOV DPTR,#MsgD1          ;'SourceAdd X:    '
        CALL Ent2B               ;Get New Source Add
        CJNE A,#1CH,ModD1
        Jmp ModDX

ModD1:  Push 01H                 ;Store Source in memory
        Push 02H
        MOV DPTR,#MsgD2          ;'Dest. Add X:    '
        MOV LoAdd,#00H           ;Clear Add
        MOV HiAdd,#00H

        CALL Ent2B               ;Get Destination Add
        CJNE A,#1CH,ModD2
MdDX1:  MOV SP,#07H
        Jmp ModDX

ModD2:  PUSH 02H                 ;Store Hi Destination in Stack
        MOV R0,01H               ;Lo Destination in R0

NumBy:  MOV LoAdd,#00H
        MOV HiAdd,#00H
        MOV DPTR,#MsgC2          ;'No.of Bytes:    '

        CALL Ent2B               ;Get Number of Bytes
        CJNE A,#1CH,ModD3
        Jmp MdDX1

ModD3:  Pop 03H                  ;Destination Hi
        Pop DPH                  ;Placement: R0 R1 R2 R3 DPH DPL
        Pop DPL                  ;            DL CL CH DH  SH  SL

        DEC 01H                  ;Minus 1 to correct Addition
        CJNE R1,#0FFH,TDR15      ;Test roll over
        DEC 02H

TDR15:  MOV A,DPL                ;Source Lo
        ADD A,R1                 ;Add Lo Count to test Carry
        MOV A,DPH                ;Source Hi
        ADDC A,R2                ;Add Hi Count, C=1 if > 64K
        JC Ovr64

        MOV A,R0                 ;Destina Lo
        ADD A,R1                 ;Add Lo Count to test Carry
        MOV A,R3                 ;Destina Hi
        ADDC A,R2                ;Add Hi Count, C=1 if > 64K
        JNC TDBTS

Ovr64:  PUSH DPL                 ;Store Source Add
        PUSH DPH
        PUSH 03H                 ;Store Hi Dest
        MOV DPTR,#MsgD5          ;'Too Many Bytes! '
        CALL DspMg
        CALL W1Sec

        JMP NumBy                ;Get new No. of Bytes
```

```
TDBTS:  MOV A,DPH               ;If SAH > DAH then C=0
        CJNE A,03H,TsCar        ; to move upwards.
        MOV A,DPL               ; Else downwards,
        CJNE A,00H,TsCar        ; or end if equal

MvDon:  MOV DPTR,#MsgD4         ;'Block Move Done!'
        CALL DspMg
        CALL W1Sec

ModDX:  MOV Mode,#0D0H          ;Reset Mode
        JMP DspMd               ;To Main Menu

TsCar:  PUSH PSW                ;Save Carry C=0 Up, C=1 Down
        PUSH DPL
        PUSH DPH
        MOV DPTR,#MsgD3         ;'Moving Mem Block'
CALL DspMg
        POP DPH                 ;     R0 R1 R2 R3 DPH DPL
        POP DPL                 ;     DL CL CH DH  SH  SL
        POP PSW

;       Test for Block Overlap for Downwards transfers

        JNC MovUP               ;Test for overlap DA>SA

        MOV A,DPH               ;Acc = Source Hi
        ADD A,R2                ;Add Hi Count
        CJNE A,03H,TsCr2        ;Set C if DH > SH+CH
        MOV A,DPL               ;Source Lo
        ADD A,R1                ;Add Lo Count
        CJNE A,00H,TsCr2        ;Set C if DL > SL+CL

TsCr2:  JC MovUP                ;Goto Move by Increment

        MOV A,DPL               ;Move by Decrement
        ADD A,R1
        MOV DPL,A               ;Source = Source + Count
        MOV A,DPH
        ADDC A,R2
        MOV DPH,A

        MOV A,R0                ;Destination
        ADD A,R1
        MOV R0,A                ;Destina = Destina + Count
        MOV A,R3
        ADDC A,R2
        MOV R3,A
        JMP TDR13               ;Skip first decriment

;       Move Block by Decrementing Addresses

NxDn:   DEC 01H                 ;Decriment Count
        CJNE R1,#0FFH,TDR13     ;Test if R1 Roll over
        DEC 02H

TDR13:  MOVX A,@DPTR            ;Read from Source Add
        PUSH Acc                ;Store Data
        DEC DPL
        MOV A,DPL
        CJNE A,#0FFH,TDR11      ;Test for roll over
        DEC DPH
TDR11:  POP Acc
        PUSH DPL                ;Store new Source Add
        PUSH DPH
        MOV DPL,R0              ;Destination
        MOV DPH,R3
```

```
        MOVX @DPTR,A            ;Write to Destination Add
        DEC R0                  ;Decriment Destination Add
        CJNE R0,#0FFH,TDR12
        DEC R3
TDR12:  POP DPH                 ;Restore Source Add in DPTR
        POP DPL
        MOV A,R2
        ORL A,R1
        JNZ NxDn                ;Test if Count = 0
        JMP MvDon

MovUp:  CALL BlkMv
        JMP MvDon               ;Go to Done

;*************************************************************
;       Move Block by Incrementing Addresses Subroutine
;*************************************************************

;       Enter: DPTR  = Source Address
;              R3,R0 = Destination Add
;              R2,R1 = Number of Bytes - 1
;       Conditions: Blocks Non-Overlapping & Sufficient
;                   Address space available
;       Corrupt: DPTR, R0 to R3, Acc, C, 4 Stack Bytes

NxUp:   DEC R1                  ;Decrement Counter
        CJNE R1,#0FFH,BlkMv
        DEC R2

BlkMv:  MOVX A,@DPTR            ;Read Source Add
        INC DPTR
        PUSH DPL                ;Store Source
        PUSH DPH
        MOV DPL,R0              ;Destination Add
        MOV DPH,R3
        MOVX @DPTR,A            ;Write to Destination
        INC DPTR
        MOV R0,DPL              ;Restore Desination
        MOV R3,DPH
        POP DPH                 ;Restore Source
        POP DPL
        MOV A,R2
        ORL A,R1
        JNZ NxUp                ;Test if count = 0
        RET

;*************************************************************
```

## 10.17. Modes E and F: Protect/Unprotect Memory and Save and Off Modes

The Protect/Unprotect option will test if it is possible to write successfully to the last byte in the Operating System memory area. It will write a '55' to the byte and confirm that a '55' is read. Then it will write 'AA' to the byte and test again. If either of the tests fails, the block is protected and the message "Enter to Unprot" will be displayed. If the write tests were successful, it will display "Enter to Protect". Pressing the 'Enter' key will toggle the Write Protect bit in the LCA Control Register, and repeat the test. An 'Esc' key will return the control to the main menu. The external data memory areas that will be protected against accidental overwriting, are:

**The Operating System:** D000h to EFFFh (5000h to 6FFFh Code)

**The LCA Configuration:** F800h to FFFFh (Not available as Code)

The "Save and Off" Mode will bulk store the current internal data of the micro-controller in the Code Memory IC before the power is switched off. The data bulk storage is normally used to store the place and current system variables of a specific point in a user program. Thus one can reload the data and continue from the same point and with exactly the same data and system variables.

Bulk storing at any other point, or the "Store and Off Mode" will overwrite the previous stored values. If one wants to conserve the previous values before switching the system off, the "Restore System Variables and Data" of Mode 4 can be called directly from within the Mode F menu, by simply pressing key '4'. After the bulk storage data has been restored to the micro-controller and the

system variables, a normal "OFF" can follow. It will place the original stored values back into the bulk storage.

To continue from the "stored point" in the user program after a reset or power up, Mode 4 can be used to reload the bulk storage back into the micro-controller. The user can then continue running the program in any of the *Running Modes*. The Operating Diagrams, the Flow Diagrams, and the Assembler Listings of Modes E and F are included on the next few pages:

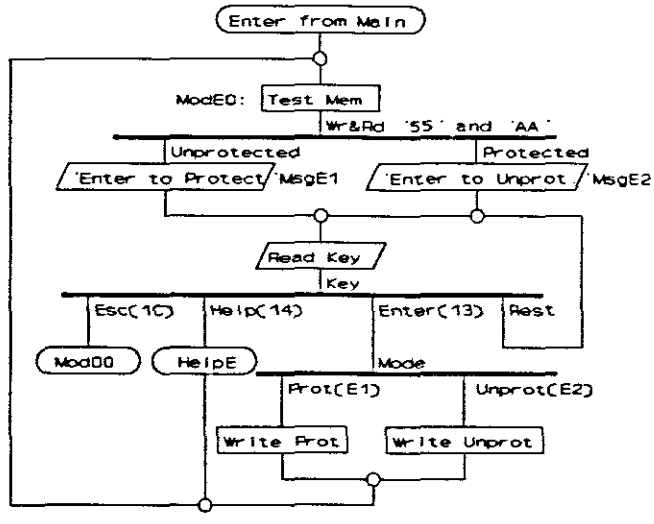



**Fig.10.36. Protect/Unprotect Memory Operating Diagram**

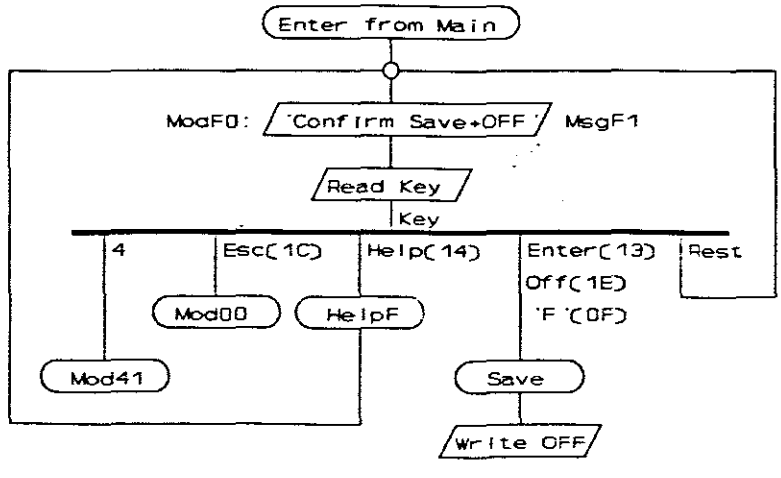



**Fig.10.37. Save and OFF Operating Diagram**

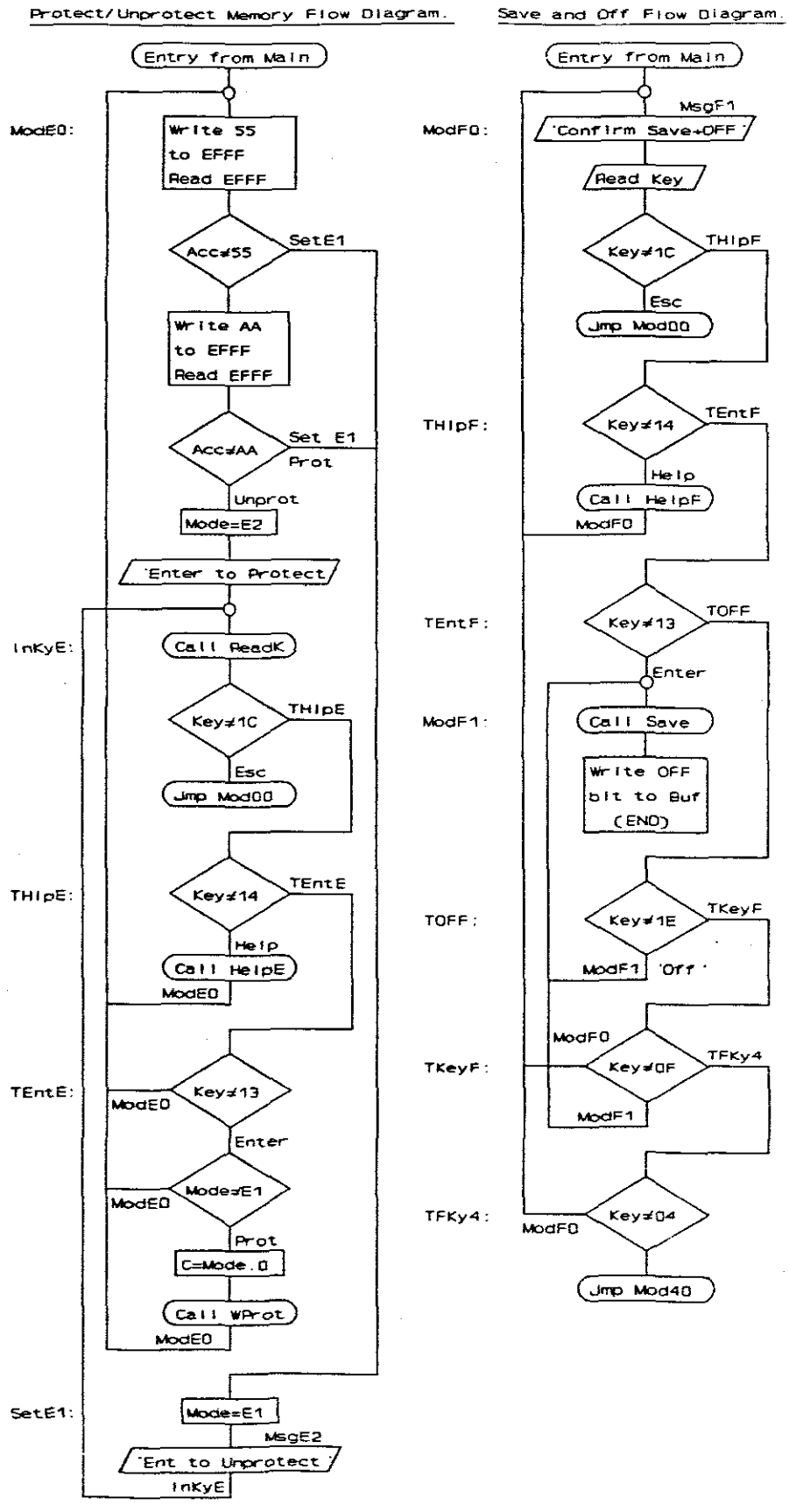



**Fig.10.38. Protect/Unprotect Memory and OFF Flow Diagrams**

```
;*************************************************************
;       Protect/Unprotect Memory - Mode E0
;*************************************************************

ModE0:  MOV DPTR,#0EFFFH            ;Select Test Byte
        MOV A,#55H
        MOVX @DPTR,A                ;Write '55H'
        MOVX A,@DPTR                ;Read same byte
        CJNE A,#55H,SetE1           ;Jmp if not the same
        MOV A,#0AAH
        MOVX @DPTR,A                ;Write 'AAH'
        MOVX A,@DPTR                ;Read same byte
        CJNE A,#0AAH,SetE1          ;Jmp if not the same
        MOV Mode,#0E2H              ;Is Unprotected (0010)
        MOV DPTR,#MsgE1             ;'Enter to Protect'
InKyE:  CALL DspMg

        CALL ReadK

        CJNE A,#1CH,THlpE           ;Test if Esc Key
        JMP Mod00

THlpE:  CJNE A,#14H,TEntE           ;Test if Help Key
        MOV DPTR,#HelpE
        CALL Help
        JMP ModE0

TEntE:  CJNE A,#13H,ModE0
        MOV A,Mode                  ;C=0 to Protect
        MOV C,Acc.0                 ;C=1 to Unprotect
        CALL WProt
        JMP ModE0

SetE1:  MOV Mode,#0E1H              ;Is Protected (0001)
        MOV DPTR,#MsgE2
        JMP InKyE

;*************************************************************
;       Write Protect/Unprotect OPSYS in Xdata Segment Sub
;*************************************************************

;       Enter: C = 0 for protect, C = 1 for Unprotect
;       Corrupt: DPTR, Acc, C, 2 Stack bytes

WProt:  MOV DPTR,#CntRg             ;Previous Control Setting
        MOVX A,@DPTR
        MOV Acc.1,C                 ;Write Protect Bit
        MOVX @DPTR,A                ;Store in Memory
        RET


;*************************************************************
;       Save and Off - Mode F0
;*************************************************************

ModF0:  MOV DPTR,#MsgF1             ;'Confirm Save+OFF'
        CALL DspMg
        CALL ReadK

        CJNE A,#1CH,THlpF           ;Test if Esc Key
        JMP Mod00
```

```
THlpF:   CJNE A,#14H,TEntF          ;Test if Help Key
         MOV DPTR,#HelpF
         CALL Help
         JMP ModF0

TEntF:   CJNE A,#13H,TOFF           ;Test if Enter Key

ModF1:   CALL Save
         MOV DPTR,#CntRg
         MOVX A,@DPTR
         SETB Acc.2                 ;Auto-Off bit D2=1
         MOVX @DPTR,A               ;Switch OFF
         ORL PCON,#02H              ;Stop operating

TOFF:    CJNE A,#1EH,TKeyF          ;Test if OFF key
         JMP ModF1

TKeyF:   CJNE A,#0FH,TFKy4          ;Test if 'F' Key
         JMP ModF1

TFKy4:   CJNE A,#04H,ModF0          ;Test if '4' key
         JMP Mod40                  ;Retrieve Data & SFR

;************************************************************
```

## 10.18. The Help Subroutine

Many of the Help File lines are repeated in various other help files as well. To conserve space, each Help File was made up by a *sequence line*. It starts with the number of lines in the file, followed by the reference numbers of each Help line. In this fashion, only one Help File subroutine was required, with a set of lookup tables. The Operating Diagram is shown below, and the Flow Diagram and Assembler listing are included on the next pages:

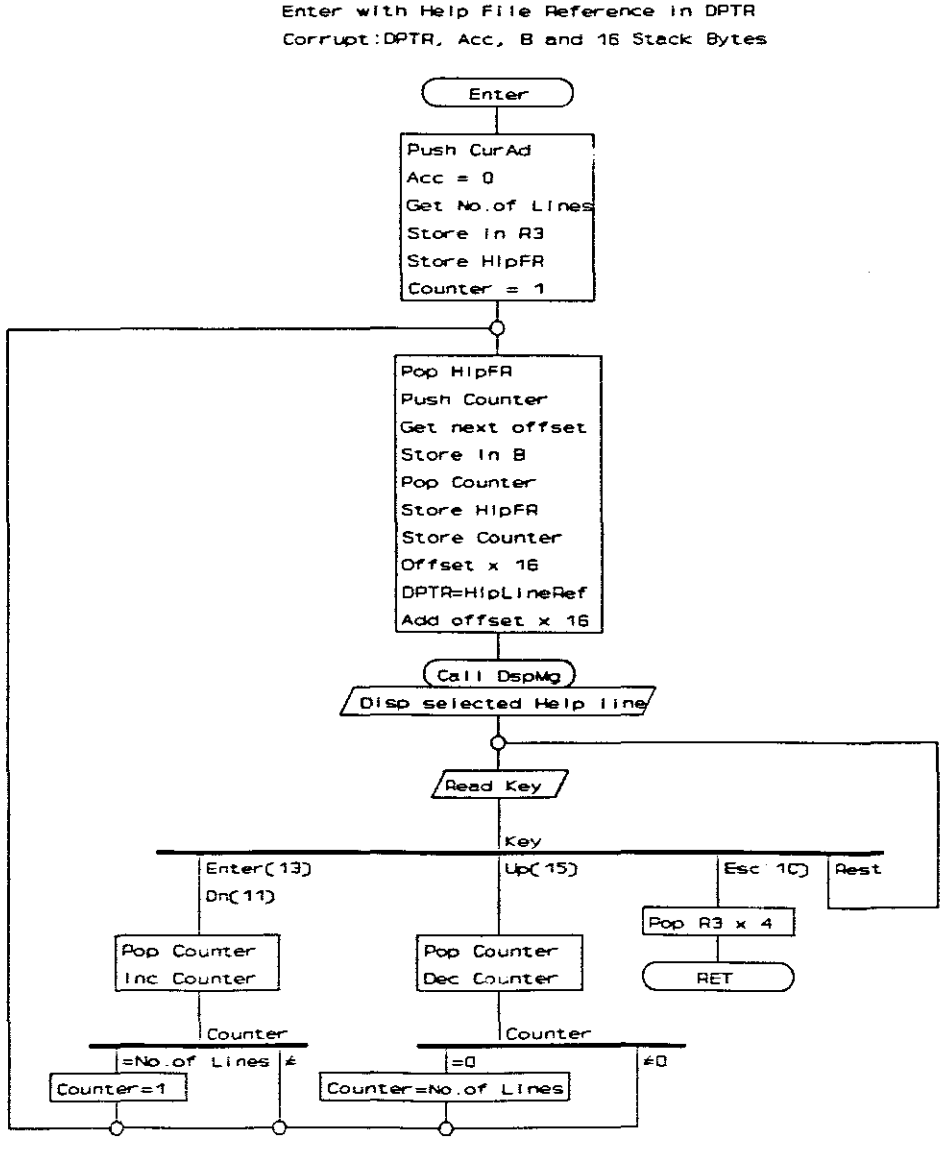



**Fig.10.39. Help Subroutine Operating Diagram**

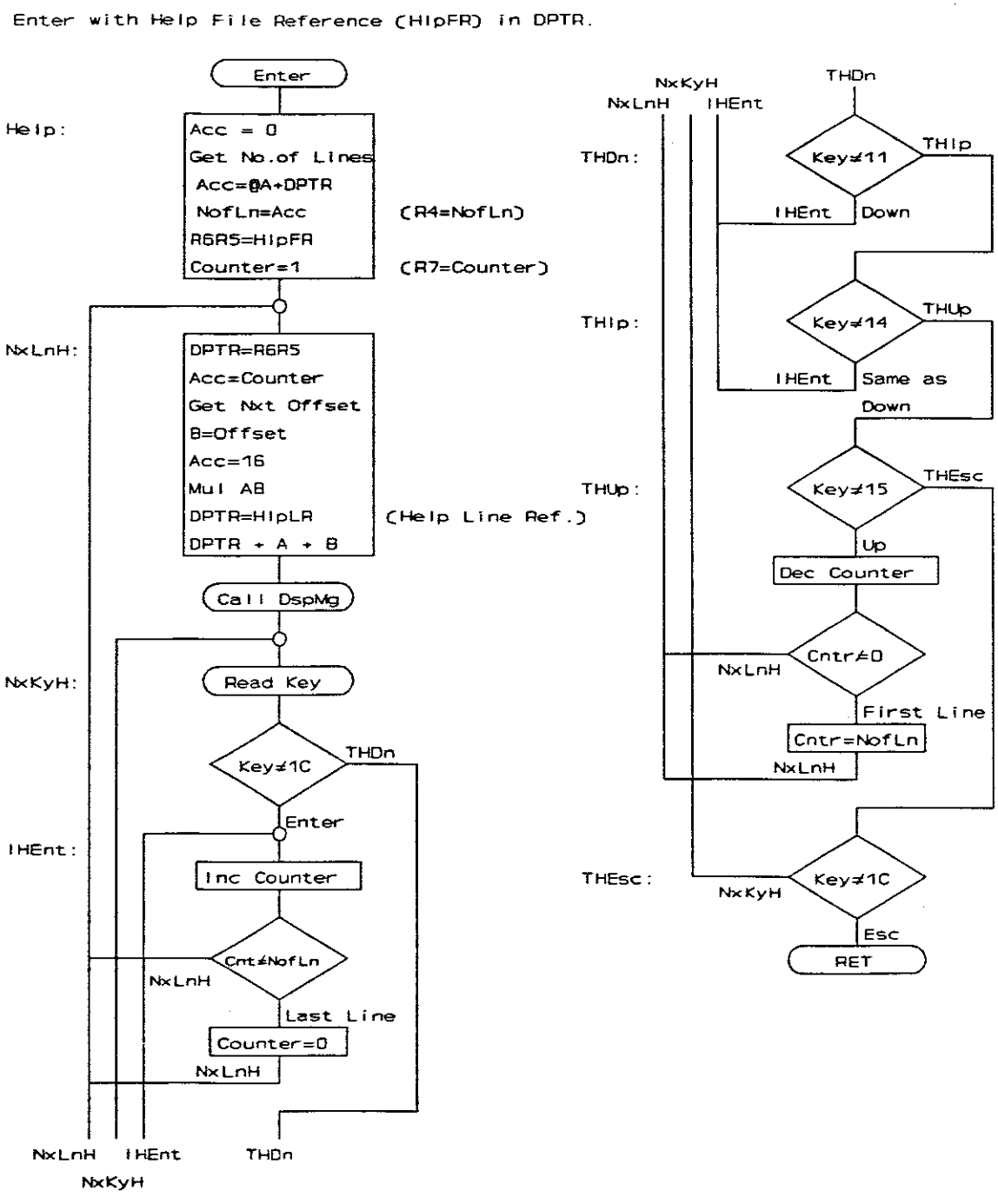



**Fig.10.40. Help Subroutine Flow Diagram**

## Assembler Listing

```
;*************************************************************
;       Help Subroutine
;*************************************************************

;       Enter:Help Reference in DPTR
;       Corrupt:DPTR, Acc, B, C, R4-R7, EX0=1
;       Subs & Stack: 10 = 2 (DspMg 4 (WrDsp 2 (T0INT 2))),
;                     10 = 2 (ReadK 2 (X0INT 6))

Help:   CLR A
        MOVC A,@A+DPTR              ;No of Lines to Acc
        MOV R4,A                    ;Store Number of lines
        INC DPTR                    ;Point to first line no.
        MOV R5,DPL                  ;Store Help Reference
        MOV R6,DPH
        MOV R7,#00H                 ;Set Counter = 0

NxLnH:  MOV DPL,R5
        MOV DPH,R6
        MOV A,R7                    ;Get Count
        MOVC A,@A+DPTR              ;Cet offset = HlpFR+Count
        DEC A                       ;Correction for line 0
        MOV B,A                     ;B=offset from Help base
        MOV A,#16
        MUL AB                      ;BA = No of Help offset bytes
        MOV DPTR,#HlpLR             ;Help base address
        ADD A,DPL                   ;Calculate new offset
        MOV DPL,A                   ; and add to base add
        MOV A,B
        ADDC A,DPH
        MOV DPH,A
        CALL DspMg

NxKyH:  CALL ReadK
        CJNE A,#13H,THDn            ;Test Enter
IHEnt:  INC R7                      ;Increment Count
        MOV A,R7
        CJNE A,04H,NxLnH            ;Nx line if more lines
        MOV R7,#00H                 ;Start again if last line
        JMP NxLnH

THDn:   CJNE A,#11H,THlp            ;Test Down
        JMP IHEnt

THlp:   CJNE A,#14H,THUp            ;Test Help Key
        JMP IHEnt

THUp:   CJNE A,#15H,THEsc           ;Test UP
        DEC R7
        CJNE R7,#0FFH,NxLnH         ;Prev line if not at top
        MOV 07H,R4                  ;Last line+1 if at top
        DEC R7
        JMP NxLnH

THEsc:  CJNE A,#1CH,NxKyH           ;Not Esc, read key again
        RET

;*************************************************************
```

```
;*************************************************************
;       Help file Sequences
;*************************************************************

Help0:   DB  18,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,98
Help1:   DB  19,1,18,19,20,21,22,23,24
         DB  5,25,26,27,28,29,30,31,17,32,98
Help2:   DB  16,1,33,34,35,36,37,38,39,5,40,41,26,25,17,32,98
Help3:   DB  6,1,42,43,44,45,98
Help4:   DB  6,1,46,47,48,49,98
Help5:   DB  11,1,50,51,52,53,5,26,25,17,32,98
Help6:   DB  7,1,54,55,5,56,32,98
HlpBP:   DB  12,1,57,58,59,60,5,25,26,61,56,32,98
Help7:   DB  6,1,62,63,56,32,98
Help8:   DB  6,1,64,65,66,67,98
Help9:   DB  6,1,68,69,70,67,98
HelpA:   DB  7,1,82,83,5,75,32,98
HelpB:   DB  8,1,71,72,73,74,75,32,98
HelpC:   DB  11,1,76,77,78,79,80,5,26,81,32,98
HelpE:   DB  7,1,84,85,86,87,88,98
HelpF:   DB  11,1,89,90,91,92,93,5,94,95,32,98
HelpS:   DB  9,1,96,97,5,26,25,75,32,98

;*************************************************************
;       Help File Messages
;*************************************************************

HlpLR:   DB 6,'=Next(Esc=Exit)'  ;  1
         DB 'Main Menu Help:-'   ;  2
         DB 'Step to Option &'   ;  3
         DB 'press Enter key.'   ;  4
         DB 'Key functions:- '   ;  5
         DB 6,'=Next Option   '  ;  6
         DB 7,'=PreviousOption'  ;  7
         DB 'Ent=SelectOption'   ;  8
         DB 'Esc=Top of Menu '   ;  9
         DB 'CODE=Edit Code  '   ; 10
         DB 'DATA=Edit Data  '   ; 11
         DB 'SFR=Edit SFRegs.'   ; 12
         DB 'XDAT=Edit Xdata '   ; 13
         DB 'STO=Store Da&Var'   ; 14
         DB 'OFF=Store + OFF '   ; 15
         DB '0-F=Quick Select'   ; 16
         DB 'EXE=Run/Continue'   ; 17
         DB 'Key in Add&Data.'   ; 18
         DB ' C:0000-7FFF is '   ; 19
         DB 'also X:8000-FFFF'   ; 20
         DB 'C:5000 - 7FFF is'   ; 21
         DB 'write protected.'   ; 22
         DB 'Unprotect using '   ; 23
         DB 'menu option  E  '   ; 24
         DB '0-F=ReplaceDigit'   ; 25
         DB '</>=Move Cursor '   ; 26
         DB 6,'/',7,'=Inc/Dec Add.'   ; 27
         DB 'ENT=Display Data'   ; 28
         DB 'HxBn=Dsp Hex/Bin'   ; 29
         DB 'CODE/DATA/SFR...'   ; 30
         DB '../XDAT=Reselect'   ; 31
         DB 'Esc=To Main Menu'   ; 32
         DB 'Only significant'   ; 33
         DB 'in SS, BP or Exe'   ; 34
         DB ' running modes. '   ; 35
         DB 'Display user Pgm'   ; 36
         DB ' pre-interrupt  '   ; 37
         DB 'SFRs & ID:00-17H'   ; 38
         DB 'to Inspect&Edit.'   ; 39
```

```
        DB 6,'/Ent=Next Reg. '   ; 40
        DB 7,'=Previous Reg. '    ; 41
        DB 'Ent=Save current'    ; 42
        DB 'IData & System..'    ; 43
        DB 'variables in mem'    ; 44
        DB 'Esc=No Saving...'    ; 45
        DB 'Restore previous'    ; 46
        DB 'saved IData&Vars'    ; 47
        DB 'Ent=Restore...  '    ; 48
        DB 'Esc=No Restore..'    ; 49
        DB 'Key in Start Add'    ; 50
        DB 'Set SW 1&2 ON to'    ; 51
        DB 'exe program. Use'    ; 52
        DB 'Esc to interrupt'    ; 53
        DB 'Execute a single'    ; 54
        DB 'prog step & exit'    ; 55
        DB 'Exe/Ent=Continue'    ; 56
        DB 'Key in BrPnt Add'    ; 57
        DB 'A BP is a 2 Byte'    ; 58
        DB 'Hi&Lo Code Add. '    ; 59
        DB 'End table =00 00'    ; 60
        DB 6,'/',7,'=Inc/Dec BPNo'    ; 61
        DB 'Will exe program'    ; 62
        DB 'till PC=Any BrPt'    ; 63
        DB 'Clears internal '    ; 64
        DB 'Data memory only'    ; 65
        DB 'ENT=Clear Da&SFR'    ; 66
        DB 'ESC=No clearing.'    ; 67
        DB 'Clear User Code '    ; 68
        DB 'C:0000 to C:4FFF'    ; 69
        DB 'ENT=Clear Code  '    ; 70
        DB 'Check Con.cable.'    ; 71
        DB 'Match Baud Rate.'    ; 72
        DB 'Check data is in'    ; 73
        DB 'INTEL HEX format'    ; 74
        DB 'ENT=Continue    '    ; 75
        DB 'Move cursor to  '    ; 76
        DB 'select Baud Rate'    ; 77
        DB ' 3=300   6=600  '    ; 78
        DB '12=1200  24=2400'    ; 79
        DB '48=4800  96=9600'    ; 80
        DB 'ENT=Select BaudR'    ; 81
        DB 'Clears entire...'    ; 82
        DB 'BreakPoint Table'    ; 83
        DB 'Toggles between '    ; 84
        DB 'write protect or'    ; 85
        DB 'write unprotect '    ; 86
        DB 'ENT=Toggle P/UnP'    ; 87
        DB 'ESC=Leave as is.'    ; 88
        DB 'Save current    '    ; 89
        DB 'IData&Vars & OFF'    ; 90
        DB 'For no save, re-'    ; 91
        DB 'store IData&Vars'    ; 92
        DB 'Then SAVE + OFF.'    ; 93
        DB 'ENT/OFF=Save+OFF'    ; 94
        DB '4=Restore IData '    ; 95
        DB 'Key in Digits as'    ; 96
        DB ' required & Ent.'    ; 97
        DB 'End of Help.....'    ; 98

;*************************************************************
```

## 10.19. The Interrupt 0 Subroutine

All the hardware interrupts used by the Operating System, and the Single Step and Breakpoint software interrupts, make use of the External Interrupt 0. Free access must be allowed for a user to all his interrupt subroutines. Careful manipulation of the current byte values was therefore required to not corrupt any of the user bytes or registers, when this interrupt subroutine is called. Two spare bits in the LCA Control Register, together with the Interrupt 0 flip-flops in the LCA, help the software to identify the source from where the interrupt request occurred. The bit allocation for the Control Register is as follows:

**Control Register Bit Functions** (Address X:F200H)

D0 OPS bit: 0 = Swap interrupt vectors

D1 WP bit: 0 = Operating System is write protected

D2 OFF bit: 1 = System power OFF

D3-D5 (not used)

D6 KB bit: 1 = Busy with the Keyboard Read subroutine

D7 RUN bit: 1 = In a Running Mode, 0 = in Edit Mode

The Interrupt subroutine had to cater for all external interrupt sources while in the Operating System: This includes the 'Escape' latch interrupt when the 'Escape' key was pressed during the execution of a user program; the Running Mode Selection switches switched from the Execution mode to one of the other Running Modes; and also the software interrupt when running the user program in one of the monitor modes.

The routine must also test if there were any other user interrupt requests and whether they were enabled. Their interrupt addresses will then be pushed into the stack as the return address. In this way the routine will *simulate* a user interrupt call that is normally done by the micro-controller internal hardware.

## Interrupt Vector Swapping

When an interrupt is requested, the micro-controller will automatically call the appropriate interrupt vector or address in the code memory. A different set of interrupt subroutines had to be used for the Operating System that would not interfere with the user interrupt routines. A programmable bit in the LCA, the OPS bit, will cause the swapping of the code memory addresses between 0000H and 002FH with addresses 5000H to 502FH through hardware decoding. It will simply invert the logic levels of addresses **A12** and **A14**.

The default or reset value of the OPS bit is '0', so when the micro-controller calls address 0000H, the memory decoding will actually return the address 5000H. A "Long Jump" will place the control into the Operating System. All the interrupts used by the Operating System will generate the normal interrupt addresses between 0003H and 0023H, but the code memory decoding will also return the contents of the addresses between 5003H and 5023H, instead.

Before control is handed over to the user program in one the Running Modes, the OPS bit is set to *not* swap the starting and interrupt vector addresses. The user program can then be executed from address 0000H and all the interrupt vectors are available at their normal addresses.

When the 'Esc' key is pressed, or one of the switches switched to either the *Single Step* or *Break Point* running modes during the execution of a user program, a latch in the LCA, the "Escape Latch", will swap the addresses 0000H to 0007H with 5000H to 5007H. It will also generate an External Interrupt 0. The Operating System's Interrupt 0 subroutine will therefore be executed, in place of the normal user interrupt. A "Long Jump" will take the control out of the swapped interrupt memory area into the Operating System area.

The Escape Latch is cleared by reading any data from the keyboard buffer at external address F300H. At that stage, control will already be out of the swapped interrupt vector area. The pre-interrupt register values will be stored, the 'OPS' bit cleared and the control will go to Mode 2 of the Operating System. When the 'Exe' key is pressed, the pre-interrupt register values will be reloaded back into the micro-controller, the 'OPS' bit will be set. A normal stack "Return from Interrupt" will jump back to execute the next instruction of the user program.

## Single Step and Break Point Running Mode Operations

The Single Step and Break Point operation relies on the fact that an interrupt will not occur directly after a "Return from Interrupt" instruction, or after any other instruction that accesses any of the interrupt control registers. The interrupt will only occur *after the next* instruction has been executed. The External Interrupt must be set to high priority and edge triggering.

To execute only a single user instruction and return to the Operating System immediately, the user program return address must be pushed into the top of the stack. The Interrupt 0 request flag is set by software and the interrupt is enabled. A "Return from Interrupt" instruction will then return control to the user program. Both these last two instructions will ignore the enabled interrupt triggered by the software setting of the interrupt request flag.

One complete instruction will be executed before the interrupt request will take effect. Control is then handed back to the Operating System in the Single Step Mode, or the return address is compared with the addresses in the break point table. If a match is found, control goes to the Operating System. If not, the register contents of the micro-controller are restored, an interrupt triggered, and the next user instruction executed.

None of the SFRs or the internal data of the micro-controller will be corrupted by a *Single Step*, *Break Point* or an *Escape* break, except for nine stack bytes above the point that the stack was at that stage of the user program. If any of the timers were running, they will be stopped temporarily, and restored again when control goes back to the user program. Only a few counts will be lost.

## Simulation of Other User Interrupt Requests

In the Single Step or Break Point running mode, Interrupt 0 at highest priority, is the only true interrupt that is executed. To enable other interrupt requests in the user program as well, their interrupt flags are polled by the Interrupt 0 subroutine of the Operating System. If set and enabled, the user interrupt return

address will be pushed into the top of the stack and it's associated interrupt flag cleared. When control returns to the user program, the user interrupt subroutine will be executed.

Under these conditions, the user interrupts are executed as normal program instructions, and no *normal-* or *higher priority* blanking occurs. Another interrupt request may now interrupt the program while it is still busy with a current interrupt subroutine. This deviation from normal operation is a small price to pay to enable single stepping through the user interrupt subroutines as well. In the *Execute Mode*, however, the *normal-* and *higher priority* blanking is maintained.

In the Single Step and Break Point modes, the OPS bit stays cleared to swap the interrupt vectors. The user interrupt simulation is done by placing the user interrupt vector address *plus* 5000H in the stack. When the program returns, the swapped memory areas will return the *user code*, now decoded between 5000H and 502FH. The OPS bit is still cleared.

This partial swapping may create a problem if a user interrupt subroutine steps over the 002FH and 0030H boundary, or when a relative or absolute jump is used. The calculated program counter value will return the wrong code. If a "Long Jump" is used instead to jump out of the interrupt vector area, the complete and correct address is specified in the code and the instruction at the correct address will be executed.

Programs compiled by a C-compiler usually make use of a jump instruction at all the interrupt addresses and place the actual subroutine somewhere higher in the code. When writing a program in assembler, and an interrupt subroutine is longer than the allowed eight bytes, the user should also adopt this method.

## A User External Interrupt 0 Pin, IRQ

Provision has been made for a user hardware Interrupt 0 input on the 'IRQ' pin of the edge connector, in place of the normal P3.2 pin. A high-to-low transition on the IRQ pin, will set a latch in the LCA. It will pull pin P3.2 low and generate a normal interrupt. There are three latches in the LCA that can pull pin P3.2 low: One is activated by the pressing of any key, and reset when a keyboard read instruction is executed. The second latch is activated by the IRQ pin, and reset when a write instruction is executed to the LCA Control register. This write instruction must follow a read instruction from the same address, to rewrite the same previous data and not upset the 'OFF', 'Write Protect' and 'OPS' bits. The third latch is the "Esc key or the SS or BP switch" latch that will also be reset when reading from the Keyboard Register.

To test if the interrupt was generated by either the keyboard or from the user hardware Interrupt 0, or from both, the state of pin P3.2 can be monitored while resetting the Key Latch: If the pin P3.2 does not go high, the interrupt was from the IRQ pin. To test if a simultaneous interrupt also occurred from the keyboard, the DAV bit (Data bit 5) from the keyboard will stay set while a key is being pressed. When implemented, these tests must all be included in the user interrupt subroutine, or directly after it was executed.

## The Interrupt 0 Subroutine Operation

The jump vector for the Interrupt 0 subroutine is located at address 5003h, but it will be decoded as address 0003h when either the 'OPS' bit in the Control Register, or the 'Escape Latch' are set. The program will jump immediately to a location outside the swapped memory area to the rest of the subroutine located in the Operating System code.

Immediately after entering the subroutine, the TCON register will be stored in the stack and the timers disabled to try to maintain and conserve a possible timing process in the user program. The TCON register value will be restored just before the user program continues so that the timers can continue if they were enabled originally. Only a few counts will be lost.

Afterwards, the registers used by the subroutine will also be stored in the stack to conserve the current user values. If the system was busy executing a user program in one of the *Running Modes*, the RUN bit in the Control Register would have been set. This is used to select if this interrupt was a software interrupt or an interrupt from the keyboard while in the Operating System. If it were from the keyboard, the registers will be restored and the control handed back to the keyboard routine with a normal "Return From Interrupt" instruction.

While in one of the *Running Modes*, i.e. RUN was set, the interrupt may be from one of the following sources: It can be a user keyboard routine; an external user interrupt on the 'IRQ' pin; a break created by a software interrupt; or an 'Esc' key break. A user keyboard interrupt must have the KB bit in the Control

Register set to restore the registers and return the control to the user keyboard program. If a user program is run in the *Execute Mode*, the vector memory swapping will not occur for a user keyboard or external interrupt. The user must therefore supply his own interrupt subroutine at the normal vector address of 0003h.

The Interrupt 0 subroutine will make various test to test the origin of the interrupt, as well as the state of the Running Mode Select switches, and act accordingly. If a user interrupt was requested on Interrupt 0 or any of the other enabled sources, the user interrupt vector address will be pushed into the stack. This will simulate a normal hardware controlled interrupt subroutine. When the system is set to the *Break Point* mode, the user program counter value in the stack will be compared with the Break Point Table values. If a match was not found, it will restore the registers and return to execute one more user instruction. Should a Break Point match be found, or in all cases when the Single Step mode was selected, or when the 'Esc' key was pressed, the micro-controller's SFR and internal data that will be corrupted by the Operating System, will be stored in the Pre-Interrupt registers. Even the current displayed characters and the cursor address of the display unit will be stored before control is handed over to the "Edit Pre-Interrupt Register" mode

The Interrupt 0 Subroutine Operating Diagram, the Flow Diagram, and the Assembler Listings are included on the next pages:

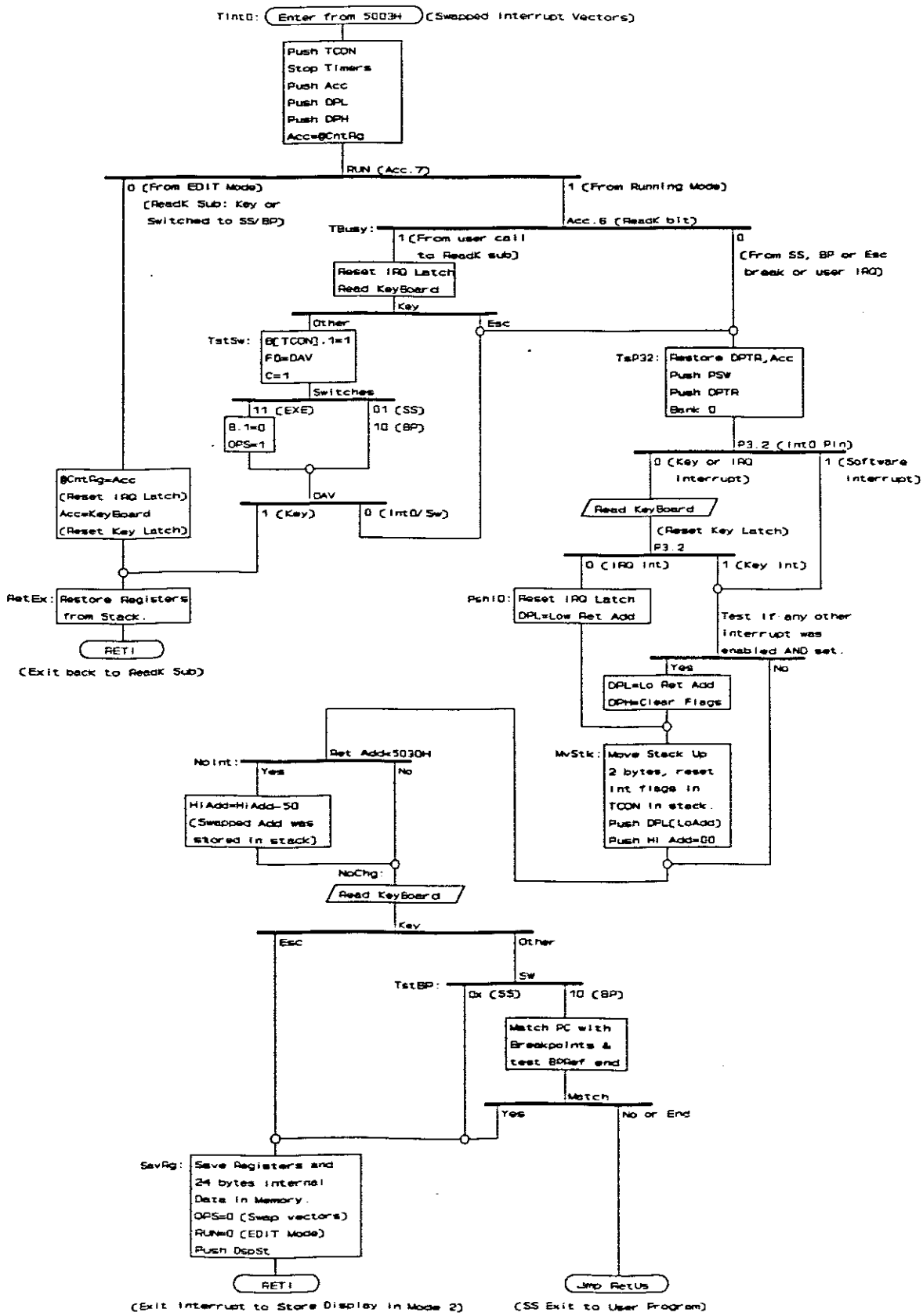



**Fig.10.41. Interrupt 0 Subroutine Operating Diagram**

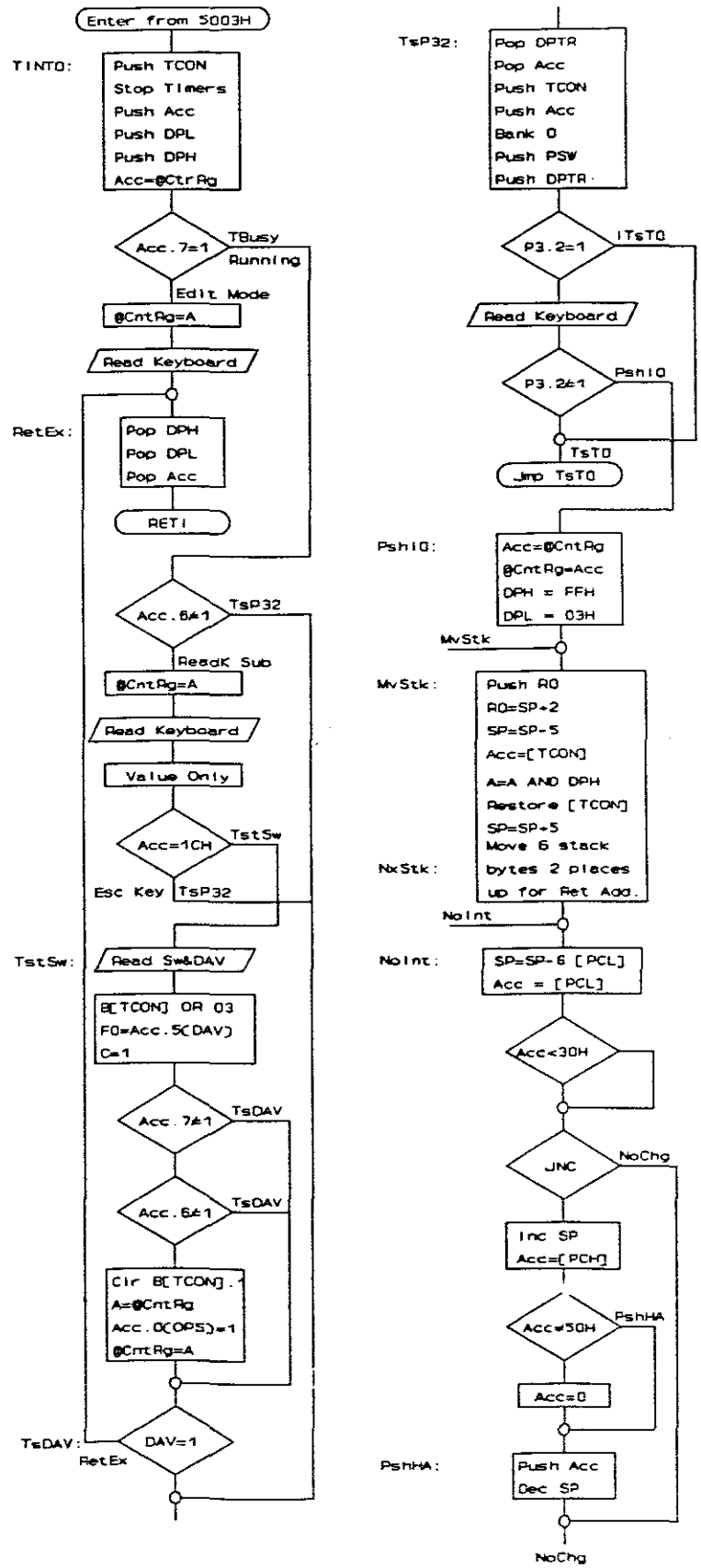



**Fig.10.42. Interrupt 0 Subroutine Flow Diagram (page 1 of 2)**

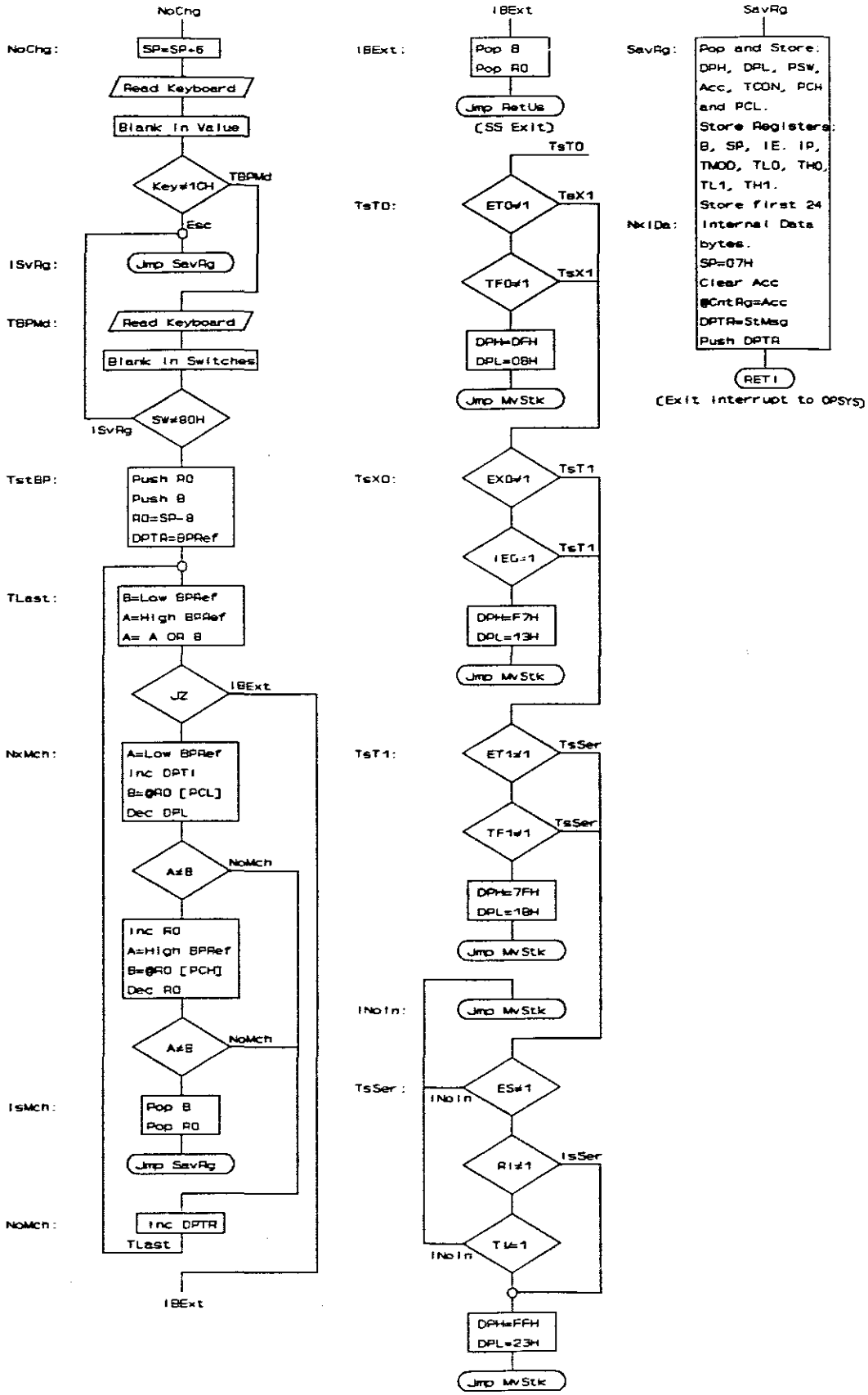



**Fig.10.43. Interrupt 0 Subroutine Flow Diagram (page 2 of 2)**

## Assembler Listing

```
;************************************************************
;  Operating System Reset and Interrupt Vectors
;************************************************************

          ORG 5000H
          JMP START

          ORG 5003H
XINT0:    JMP TINT0
          DB 255,255,255,255,255

          ORG 500BH
T0INT:    RETI
          DB 255,255,255,255,255,255,255

          ORG 5013H
XINT1:    RETI
          DB 255,255,255,255,255,255,255

          ORG 501BH
T1INT:    CLR C
          RETI
          DB 255,255,255,255,255,255

          ORG 5023H
SXINT:    RETI
          DB 255,255,255,255,255,255,255
          DB 255,255,255,255,255

;************************************************************
;         Interrupt 0 Subroutine from 5003H
;************************************************************

;         Enter from Software SS, BP or Esc Key Interrupts
;         only if OPS=0, or if (OPS=1) AND (ESC key pressed OR
;         switches switched to SS or BP).
;         Corrupt in :
;          * EDIT Mode: 6 stack bytes
;          * Running Mode (SS&BP&Esc): TCON.1, 9 stack bytes
;                                                 (+2 IntSub)
;          * User ReadK sub: B.1, C, F0, 6 stack bytes

TINT0:    PUSH TCON
          ANL TCON,#10101111B          ;Stop Possible timers
          PUSH Acc
          PUSH DPL
          PUSH DPH
          MOV DPTR,#CntRg
          MOVX A,@DPTR                 ;Get RUN Bit

          JB Acc.7,TBusy               ;Jump if in Running mode
                                       ;Else, in Edit Mode - Restore
          MOVX @DPTR,A                 ;Reset possible Int0 Latch
          MOV DPTR,#KyBrd
          MOVX A,@DPTR                 ;Read key & reset Latch

RetEx:    POP DPH                      ;Restore registers
          POP DPL
          POP Acc
          POP TCON
          RETI                         ;Exit back to OPSYS/ReadK Sub

TBusy:    JNB Acc.6,TsP32              ;Jmp if KBusy flag not set
```

```
                                    ; Else, busy with ReadK sub
        MOVX @DPTR,A                ;Ignore possible Int0 Latch
        MOV DPTR,#KyBrd
        MOVX A,@DPTR                ;Read Key & Reset Key Latch
        ANL A,#1FH                  ;Blank in Key Value
        CJNE A,#1CH,TstSw           ;Test if Esc key and
        JMP TsP32                   ; go through to SavRg

TstSw:  MOVX A,@DPTR                ;Read SW and DAV
        ORL B,#03H                  ;Set software int on ReadK exit
        MOV C,Acc.5                 ; and ensure Edge Triggered
        MOV F0,C                    ;Store DAV = Acc.5
        SETB C                      ;Indicate Key-int for ReadK
        JNB Acc.7,TsDAV             ;Test mode setting and
        JNB Acc.6,TsDAV             ; if SW  11 (EXE), clear
        CLR B.1                     ; software Int in B[TCON].1
        MOV DPTR,#CntRg             ; to prevent interrupt when
        MOVX A,@DPTR                ; returning from ReadK sub.
        SETB Acc.0                  ;OPS=1 for user int vectors
        MOVX @DPTR,A

TsDAV:  JB F0,RetEx                 ;Return if it was onother key

TsP32:  Pop DPH
        Pop DPL
        Push PSW
        ANL PSW,#11100111B          ;Select Bank 0
        Push DPL
        Push DPH

        JB P3.2,ITsT0               ;Jmp if Software Int

        MOV DPTR,#KyBrd
        MOVX A,@DPTR                ;Read Key
        JNB P3.2,PSHI0              ;Jmp if Usr Int0

ITsT0:  Jmp TsT0                    ;Intermediate Long Jump

PshI0:  MOV DPTR,#CntRg
        MOVX A,@DPTR
        MOVX @DPTR,A                ;Write to reset Int0 Latch
        MOV DPH,#0FFH               ;No alterations to TCON
        MOV DPL,#03H                ;User Sub Low Address

MvStk:  Push 00H                    ;Store R0
        MOV R0,SP
        Inc R0
        Inc R0                      ;To new R0 position
        MOV A,SP
        ADD A,#0FBH                 ;Subtract 5
        MOV SP,A                    ;Point to saved TCON
        Pop Acc
        ANL A,DPH                   ;Clear Int Flag
        Push Acc                    ;Store TCON
        MOV A,SP
        ADD A,#05H                  ;Back to old R0
        MOV SP,A
        MOV DPH,#06H

;       Move Stack up 2 bytes

NxStk:  POP Acc                     ;Get byte & Dec SP
        MOV @R0,A                   ;Store in new place
        DEC R0
        DJNZ DPH,NxStk              ;Move 6 bytes
```

```
          PUSH DPL                    ;Low User Int Add
          CLR A
          PUSH Acc                    ;Hi User Int Add
          MOV A,SP
          ADD A,#06H                  ;Point to new R0 pos
          MOV SP,A
          POP 00H                     ;Restore R0

NoInt:    MOV A,SP
          ADD A,#0FAH                 ;Point to Hi Return Add
          MOV SP,A
          POP Acc                     ;Dec & Pop Lo Return Add
          PUSH Acc                    ;Restore SP
          CJNE A,#30H,$+3
          JNC NoChg                   ;No change if LoAdd not < 30H

          INC SP
          POP Acc                     ;Dec & Pop Hi Return Add
          CJNE A,#50H,PshHA           ;Jmp if not Busy in User Int
          CLR A
PshHA:    Push Acc
          DEC SP                      ;Restore SP

NoChg:    MOV A,SP
          ADD A,#06H
          MOV SP,A

;         Test for Esc key

          MOV DPTR,#KyBrd
          MOVX A,@DPTR
          ANL A,#1FH                  ;Blank in Key Value
          CJNE A,#1CH,TBPMd           ;Test if Esc key was pressed

ISvRg:    Jmp SavRg                   ;Also intermediate Long Jmp

TBPMd:    MOVX A,@DPTR
          ANL A,#0C0H                 ;Blank in Switches
          CJNE A,#80H,ISvRg           ;Jmp if not BP Mode

TstBP:    PUSH 00H                    ;Store R0
          PUSH B                      ;Store B
          MOV A,SP
          ADD A,#0F8H                 ;Subtract 8
          MOV R0,A                    ;Point to PCL
          MOV DPTR,#BPRef             ;Lo BPRef

;         Test if Last BPRef (=0000H)

TLast:    MOVX A,@DPTR                ;Get LoBPR
          MOV B,A
          INC DPL
          MOVX A,@DPTR                ;Get HiBPR
          DEC DPL                     ;DPTR->LoBPR
          ORL A,B                     ;OR with LoBPR
          JZ IBPEx                    ;Test if last BPRef=0000

;         Match Break Point with Program Counter

NxMch:    MOVX A,@DPTR                ;Get Lo BpRef
          INC DPTR                    ;DPTR->HiBPR
          MOV B,@R0                   ;Lo PC
          CJNE A,B,NoMch              ;Test if LoPC=LoBPR
          INC R0                      ;R0->HiPC
          MOVX A,@DPTR                ;Get HiBPR
          MOV B,@R0                   ;Get HiPC
```

```
        DEC R0                          ;[R0]->PCL (Reset R0)
        CJNE A,B,NoMch                  ;Test if HiPC=HiBPR

IsMch:  POP B                           ;Reset Stack to Save Reg
        POP 00H
        JMP SavRg                       ; entrys to SavRg

NoMcH:  INC DPTR                        ;[DPTR]=Next LoBPR
        JMP TLast                       ;Jmp to test if last BP

IBPEx:  POP B                           ;Reset stack to exit
        POP 00H
        JMP RetUs                       ;RETI exit to User Program

;       Test interrupts from T0, X1, T1 and Serial

TsT0:   JNB ET0,TsX1
        JNB TF0,TsX1
        MOV DPH,#11011111B              ;Clear TF0
        MOV DPL,#0BH                    ;User T0 Int Lo Add
        Jmp MvStK

;       Test User External Interrupt 1

TsX1:   JNB EX1,TsT1
        JNB IE1,TsT1
        MOV DPH,#11110111B              ;Clear IE1
        MOV DPL,#13H                    ;User Int1 Lo Add
        Jmp MvStk                       ;Jmp to Move stack up 2x

;       Test User Timer 1 Interrupt

TsT1:   JNB ET1,TsSer
        JNB TF1,TsSer
        MOV DPH,#01111111B              ;Clear TF1
        MOV DPL,#1BH                    ;User T1 Int Lo Add
        Jmp MvStk                       ;Jmp to Move stack up 2x

;       Test User Serial Interrupt

INoIn:  JMP NoInt
TsSer:  JNB ES,INoIn
        JB RI,IsSer
        JNB TI,INoIn
IsSer:  Push Acc
        MOV DPH,#0FFH                   ;No Changes to TCON
        MOV DPL,#23H                    ;User SerInt Lo Add
        Jmp MvStk                       ;Jmp to Move stack up 2x

;       Save Registers before going to Opsystem

SavRg:  MOV DPTR,#MDPH                  ;Memory Store
        POP Acc
        MOVX @DPTR,A                    ;Store User DPH
        DEC DPL
        POP Acc
        MOVX @DPTR,A                    ;Store User DPL
        DEC DPL
        POP Acc
        MOVX @DPTR,A                    ;Store User PSW
        DEC DPL
        POP Acc
        MOVX @DPTR,A                    ;Store User Acc
        MOV DPTR,#MTCON
        POP Acc
        MOVX @DPTR,A                    ;Store User TCON
```

```
        MOV TCON,#01H           ;Reset Ex & Timer Int Flags
        MOV DPTR,#MPC+1
        POP Acc
        MOVX @DPTR,A            ;Store User PCH
        DEC DPL
        POP Acc
        MOVX @DPTR,A            ;Store User PCL
        MOV DPTR,#MBReg
        MOV A,B
        MOVX @DPTR,A            ;Store User B Reg

        INC DPTR
        MOV A,SP
        MOVX @DPTR,A            ;Store User SP
        INC DPTR
        MOV A,IE
        CLR Acc.7               ;Prevent Ints when restoring
        MOVX @DPTR,A            ;Store User IE
        MOV IE,#80H             ;Reset current Interrupts
        INC DPTR
        MOV A,IP
        MOVX @DPTR,A            ;Store User IP
        MOV IP,#01H             ;All Low except PX0
        INC DPTR
        MOV A,TMOD
        MOVX @DPTR,A            ;Store User TMOD
        INC DPTR
        INC DPTR                ;Step over MTCON
        MOV A,TL0
        MOVX @DPTR,A            ;Store User TL0
        INC DPTR
        MOV A,TH0
        MOVX @DPTR,A            ;Store User TH0
        INC DPTR
        MOV A,TL1
        MOVX @DPTR,A            ;Store User TL1
        INC DPTR
        MOV A,TH1
        MOVX @DPTR,A            ;Store User TH1
        MOV DPTR,#MIDat
        MOV A,00H               ;R0 bank 0
        MOVX @DPTR,A            ;Store User R0 Bank 0
        INC DPTR
        MOV R0,#01H             ;Point to 2nd Idata

NxIDa:  MOV A,@R0               ;Read & Store Idata
        MOVX @DPTR,A
        INC R0
        INC DPTR
        CJNE R0,#18H,NxIda      ;Store next 23 bytes

        MOV SP,#07H             ;Define stack for Mem Edit
        MOV DPTR,#CntRg
        MOVX A,@DPTR
        ANL A,#01111110B        ;Clear EDIT and OPS bits
        MOVX @DPTR,A            ;OPS=0 to swap Int vectors
        MOV DPTR,#StMsg         ;Jmp to Store Display data
        PUSH DPL                ; in Mode 2
        PUSH DPH
        RETI                    ;RETI to StMsg & exit interrupt
                                ; mode to enable interrupts

;*************************************************************
```

## 10.20. Restore and Return to Execute the User Program Routine

This routine will be executed when the 'Exe' key is pressed from within the Operating System. It will continue the execution of the user program after a break.

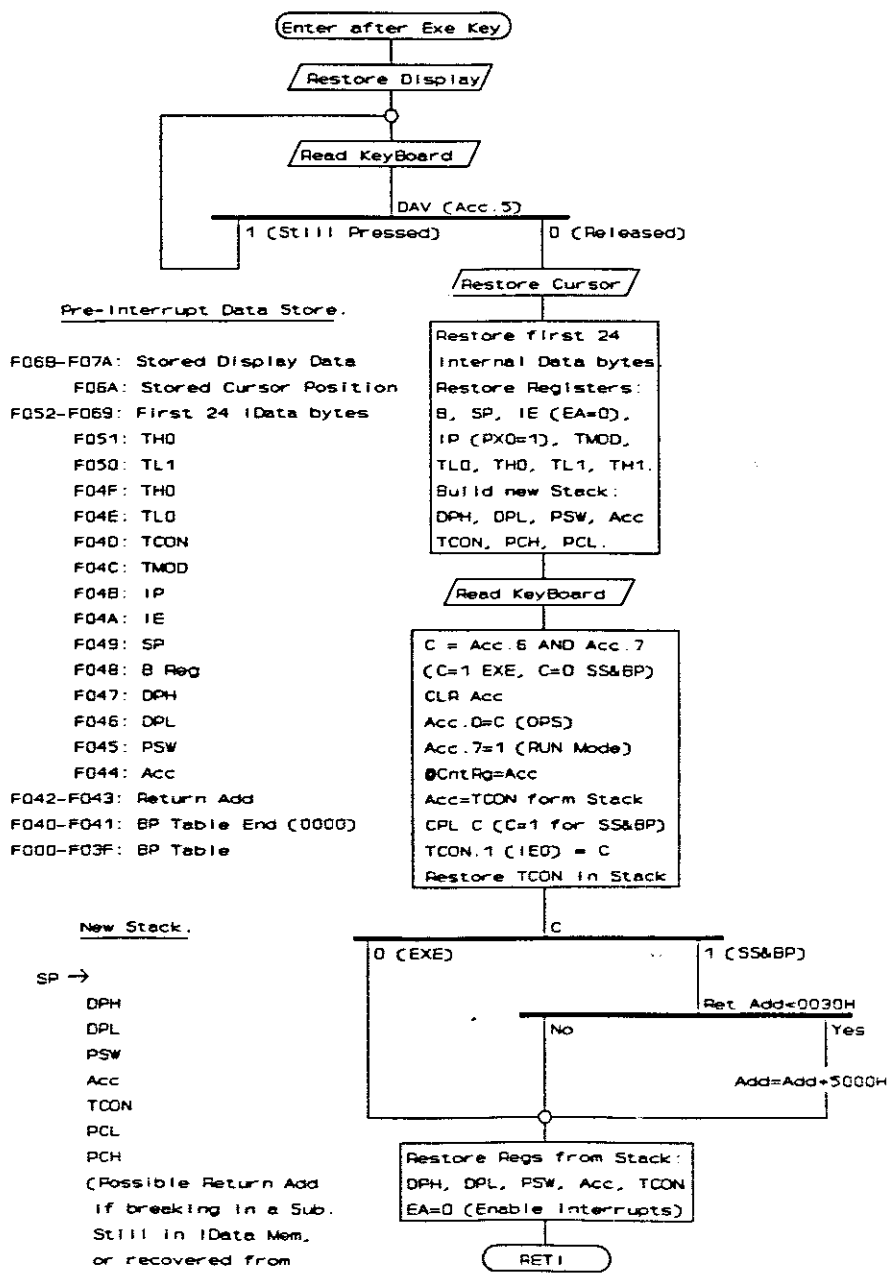



**Fig.10.44. Restore and Return Operating Diagram**

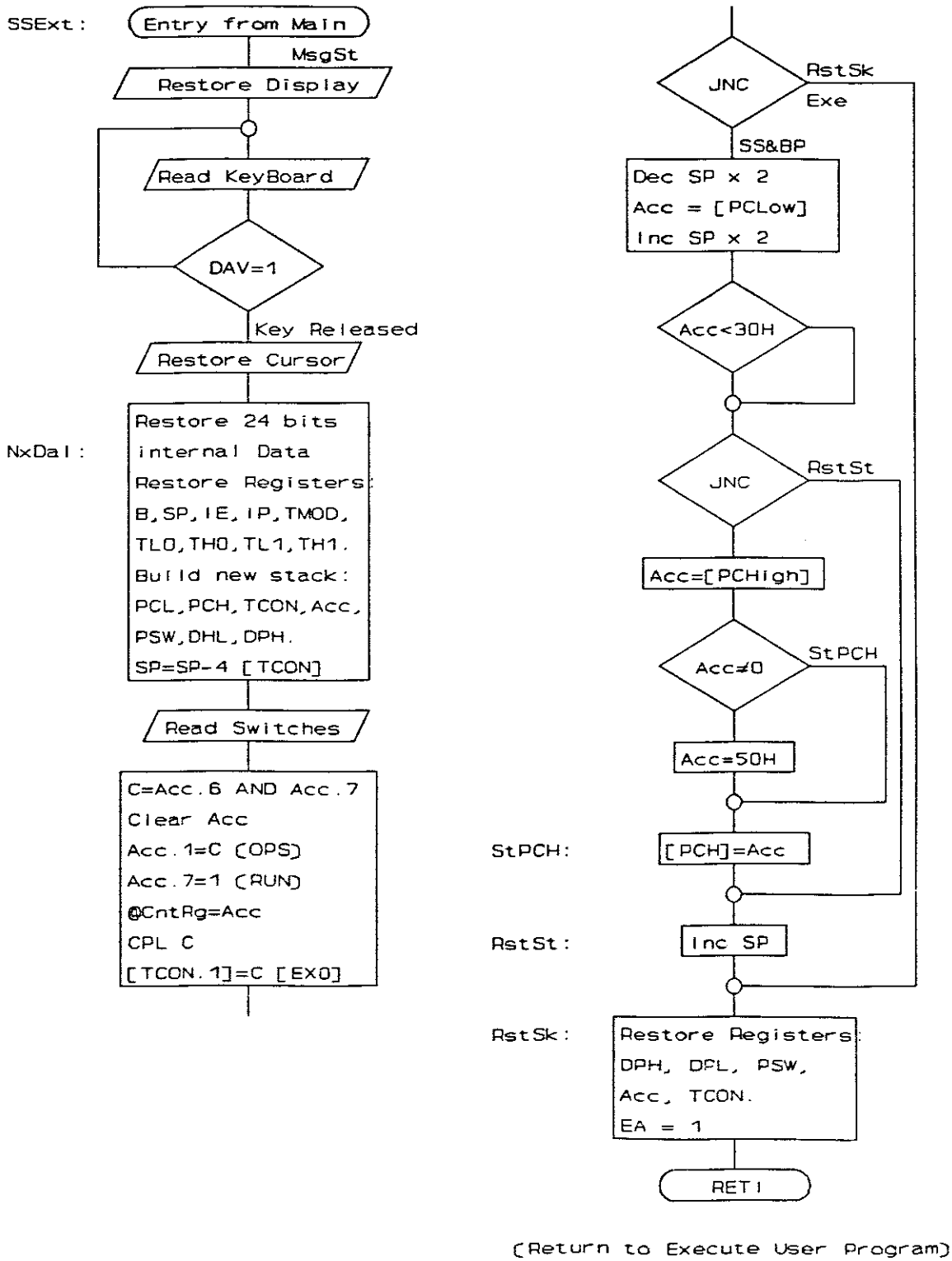



**Fig.10.45. Restore and Return Flow Diagram**

## Assembler Listing

```
;*************************************************************
;       Restore & Exit Routine
;*************************************************************

SSExt:  ANL PSW,#11100111B          ;Set Bank 0

;       Restore Previous Display Message & Cursor

        MOV DPTR,#MsgSt             ;'(Previous Message)'
        CALL DspMg                  ; at the Xdata Address
        MOV DPTR,#KyBrd
        MOVX A,@DPTR
        JB Acc.5,$-1                ;Wait for key release
        MOV DPTR,#DsCSt             ;Get old Cursor Add
        MOVX A,@DPTR
        MOV DPTR,#DspCm
        CALL WrDsp                  ;Replace CurAdd


;     Restore first 24 Internal Data bytes

        MOV R0,#17H                 ;23 bytes
        MOV DPTR,#MIdat+23

NxDaI:  MOVX A,@DPTR
        MOV @R0,A
        DEC DPL
        DJNZ R0,NxDaI
        MOVX A,@DPTR                ;Get R0
        MOV R0,A

;       Restore SFRs

        MOV DPTR,#MBReg
        MOVX A,@DPTR
        MOV B,A
        INC DPTR
        MOVX A,@DPTR
        MOV SP,A
        INC DPTR
        MOVX A,@DPTR
        CLR Acc.7                   ;EA = 0
        MOV IE,A
        INC DPTR
        MOVX A,@DPTR
        SETB Acc.0                  ;Int0 Hi-Priority
        MOV IP,A
        INC DPTR
        MOVX A,@DPTR
        MOV TMOD,A
        INC DPTR                    ;Over TCON
        INC DPTR
        MOVX A,@DPTR
        MOV TL0,A

        INC DPTR
        MOVX A,@DPTR
        MOV TH0,A
        INC DPTR
        MOVX A,@DPTR


        MOV TL1,A
```

```
          INC DPTR
          MOVX A,@DPTR
          MOV TH1,A

          MOV DPTR,#MPC              ;Build New Stack
          MOVX A,@DPTR               ;PCL->Stack
          PUSH Acc
          INC DPTR
          MOVX A,@DPTR               ;PCH->Stack
          PUSH Acc
          MOV DPTR,#MTCON
          MOVX A,@DPTR               ;TCON->Stack
          SETB Acc.0                 ;Set Int0 edge triggered
          PUSH Acc

          MOV DPTR,#MAcc
          MOVX A,@DPTR               ;Acc->Stack
          PUSH Acc
          INC DPTR
          MOVX A,@DPTR               ;PSW->Stack
          PUSH Acc
          INC DPTR
          MOVX A,@DPTR               ;DPL->Stack
          PUSH Acc
          INC DPTR
          MOVX A,@DPTR               ;DPH->Stack
          PUSH Acc


;         Clr/Set OPS, IE0 and add 5000H to Int Vectors if < 003

RetUs:    MOV A,SP
          ADD A,#0FCH                ;Subtract 4
          MOV SP,A                   ;TCON is Top of Stack

          MOV DPTR,#KyBrd
          MOVX A,@DPTR
          MOV C,Acc.6                ;SS bit
          ANL C,Acc.7                ;C=1 for Exe, C=0 for SS/BP
          MOV DPTR,#CntRg            ;Set up CntRg
          MOVX A,@DPTR
          MOV Acc.0,C                ;OPS=1 for Exe, 0 for SS &
          SETB Acc.7                 ;EDIT=1 for all Running Mod
          MOVX @DPTR,A               ;Write to LCA Control Reg
          CPL C                      ;C=1 for SS & BP modes

          JB Acc.6,TRstS             ;No software int for ReadK
          POP Acc                    ;TCON in Acc
          MOV Acc.1,C                ;Set up for Software Int
          PUSH Acc                   ;TCON is top of stack

TRstS:    JNC RstSk                  ;No Change of PC for Exe Md
                                     ;Hi Add => 50H for SS or BP
          DEC SP                     ;  if Add < 0030H
          DEC SP                     ;Point to PCL
          POP Acc                    ;Acc = PC Low
          INC SP                     ;Restore SP
          INC SP                     ;PCH is top of stack
          CJNE A,#30H,$+3            ;Test Acc < 30H
          JNC RstSt                  ;C = 1 if Add < 30H
          POP Acc                    ;A <- PCH
          JNZ StPCH                  ;Jmp if PCH not 00H
          MOV A,#50H                 ;User Int swapped Hi Add

StPCH:    PUSH Acc                   ;Restore PCH
```

```
RstSt:   INC SP                          ;Point to TCON

RstSk:   MOV A,SP
         ADD A,#04H
         MOV SP,A                        ;Stack points to DPH
         POP DPH                         ;Restore User DPH
         POP DPL                         ;Restore User DPL
         POP PSW                         ;Restore PSW and Bank
         POP Acc                         ;Restore User Acc
         POP TCON                        ;Restore TCON and Int Flags
         SETB EA                         ;Enable Interrupts

         RETI                            ;Jump to UsrPg or UsrInts

;*************************************************************
```

## 10.21. Supplementary Subroutines: Delay, Display and Keyboard Drive

Among the rest of the subroutines are a few delay routines, which can delay operation from very short delays from approximately 40 micro-seconds up to one second. These are used for display driving routines and the flashing of messages on the screen. These routines use Timer 0, and the Timer 0 Interrupt subroutine contains only a "Return From Interrupt" instruction.

The Display Module driving subroutine is used extensively by the Operating System, and is also available to the user. All that need to be done by the user, is to define a 16-character data block, and to enter the routine with the address of the first character of the data block in the Data Pointer. The display driving subroutine uses the delay subroutines above, and the necessary timer interrupt subroutines must be included in the user program.

If a user program is run on the board, it is not required to initialise the display. That will be done automatically on power-up. If the user program is going to be used on another board, the initialisation sequence for the display unit must be included. The routines in the Operating System may be used as a template for driving any similar LCD Display Module.

The Keyboard Read subroutine used for the Operating System is a bit more complex than what would normally be required by a system, but it was necessary to cater for all the possible scenarios in using the entire system. The 3-minute auto-off facility is also contained in this subroutine. A timer and the DPTR as a down counter were used. The keyboard subroutine is also available for user

subroutines, by simply calling the correct subroutine address in the Operating System. The starting addresses for useful subroutines in the Operating System are included in the appendix.

When entering the keyboard read subroutine, it will wait for any previous keys to be released first, before it will go into the power conserving idle mode of the micro-controller. The controller will only "wake up" when a key is pressed and it will return with the key value from '00h' to '1Fh' in the accumulator, without waiting for the key to be released. This will create the effect that when a key is pressed, immediate action is taken, and not only after the key was released.

The 'Esc' key and the 'Reset' key should not be implemented in a user program. The 'Esc' key will return control to the Operating System, and the 'Reset' key will cause a system reset.

The user External Interrupt 0 and Timer 0 subroutines to serve the Keyboard subroutine need only contain single "Return From Interrupt" instructions.

**Keyboard Buffer Contents** (Address X:F300H)

| | |
|---|---|
| D0-D4 | The current or the last key value |
| D5 | Data Available (DAV): 1 = Key pressed, 0 = Key released |
| D6 | Single Step Selection switch state - Sw1 (1 if ON) |
| D7 | Break Point Selection switch state - Sw2 (1 if ON) |

The Flow Diagrams and Assembler listing for the above and related subroutines are included on the next few pages:

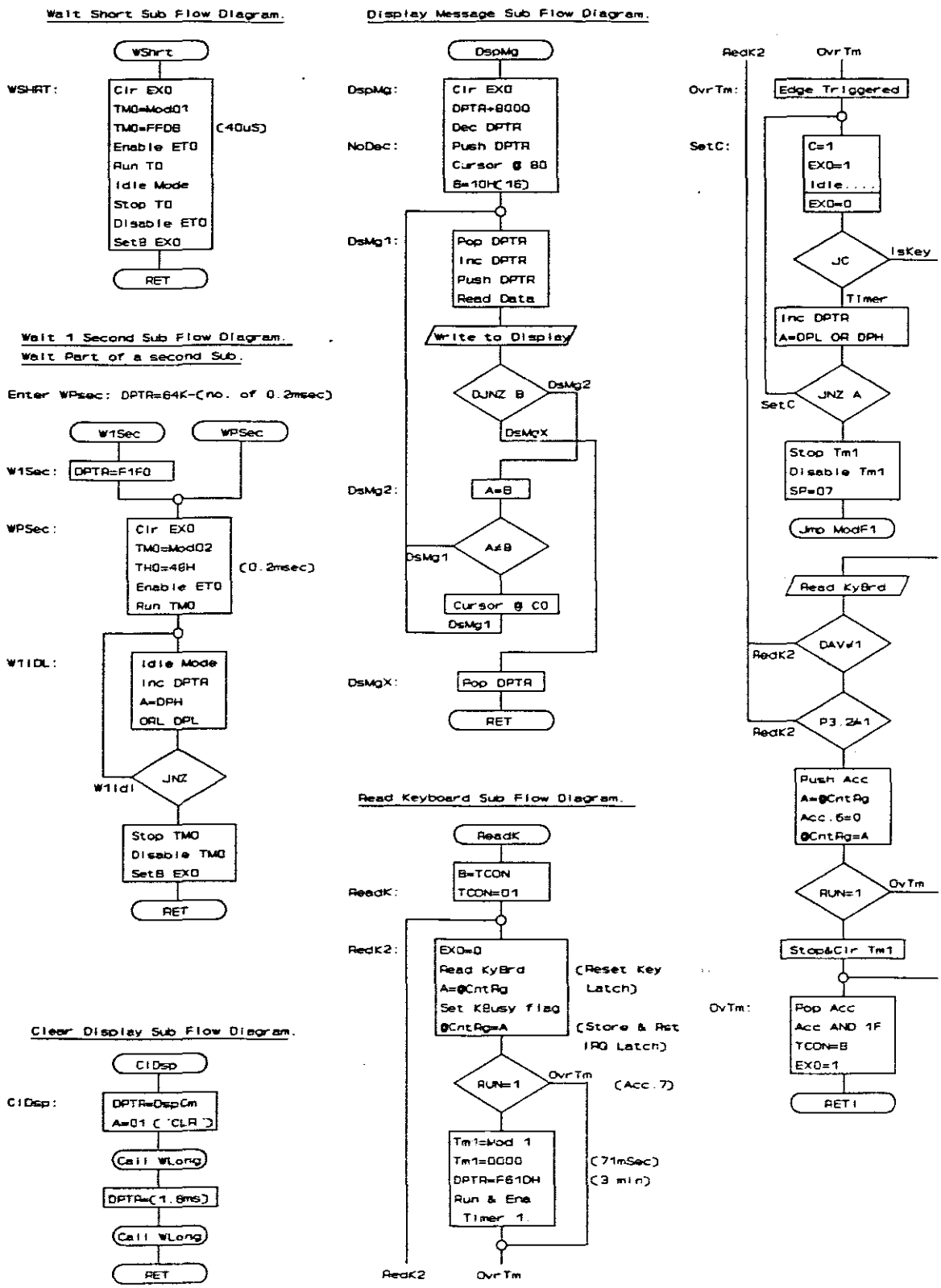



**Fig.10.46. Supplementary Subroutine Flow Diagrams**

## Assembler Listing

```
;*************************************************************
;       Supplementary Subroutines
;*************************************************************
;       Write some Data or Instruction to Display Module
;*************************************************************
;       Wait Short Subroutine (40 µS)
;*************************************************************

;       Corrupt: Tm0, EX0=1
;       Subs & Stack: 4 = 2 (T0Int 2)

WrDsp:  MOVX @DPTR,A

WShrt:  CLR EX0                     ;Disable SS ints
        ANL TCON,#11001111B         ;Stop & Clr Timer 0
        ANL TMOD,#0F0H              ;Reset Timer 0
        ORL TMOD,#01H               ;Timer 0 = Mode 1
        MOV TH0,#0FFH
        MOV TL0,#0DBH               ;40 µS
        SETB TR0                    ;Run Timer 0
        SETB ET0                    ;Enable Timer 0 Interrupt
        ORL PCON,#01H               ;Idle Mode
        CLR TR0                     ;Stop Timer
        CLR ET0                     ;Clear Interrupt
        SETB EX0                    ;Enable software ints
        RETI


;*************************************************************
;    Wait 1 or Wait a Part of a Second Subroutine
;*************************************************************

;    Enter: WPSec with DPTR = 64K - Loop Counter (0.2 msec/count)
;    Corrupt: Tm0, DPTR, ACC, EX0=1
;    Subs & Stack: 4 = 2 (T0INT 2)

W1Sec:  MOV DPTR,#0E284H            ;64K - E284h = 1 sec

WPsec:  CLR EX0                     ;Prevent SS ints
        ANL TCON,#11001111B         ;Stop & Clr Timer 0
        ANL TMOD,#0F0H              ;Reset Timer 0
        ORL TMOD,#02H               ;Timer 0 = Mode 2
        MOV TH0,#48H                ;0.2 msec each
        MOV TL0,#48H
        SETB TR0                    ;Run Timer 0
        SETB ET0                    ;Enable Timer 0 Interrupt

W1IDL:  ORL PCON,#01H               ;Idle Mode
        INC DPTR
        MOV A,DPH
        ORL A,DPL
        JNZ W1IDL                   ;Test if loop done
        CLR TR0                     ;Stop Timer
        CLR ET0                     ;Clear Interrupt
        SETB EX0                    ;Enable possible SS int
        RET

;*************************************************************
```

```
;*************************************************************
;       Clear Display Subroutine
;*************************************************************

;       Corrupt: DPTR, Acc, Tm0, EX0=1
;       Subs & Stack: 6 = 2 (WPsec 2 (TOINT 2)

ClDsp:  MOV DPTR,#DspCm
        MOV A,#01H                  ;'Reset & Clear Instruction'
        MOVX @DPTR,A                ;Write command
        MOV DPTR,#0FFF6H            ;Wait 1.8 mS
        CALL WPSec
        RET


;*************************************************************
;       Display 16 Character Message Subroutine
;*************************************************************

;       Enter: Message Lable in DPTR (Code Address)
;       Corrupts: Acc, B, C, Tm0, EX0=1
;       Subs & Stack: 8 = 4 (WrDsp 2 (TOINT 2)

DspMg:  CLR EX0                     ;Prevent SS interrupts
        ORL DPH,#80H                ;Set DPH.7 to read as Xdata
        DEC DPL                     ;1 less than Message Ref
        MOV A,#0FFH
        CJNE A,DPL,NoDec            ;Jmp if No decrement of DPH
        DEC DPH

NoDec:  PUSH DPH
        PUSH DPL
        MOV DPTR,#DspCm
        MOV A,#80H                  ;Place cursor on 1st Char
        CALL WrDsp
        CLR EX0
        MOV B,#10H                  ;Byte Counter

DsMg1:  POP DPL                     ;Get Message Ref in DPTR
        POP DPH
        INC DPTR                    ;Increment &
        PUSH DPH                    ;  store for next byte
        PUSH DPL
        MOVX A,@DPTR                ;Get Character as XData
        MOV DPTR,#DspDa             ;Display Data Address
        CALL WrDsp                  ;Write Character
        CLR EX0
        DJNZ B,DsMg2                ;Done!
        Jmp DsMgX

DsMg2:  MOV A,B
        CJNE A,#08H,DsMg1

        MOV DPTR,#DspCm
        MOV A,#0C0H                 ;Cursor on 2nd disp line
        CALL WrDsp
        CLR EX0
        JMP DsMg1

DsMgX:  POP DPH                     ;Reset Stack
        POP DPL
        SETB EX0                    ;Enable SS Interrupts
        RET

;*************************************************************
```

```
;*************************************************************
;        Read Keyboard Subroutine & Auto-off Timing
;*************************************************************
;        Exit: Key value in Acc, 3 MSB's masked, (Auto-off)
;        EDIT Mode:
;         Corrupt: Tm1, DPTR, Acc, B, C, EX0=1
;         Subs & Stack: 8 = 2 (XINT0 6), 4 = 2 (T1INT 2)
;        RUN Mode: (User ReadK call)
;         Corrupt: DPTR, Acc, B, C, F0, TCON.1, EX0=1
;         Subs & Stack: 8 = 2 (XINT0 6)

ReadK:   MOV B,TCON                  ;Preserve IE0 for SS&BP if
         MOV TCON,#01H               ; sub is used by a User Call
RedK2:   CLR EX0                     ;Prevent SS & BP Interrupts
         MOV DPTR,#KyBrd
         MOVX A,@DPTR                ;Read key to reset Key latch
         MOV DPTR,#CntRg             ; during this sub
         MOVX A,@DPTR                ;Get RUN bit
         SETB Acc.6                  ;Indicate busy with ReadK sub
         MOVX @DPTR,A                ;Store & Reset Usr Int Latch
         JB Acc.7,OvrTm              ;Don't set Auto-Off if in
         ANL TMOD,#0FH               ; a Running Mode (RUN = 1)
         ORL TMOD,#10H               ;Set Timer 1 to Mode 1
         MOV TH1,#00H                ;Max Timer count = 71 msec
         MOV TL1,#00H
         MOV DPTR,#0F61DH            ;?531 x 71ms = 180 seconds
         SETB TR1                    ;Run Timer 1
         SETB ET1                    ;Enable Timer 1 interrupt

OvrTm:   SETB IT0                    ;Int 0 edge triggered - must
                                     ; release key & press again
SetC:    SETB C                      ;C will indicate which int-sub
         CLR IE0                     ;Clear INT0 flag if set before
         ORL IE,#81H                 ;Enable All and INT0
         ORL PCON,#01H               ;Wait for Key (or Timer)
                                     ;Acc is still @CntRg
         CLR EX0                     ;Prevent Software Int
         JC IsKey                    ;Jump if int was from Pin
         INC DPTR                    ;Else it was from Timer
         MOV A,DPL
         ORL A,DPH
         JNZ SetC                    ;To idle if not terminal count
         CLR TR1                     ;Stop & Disable Timer
         CLR ET1
         MOV SP,#07H                 ;Empty Stack
         JMP ModF1                   ;To direct save and off

IsKey:   MOV DPTR,#KyBrd             ;Keyboard Address
         MOVX A,@DPTR                ;Read Keyboard & Rst Key latch
         JNB Acc.5,RedK2             ;Repete if it was the switches
         JNB P3.2,RedK2              ;Repete if INT0 was Usr Int0
         PUSH Acc                    ;Store Key value

         MOV DPTR,#CntRg             ;Get EDIT bit. Cleared while
         MOVX A,@DPTR                ; in Opsys or Edit Mode.
         CLR Acc.6                   ;Almost done with ReadK sub
         MOVX @DPTR,A
         JB Acc.7,OvTm               ;Over Timer in a running mode
         CLR TR1                     ;Stop Timer 1
         CLR ET1                     ;Disable Timer 1 interrupt

OvTm:    POP Acc                     ;Recover Key value
         ANL A,#1FH                  ;Mask out 3 MSB's(SS,BP & DAV)
         MOV TCON,B                  ;Restore TCON (Software Int)
         SETB EX0
         RETI
```

```
;*************************************************************
;       Binary Nibble to ASCII Conversion Subroutine
;*************************************************************

;       Enter:Nibble in LSN of Acc
;       Exit:ASCII in Acc
;       Corrupt: Acc, C, 2 Stack Bytes

BnASC:  ANL A,#0FH                  ;Mask MSN
        CJNE A,#0AH,$+3             ;Test > 10H
        JC BnAsS                    ;Add only 48 if is
        ADD A,#07H                  ;Correct for A - F
BnAsS:  ADD A,#30H                  ;Add 48
        RET

;*************************************************************
;       ASCII to Binary Conversion Subroutine
;*************************************************************

;       Enter:ASCII in Acc
;       Exit:Nibble in Acc
;       Corrupt: Acc, C, 2 Stack Bytes

AscBn:  ADD A,#0D0H                 ;Subtract 48
        CJNE A,#0AH,$+3             ;Test > 10H
        JC ASBnS                    ;Leave as is if not
        SUBB A,#07H                 ; correct for A - F
ASBnS:  RET

;*************************************************************
;       Messages Listing
;*************************************************************

Msg00:  DB 'Main Menu(',6,',Ent)'   ;Start of Main Menu
Msg10:  DB 'Edit Mem Segment'
Msg20:  DB 'Edit Pre-Int Reg'       ;Pre-Interrupt Reg.
Msg30:  DB 'Store IData&SysV'       ;IData & System Variables
Msg40:  DB 'Restore Data&SyV'
Msg50:  DB 'Exe User Program'
Msg60:  DB 'Single Step Run.'       ;Run Program in Single Step
Msg70:  DB 'Run with BrkPnts'       ;Defining BP Table Reference
Msg80:  DB 'Clear IData Mem.'       ;Internal Data memory
Msg90:  DB 'Clear User Prog.'       ;User Program Area
MsgA0:  DB 'Clear BrkPtTable'       ;Break Point Table
MsgB0:  DB 'RS232 Mem Load..'       ;Serial loading
MsgC0:  DB 'RS232 Mem Dump..'       ;Serisl dumping
MsgD0:  DB 'Xdata Block Move'
MsgE0:  DB 'Prot/Unprot Mem.'       ;Protect/Unprotect Top RAM
MsgF0:  DB 'Save IMem + OFF '       ;Saving Internal Memory + Off
Msg11:  DB 'Data Add:       '       ;Edit Data Segment (Hex)
Msg15:  DB ' SFR Add:       '       ;Edit SFR Segment (Hex)
Msg19:  DB 'XDat Add:       '       ;Edit XData Segment (Hex)
Msg1D:  DB 'Code Add:       '       ;Edit Code Segment (Hex)
Msg21:  DB 'Only SS,BP&ExeMd'       ;Warning for NOT SS,BP or Exe
Msg22:  DB 'Prog. Cntr:     '       ;Program Counter
Msg23:  DB ' PSW  Acc :     '
Msg24:  DB ' DPH  DPL :     '
Msg25:  DB ' SP   B Reg:    '
Msg26:  DB ' IP    IE  :    '
Msg27:  DB 'TCON  TMOD:     '
Msg28:  DB ' TH0  TL0 :     '
Msg29:  DB ' TH1  TL1 :     '
Msg31:  DB 'Enter to Save...'       ;Confirm Saving
Msg32:  DB 'Saving Data.....'       ;Busy....
Msg33:  DB 'Data&Vars saved!'       ;Done!
Msg41:  DB 'Enter to Restore'       ;Confirm Restoring
```

```
Msg42:   DB 'Restoring Data..'      ;Busy...
Msg43:   DB 'Data restored!  '      ;Done!
Msg51:   DB 'Sw 1&2 on & Ent.'      ;Setting Switches for Exe.
Msg53:   DB 'Executing Prog..'      ;Busy...
Msg61:   DB 'Set Sw1 on & Ent'      ;Switches for Single Step run
Msg71:   DB 'BrkPnt No  :    '      ;Display & Edit Breakpoints.
Msg72:   DB 'Set Sw2 on & Ent'      ;Switches for Break Point run
Msg81:   DB 'Ent to Clr IData'      ;Confirm Clearing IData
Msg82:   DB 'Clearing IData..'
Msg83:   DB 'IData cleared!  '      ;Done!
Msg91:   DB 'Ent to Clr UsrPg'      ;Confirm Clearing User Program
Msg92:   DB 'Clearing program'      ;Busy..
Msg93:   DB 'Program cleared!'      ;Done!
MsgA1:   DB 'Ent to Clr BkPts'      ;Confirm Clearing Break Points
MsgA2:   DB 'Clearing BPTable'      ;Busy..
MsgA3:   DB 'BP Table cleared'      ;Done!
MsgB1:   DB 'Select Baud Rate'      ;Flash Baud Rate Request
MsgB2:   DB '3  6 12 24 48 96'      ;Baud Rate Options
MsgB3:   DB 'RS232 Load Ready'      ;Wait for Serial Train
MsgB4:   DB 'RS232 Loading...'      ;Loading Serial Data
MsgB5:   DB 'Loading Done!   '
MsgB6:   DB 'Load Aborted....'
MsgB7:   DB 'Checksum Error! '
MsgC3:   DB 'Dumping Memory..'
MsgC4:   DB 'Dumping done!   '
MsgC5:   DB 'Dumping Aborted!'
MsgD3:   DB 'Moving Block....'
MsgD4:   DB 'Block Move Done!'
MsgD5:   DB 'Too Many Bytes! '
MsgE1:   DB 'Enter to Protect'      ;Protect Memory
MsgE2:   DB 'Enter to Unprot.'      ;Unprotect Memory
MsgF1:   DB 'Confirm Save+OFF'      ;Confirm Save then OFF

Msg52:   DB 'Exe Prog. C:    '      ;Start Code Add (Same Hi Add)
MsgC1:   DB 'Start Add X:    '      ;Request Xdata Start Add
MsgC2:   DB 'No of Bytes:    '      ;Request Number of Bytes
MsgD1:   DB 'SourceAdd X:    '      ;Request Source Address
MsgD2:   DB 'Dest. Add X:    '      ;Request Destination Addres

MsgI1:   DB 'Cape Technikon  '      ;Initialisation Messages
MsgI2:   DB 228,'C Trainer V:4.3 '

;************************************************************
```

# 11. Power Economy and Conservation

The onboard battery for the system is a standard 9 volt PP3 size battery. The aim of this battery is to enable a user to run the board for short periods, only when an external power source is not available. To supply the board for extended operation, external power source connections were included on the board.

Throughout the hardware and software design of the system, power conservation was kept in mind continuously. Default signal levels were chosen to use the minimum power. All the delay and waiting routines in the software were designed to place the micro-controller in the low power consuming *Idle* mode.

For all the delay subroutines, a timer was set to run from a specific preset value before the controller was placed in the idle mode. As soon as the rollover interrupt occurred, the controller would wake up and return from the subroutine.

The entire Operating System is menu-driven and controlled by the pressing of keys. While the system is waiting for a key to be pressed, it was also placed in the idle mode. If one considers the speed at which the controller operates compared to the speed a user can read a displayed message and press a key, the controller will be in the Idle mode for almost all of the time, waiting for a key to be pressed. The keyboard routine uses a hardware generated external interrupt to wake up from the Idle mode.

For the hardware design, various power conservation tactics were attempted: One possible way to reduce power, is to lower the oscillator frequency of a system. To be able to generate various accurate baud rates, the frequency had to stay at the readily available crystal frequency of 11.059 MHz. The free running oscillator of the keyboard scanning, however, could be reduced by a factor of 10. Other modifications that were possible, was to increase all the pull-up and pull-down resistors from 10 kΩ to 33 kΩ, and to replace the LM7805 fixed voltage regulator with an LM317 adjustable voltage regulator. This made quite a difference: The quiescent current through the ground pin of the LM7805 is 4 mA and is basically wasted power. It was replaced by the quiescent current of the LM317 of approximately 50 $\mu$A, which made a difference of approximately 10% to the original average current dissipated by the board.

The inclusion and current dissipation of the 'ON' LED was under careful consideration. It was eventually placed in series with the opto-isolator internal LED that had to be *on* in any case to maintain the power to the board. The 'ON' LED could therefore stay. The "Busy Configuring" LED is only used for a short time during configuration, and for the convenience of this indicator, it was kept.

All the power limiting modifications reduced the average supply current of the first prototype board from 33 mA to 22 mA on the final boards, a factor of 33%!

---

# 12. The Hardware and Software Debugging and Performance Tests

## 12.1. Hardware Integrity and Power Supply Control Tests

Exhaustive circuit integrity testing was done by comparing each hardware connection directly with the netlist of the Printed Circuit Board, and testing for any possible short circuits between adjacent connections. With all ICs removed from the board, current limited power was applied slowly to test the voltage regulating and on/off switching of the circuit. All the ICs, except the micro-controller, were then inserted and the power brought up again slowly to test for any unwanted over-currents.

## 12.2. Fault Finding the Rest of the Hardware

In any digital circuit with a complexity such as this board, special consideration had to be given to ways and means to test the hardware. The fact that most of the hardware was embedded in the LCA made it even more difficult. The chances of a system not working for the first tests, is virtually 100%. The high speed at which the system operates, and not being able to test intermediate signal levels inside the LCA, prompted the design and construction of a testing device what can be called a **Manual Input/Output Micro-Controller Emulator.** It simulates the input and output operations of the micro-controller by means of hardware switches and LEDs.

## 12.3. The Manual Input/Output Micro-Controller Emulator

Direct connections to all the relevant pins of the micro-controller are available on the 62-pin IBM motherboard type edge connector. A blank IBM prototype development board was used to design and build the **Manual Emulator.** It made use of toggle switches to place the required logic levels on selected signal lines as *outputs* from the simulated micro-controller. Buffer-driven LEDs were used to monitor the relevant simulated *input* signal levels.

The same bi-directional Input/Output pin principle used by the micro-controller was adopted for the Manual Emulator: A switch will connect a signal line hard to ground to generate a 'low'. When the switch is open, a pull-up resistor of 10 kΩ will supply the line to generate a 'high'. When the line is high, external circuitry can pull the line down to ground. A 'low' will then be detected on the input from the circuit, or a 'high', if *not* pulled to ground. The monitoring LEDs were driven by buffers to reduce the loading on the signal lines. The relevant signal lines controller and monitored were the 16 address lines from **A0** to **A15**, the 8 data lines, and various control lines. They are summarised below:

**Outputs** (Controlled by switches and pull-up resistors)

| | |
|---|---|
| A8 - A15 | High Address Byte (Connected to Port 2) |
| PSEN | Program Store Enable |
| ALE | Address Latch Enabled |
| RD | External Data Read Control (Pin P3.7) |
| WR | External Data Write Control (Pin P3.6) |

**Bi-Directional** (Controlled by switches and Monitored by LEDs)

A0 - A7 Low Address Byte (From Address Latch to Memory)

D0 - D7 Data lines (Port 0)

**Inputs** (Monitored by LEDs)

INT0 External Interrupt 0 Pin (Pin P3.2: From LCA)

INT1 External Interrupt 1 Pin (Pin P3.3: From LCA)

RX Serial Receive (Pin P3.0: From MAX232)

T0 Timer 0 (Pin P3.4: From MAX232)

The basic circuits for individual pins of each group of inputs, outputs or bi-directional pins, are shown below:

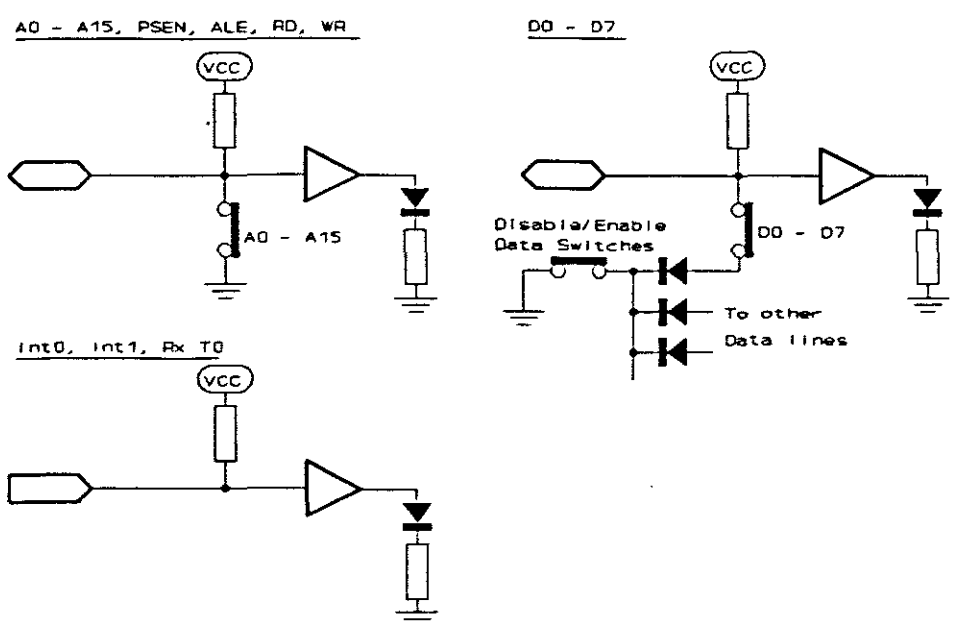



**Fig.12.1. The Manual Emulator Basic Circuits**

This circuit enabled the testing of all the hardware, by simulating the address multiplexing and the reading and writing operations of the micro-controller. The

expected results were monitored at a controlled pace and cross checked after each step.

## 12.4. The Hardware and Firmware Testing

With the micro-controller removed from its socket, the board was *seeded* and the Manual Micro-Controller Emulator testing device plugged into the extension edge connector. By pressing the 'ON/Configure' button, the LCA was configured and the hardware testing of the input and output devices could commence:

The Low Address line circuits were only used as monitors, and all the switches were switched to the inactive '1' levels. A Low Address Byte value was placed on the Data Lines and the ALE signal strobed high. The latching of the data on to the Low Address Lines could now be monitored by the Low Address LEDs. The High Address Byte was set by its switches and placed on the address lines as well.

To read a byte from the Code Address, the Data Control switches were disabled, and the PSEN control line taken to active low. The stored data from the memory would then appear on the Data LEDs. Various spot checks were made to confirm that the correct configuration and operating data bytes were at the correct addresses. Similarly, any data byte can be *written* to in the 64K external data map by activating the WR line, or *read* from memory by activating the RD line. The address had to exist and it should not be write protected.

Using this device, the correct decoding of the LCA to the two RAM IC's, and also to the Input/Output devices, such as the display unit and keyboard, was tested.

The keyboard action to give an interrupt signal, the correct key value and the state of the *Running Mode* switches, proved to react as expected. The setting and resetting of the three LCA internal interrupt latches were also tested to perform as planned.

After all the hardware was adjusted and corrected, the micro-controller could be inserted in its socket to test the software.

## 12.5. Software Testing and Debugging

The Operating or Broad Based Flow Diagrams proved to be very useful in the testing of the software. By using a coloured pen, one could test and mark every possible pathway that can be followed on the diagram. The marking ensured that all the pathways were tested. Notes of all errors or responses that were not quite satisfactory, were made directly on the diagrams. From there it was easy to trace the errors directly to the Single Line Flow Diagrams and the original Assembler listing by means if the chosen label names. Adjustments to display messages and the timing thereof were made to enable the best and most userfriendly interaction with the system.

A simple user test program was written in Assembler. It was specifically designed to test most of the features of the board and the Operating System. It tested for the correct execution of the user interrupt subroutines. The existing Operating System subroutines, the uploading though the RS232 serial port, and the *Single Step-*, the *Break Point-* and the *Execute Running Modes* were also tested. A complete report on the testing of this user program is included in the section on the

pre-requisites and guidelines for writing of a user program. This section also includes notes on what to expect when running and testing such a program.

Quite a few modifications to the original user program monitoring strategy were made that only became apparent during the application in a classroom situation. Only the *final* pre-requisites and guidelines for user programs and performance tests are included in this report.

## 12.6. Classroom Application

A production model of the board was designed and fifteen sets built. A course on the operation and application of the 8051 micro-controller was offered and each student was issued with a personal board for the duration of the course. For most of the students attending the course, micro-controllers were a complete new concept and they had to start from the very basics. It was assumed that the *visibility* of the internal registers and data of the micro-controller through the Operating System of the board would make the understanding of the operation of the controller much easier. It might have to some extend, but the general impression left was that initially the complexity of the board was rather seen as a thread. The board was left untouched by some students until much later in the course. Some students, who had previous experience of micro-controllers, adapted to the use of the board with ease. They could appreciate the flexibility and freedom to inspect the internal registers of the micro-controller and the memory segments, and especially the Single Step debugging feature.

Initially, the system was designed that the user programs only started at address 2000h, while the Operating System occupied the first 8K bytes of code memory. This strategy was chosen mainly to enable the use of the interrupt vectors by the Operating System. A user program had to be designed to start at address 2000h, and its interrupt vectors placed at 2003h, 2013h, etc. For programs written in assembler, this was easily achieved by using "ORG" statements at the beginning of the program and at the beginning of each interrupt subroutine. The "ORG" directive would redirect the Assembler to place the statements following from that specific code address, upwards.

To enable the use of the interrupts for both the Operating System and for user interrupts, the Operating System contained jump vectors to the *Operating System* interrupt routines, while in the "Edit" mode. When in a "Running Modes", the jump instructions will move the control over to the equivalent address of the *user program* section between 2003h and 202Fh. This system worked well enough, as long as the user had proper control over the placement and the content of the interrupt vectors. This is relatively easy when writing the modules in the Assembler language.

When the course got to the stage of writing program modules in the programming language "C for Micro-Controllers", endless problems started to appear to convince the C-Compiler and Linker programs to place the user modules to start at address 2000h. All the Interrupt subroutines had to be written in Assembler to force the Linker to place them at the appropriate higher addresses when they were linked to the main C-program module.

These problems lead to the redesign of the Operating System and user program interaction, and eventually to the firmware modification of the interrupt memory segment swapping. The decoding modification to the firmware was done with relative ease, because it was re-programmable. The only hardware modifications that had to be made, were the cutting of two tracks of the **A12** and **A14** address lines from the micro-controller, and the inclusion of two wire jumpers to connect these address lines to two other pins of the LCA. Two of the original four general purpose "Chip Select Outputs" from the LCA had to be sacrificed.

It was also found in the final circuit that the internal pull-down resistor on the reset pin of the micro-controller was sometimes inadequate to release the reset state of the micro-controller after configuration. An additional 4.7 kΩ pull-down resistor was added.

The new modifications enabled user programs and interrupt subroutines to be written in C, with very few pre-requisites. A linker directive, $ CODE(0030H), can be included when changing a relocatable object file to an absolute file. This directive places the start of the main program at address 0030h. The standard "ACE_DUMMY" module, included automatically by the Linker, will supply the "Jump Vector" to the beginning of the main program, after clearing the data fields and redefining the stack to just above the user variables. The exact pre-requisites for writing a user program for successful testing and debugging on the board, are included in the next section.

---

# 13. Guidelines on the Writing and Assembling of User Programs

The code memory area allocated for user programs is 20K bytes, ranging from address 0000H to 4FFFH. All the interrupt vectors are available for use. There are only minor limitations on the address usage of user interrupt subroutines, and the *same-* and *higher priority* blanking conventions. This will be discussed detail. The entire Internal Data space, all the Special Function Registers, and the first 32K bytes of External Data, are available to the user.

To write a program in **Assembler**, the user must include an "ORG 0030H" directive at the beginning of the main program. A suitable jump instruction to this location must be included at code address 0000h. This is the normal practice when interrupt subroutines are used, but it must also be included *even when no interrupts are used!* The user must also make sure that the "Interrupt 0 Subroutine" is limited to addresses 0003h to 0007h, and the others between 000Bh and 002Fh. For longer interrupt subroutines, "Long Jump" instructions to the appropriate locations of the rest of the subroutines must be used in the interrupt vector space. Any interrupt subroutine must be concluded with the standard "Return from Interrupt" instruction.

If any of the existing subroutines of the Operating System are used, the starting addresses must be defined by the CODE directive. A list of some useful existing subroutines and their starting addresses is included in the appendix.

To access the Keyboard and Display Unit directly, the appropriate external access addresses must be defined by the XDATA directive. Notes on the operation and commands of the Display Module have also been included in the appendix.

Relocatable assembler programs, or programs written in C or another high level languages, will normally be placed to start at 0000H by the Linker program, unless instructed to do otherwise. User interrupt subroutines will push the main program upwards. If this is not sufficient to move the main program past the 002Fh limit, the linker directive, $ CODE(0030H), must be included. This will encourage the linker to place the main program at address 0030H. If the problem still exists, the linker may be forced to start the program at address 0030H, by including a small assembler module to the other object files when they are fed through the linker program. This module may consist of an "ORG 0000H" directive and a data storage statement, "DS 30H", only.

The command line instruction to run the Linker program, could then be in one of the formats shown below as examples:

**C:\ASM>L51 main.OBJ,*next*.OBJ,*another*.OBJ** [Assembler programs]

**C:\ASM>L51 main.OBJ $ CODE(0030h)** [C programs]

**C:\ASM>L51 main.OBJ,*space*.OBJ** [C program + 30 byte space]

All the interrupts are available to the user. Special procedures have been included in the Operating System to make them *transparent*, although the Operating System uses its own interrupt subroutines extensively. To summarise, the user must keep the following in mind when writing and using interrupt subroutines:

* If a user program clears the "Enable All" (EA) or the "External Interrupt Enable" (EX0) flags, the *Single Step*, or *Break Point* operations, or causing a break with the *Escape key*, will be disabled until these flags are set again. Breaks will not occur after any instruction that accesses one of the Interrupt Control registers or the "Return from Interrupt" instruction.

* A Single Step, Break Point or 'Escape' break out of a user program, will corrupt nine additional bytes of the *user* stack above the current stack pointer setting.

* For Single Step and Break Point operations, the program will jump into the Operating System with a "Return from Interrupt" instruction that will remove the blanking effect of the interrupt. The Operating System can then make use of its own interrupts. When single-stepping through a user interrupt subroutine, the normal blanking effect of lower priorities will *not* take place. Any enabled interrupt will be serviced as soon as it is requested. This implies that all the registers and data that will be corrupted by an interrupt subroutine, must first be stored in the stack before the routine is executed. It must again be restored before returning to the main program.

* The Operating System sets the "External Interrupt 0" to be ***edge*** triggered, so that when it simulates an interrupt call, the interrupt request flag can be set by software. If set to *level* triggering, the flag can only be changed by changing the pin level. This could create an unwanted interrupt request repeat of the same subroutine. The Interrupt 0 level should therefore stay **edge triggered** and **high priority!**

* The user "Interrupt 0 Subroutine" must either be entirely contained between addresses 0003H and 0007H, or it must use a "Long Jump" to jump out of this area for longer routines. In a high level language, this may be encouraged by a function call in the interrupt routine. The function must then be defined somewhere else. The normal "Return from Interrupt" after the function call, will still fit into the available eight bytes reserved of the interrupt vector.

* In any other interrupts used by the user system, the interrupt subroutines must fit into the area 000Bh to 002Fh. It may also contain "Long Jump" instructions out of this address area. Higher level compilers for the 8051 do have special instructions to identify and place interrupt subroutines. The linker will normally place a jump instruction at the conventional reserved address location for the interrupt vector, and the rest of the subroutine somewhere else where it can find the space. Again, the use of a function call will encourage the use of a "Long Call" instruction at the vector address, and the function will be placed higher.

Interrupt subroutines are not normally called by assembler program instructions, and the linker may give a warning that these modules are never used. This is not a terminating error and the modules will still be included in the final program.

The Operating System always sets the Enable All, the External Interrupt 0 Enable and the Edge Trigger flags of Interrupt 0, before a user program starts to run. It is recommended that the user would include these instructions in his program as well for the sake of testing the program on the Simulator successfully.

## 13.1. Performance Testing a User Program

As an example for testing a user program and user interrupt subroutines in the various running modes, a special test program was designed. The listing of this program is shown below. Some of the existing subroutines in the Operating System will also be called from this program.

The program will use the Port 1 pins as monitors to test if user interrupt subroutines were executed. The results can either be inspected by the Operating System in Mode 1, at the SFR value 90h, or by adding LEDs to the Port 1 pins as hardware monitors. The two Timers will be preset to give roll-over interrupts. The keyboard and an external user interrupt on pin IRQ can be tested as well. Initially, the *Single Step* mode will be tested for executing the user interrupts. Then the *Break Point* mode, for the incrementing of the DPTR in a loop will be tested. After that the facility to store the settings of a program at a specific point will be tested. Finally, the *Execute* mode will test the use of the existing Operating System subroutines, and the user External Interrupt 0 subroutine.

```
MCS-51 MACRO ASSEMBLER    USERPROG

DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN USERPROG.OBJ
ASSEMBLER INVOKED BY:  C:\ASM\ASM51.EXE USERPROG.SRC

LOC  OBJ      LINE  SOURCE

                 1  $ NOMR NOPI
                 2
                 3  ;         User Test Program
                 4
  F300           5     KyBrd  XDATA    0F300H  ;Keyboard and Switches
  F700           6     DSPCM  XDATA    0F700H  ;Display Module Command
  F701           7     DSPDA  XDATA    0F701H  ;Display Module Data
  F200           8     CNTRG  XDATA    0F200H  ;LCA Control Register
  620C           9     BNASC  CODE      620CH  ;Binary to ASCII sub
  607F          10     DSPMG  CODE      607FH  ;Display Message sub
  60CB          11     READK  CODE      60CBH  ;Read Keyboard sub
  604A          12     W1SEC  CODE      604AH  ;Wait 1 second sub
  602A          13     WRDSP  CODE      602AH  ;Display 1 character sub
  5D86          14     MODF1  CODE      5D86H  ;Auto-off Exit
  0087          15     PCON   DATA      0087H  ;Power control register
```

```
                16
0000            17              ORG 0000H
                18
0000 020030     19              JMP START               ;Jmp to Main at 0030h
                20
0003            21              ORG 0003H               ;User Int. 0 Vector
                22
0003 020098     23              JMP UsrIO               ;Jump to User Int sub
                24
0008            25              ORG 0008H               ;Timer 0 Int. Sub-Rtn.
                26
0008 B292       27              CPL P1.2                ;T0   (P1 = 11111011)
000A 32         28              RETI
                29
0013            30              ORG 0013H               ;User Int. 1 Sub-Rtn.
                31
0013 B293       32              CPL P1.3                ;Int1 (P1 = 11110111)
0015 32         33              RETI
                34
001B            35              ORG 001BH               ;Timer 0 Int. Sub-Rtn.
                36
001B B294       37              CPL P1.4                ;T1   (P1 = 11101111)
001D 32         38              RETI
                39
                40  ;           Main User Test Program
                41
0030            42              ORG 0030H
                43
0030 7590FF     44  START:      MOV P1,#0FFH            ;Set Prrt 1 = 11111111
0033 758911     45              MOV TMOD,#11H           ;Timers = Md 1 (16 bt)
0036 758CFF     46              MOV TH0,#0FFH           ;Preset Timer values:
0039 758AF0     47              MOV TL0,#0EAH           ; Tm0 = FFEAh
003C 758DFF     48              MOV TH1,#0FFH           ; Tm1 = FFCAh
003F 758BE0     49              MOV TL1,#0CAH
0042 D2AF       50              SETB EA                 ;Enable All
0044 D2A9       51              SETB ET0                ;Enable Timer 0
0046 D2AB       52              SETB ET1                ;Enable Timer 1
0048 D2A8       53              SETB EX0                ;Enable Ext. Int. 0
004A D288       54              SETB IT0                ;Set Ex0 Edge Trigger
004C D2AA       55              SETB EX1                ;Enable Ext. Int. 1
004E D28A       56              SETB IT1                ;Set Ex1 Edge Trigger
0050 90FFF8     57              MOV DPTR,#0FFF8H        ;Preset DPTR
0053 438850     58              ORL TCON,#50H           ;Run Timers
                59
0056 A3         60  COUNT:      INC DPTR                ;Loop for SS & Break
0057 E583       61              MOV A,DPH               ;     Point testing
0059 A2E5       62              MOV C,ACC.5
005B 9297       63              MOV P1.7,C              ;P1.7 = DPH.0 (Monitoring)
005D 4582       64              ORL A,DPL               ;Setup for Test DPTR=0000
005F 70F5       65              JNZ COUNT               ;Repeat to Test Tm0,
                                                        ;     Tm1 & Ex1 Ints.
0061 5388AF     66              ANL TCON,#10101111B     ;Stop Timers
                67
                68  ;           Use Opsys Subroutines and Test Ex0 Interrupt
                69
0064 900088     70  MOREKY:     MOV DPTR,#Msg01
0067 12607F     71              CALL DspMg              ;Display "Press any key"
                72
006A 90F700     73              MOV DPTR,#DspCm
006D 74C7       74              MOV A,#0C7H             ;Display Cursor
006F 12602A     75              CALL WrDsp
0072 1260CB     76              CALL ReadK              ;Read Keyboard Sub
                77
0075 12620C     78              CALL BnAsc              ;Convert Key value
0078 309002     79              JNB P1.0,WRCHR          ;Jump if Keyboard Int.
007B 7458       80              MOV A,#'X'              ;'X' for User Ext Int.
007D 90F701     81  WRCHR:     MOV DPTR,#DspDa
0080 12602A     82              CALL WrDsp              ;Display Key or 'X'
0083 12604A     83              CALL W1Sec              ;Wait 1 second
0086 800C       84              JMP MOREKY              ;Repeat the loop
                85
```

```
                    86  ;*************************************************************
                    87          ;Data Storage
                    88  ;*************************************************************
                    89
0088 50726573       90  Msg01:   DB 'Press any Key:  '
008C 7320616E
0090 79204B65
0094 793A2020
                    91
                    92  ;*************************************************************
                    93  ;       User Interrupt 0
                    94  ;*************************************************************
                    95
0098 90F200         96  UsrI0:   MOV DPTR,#CntRg      ;Rd & Wr Control Register
009B E0             97          MOVX A,@DPTR
009C F0             98          MOVX @DPTR,A         ; to Reset IRQ Latch
009D 308202         99          JNB P3.2,RSTKY       ;Jump if Ex0 Pin = 0
00A0 C291          100          CLR P1.1             ;IRQ  (P1 = 11111101)
                   101
00A2 90F300        102  RSTKY:   MOV DPTR,#KyBrd
00A5 E0            103          MOVX A,@DPTR         ;Rd Key & Rst Key Latch
00A6 308202        104          JNB P3.2,NONE
00A9 C290          105          CLR P1.0             ;Keyboard (P1 = 11111110)
                   106
00AB 32            107  NONE:    RETI
                   108
                   109
                   110          END
```

SYMBOL TABLE LISTING
------ ----- -------

| NAME | TYPE | VALUE | ATTRIBUTES |
|---|---|---|---|
| ACC. . . . | D ADDR | 00E0H | A |
| BNASC. . . | C ADDR | 620CH | A |
| CNTRG. . . | X ADDR | F200H | A |
| COUNT. . . | C ADDR | 0056H | A |
| DPH. . . . | D ADDR | 0083H | A |
| DPL. . . . | D ADDR | 0082H | A |
| DSPCM. . . | X ADDR | F700H | A |
| DSPDA. . . | X ADDR | F701H | A |
| DSPMG. . . | C ADDR | 607FH | A |
| EA . . . . | B ADDR | 00A8H.7 | A |
| ET0. . . . | B ADDR | 00A8H.1 | A |
| ET1. . . . | B ADDR | 00A8H.3 | A |
| EX0. . . . | B ADDR | 00A8H.0 | A |
| EX1. . . . | B ADDR | 00A8H.2 | A |
| IT0. . . . | B ADDR | 0088H.0 | A |
| IT1. . . . | B ADDR | 0088H.2 | A |
| KYBRD. . . | X ADDR | F300H | A |
| MODF1. . . | C ADDR | 5D86H | A |
| MOREKY . . | C ADDR | 0064H | A |
| MSG01. . . | C ADDR | 0088H | A |
| NONE . . . | C ADDR | 00A8H | A |
| P1 . . . . | D ADDR | 0090H | A |
| P3 . . . . | D ADDR | 00B0H | A |
| PCON . . . | D ADDR | 0087H | A |
| READK. . . | C ADDR | 60CBH | A |
| RSTKY. . . | C ADDR | 00A2H | A |
| START. . . | C ADDR | 0030H | A |
| TCON . . . | D ADDR | 0088H | A |
| TH0. . . . | D ADDR | 008CH | A |
| TH1. . . . | D ADDR | 008DH | A |
| TL0. . . . | D ADDR | 008AH | A |
| TL1. . . . | D ADDR | 0088H | A |
| TMOD . . . | D ADDR | 0089H | A |
| USRI0. . . | C ADDR | 0098H | A |
| W1SEC. . . | C ADDR | 604AH | A |
| WRCHR. . . | C ADDR | 007DH | A |
| WRDSP. . . | C ADDR | 602AH | A |

## 13.2. Test Results

The test program was assembled and loaded into the user code area by the RS232 serial link. The menu option to start running the user program in the Single Step mode, was selected. The Running Mode Switch 1 was switched ON and the default starting address of 0000h was kept.

A brief second after the 'Enter' key was pressed, the message "Exec. User Prog" was displayed, and almost immediately replaced by displaying the first of the Pre-Interrupt Registers, the Program Counter. It reflected the address of the next instruction that would be executed, **0030h**. The first instruction was the "Jump to address 0030h" instruction. It was therefore executed successfully, and a Single Step break returned the control to the Operating System. From within the Operating System in Mode 2, it was possible to use any of the other Operating System features, such as the inspection of memory segments, etc.

When the 'Exe' key was pressed, the next instruction "To set Port 1", was executed and the address of the following instruction, **0033h**, was displayed. This instruction at address **0033h** will set the TCON register. The TCON register was therefore inspected at the *current* state, and again *after* the instruction had been executed. It confirmed that the correct operation had taken place.

In this fashion, the rest of the setup instructions were executed one at a time, and the results examined immediately in the Pre-Interrupt Registers. They reflected the state of the internal registers just before a break occurred. These register values may not be same values as one would found when using the Edit SFR and Data

options of Mode 1. Mode 1 will display the *current* values that is used by the Operating System.

Single Step breaks will not happen after any instructions that access the interrupt control registers or a "Return from Interrupt" instruction. As expected, no breaks occurred for these instructions between addresses **0044h** and **004Ah**. The break only occurred after completing the **SETB IT0** instruction at address **004Ch**.

The Pre-Interrupt values of Timers 0 and 1 just before the "Run Timers" instruction of address **0053h**, reflected the preset values of **FFEAh** and **FFBAh**. After executing this instruction, the Timer values were **FFF2h** and **FFC2h**, effectively eight counts later. These counts were lost in the Single Step break interrupt subroutine, before the current Timer values were stored. After the next instruction, the Timer values were **FFFFh** and **FFEFh**, effectively fourteen counts later. The additional counts were lost while the Timer values were reloaded and the execution of the next user instruction enabled.

The Timer 0 value at **FFFFh** was just about ready to create a roll-over interrupt. After the next instruction was executed, the program counter was at address **000Bh**, which is the first instruction of the user interrupt subroutine for Timer 0. The Port 1 pin 2 was cleared and examined at address 90h of the SFRs. For this inspection, one has to 'Esc' out of the Pre-Interrupt Mode and select the "Edit SFR" of Mode 1. Displaying the Port 1 value in the *Binary Mode*, confirmed that the pin 2 was cleared. To execute the next instruction, one had to 'Esc' back to the main menu before the 'Exe' key could be pressed.

After the Timer 0 interrupt was completed, one more instruction of the main program was executed before Timer 1 generated a roll-over interrupt. The Program Counter reflected address **001Bh**, which is the first address of the interrupt subroutine of Timer 1.

The next few instructions of the main program increments the DPTR and test of 0000h have been reached. The program looped between instruction addresses 0056h and 005Fh, with only the DPTR incremented by one for every loop. In this section, the breakpoints and the current system variable storage facility were tested. The main program was *single stepped* to address 005Dh. In Mode 7, a break point was defined at address 005Dh. The Running Mode Selection Switches were switched to select the Break Point Mode (Sw 1 = OFF, Sw 2 = ON). The 'Exe' key was pressed to execute the next instructions until the address 005Dh was encountered. The DPTR was inspected on every break to confirm an increment of one count for every loop.

When the DPTR reached a value of FFFCh, the storage facility of Mode 3 was tested by bulk storing the current values of the user program. After another two loops, when the DPTR was at FFFEh, the previous stored values were retrieved by Mode 4. When the pre-interrupt values were inspected, the value of the DPTR reflected the previous stored value of FFFCh.

Finally, the *Execute* running mode and the use of Operating System routines were tested by setting both the switches ON. After the 'Exe' key was pressed, the expected message "Press any Key: " was displayed, as specified in the user

program. The pressing of any hexadecimal key displayed the value on the screen for one second. Then it was cleared to wait for another key. When the 'Esc' key was pressed, control was returned to the Operating System, showing the program counter at 60FFh. This is somewhere in the keyboard read subroutine of the Operating System. To prove that the user Interrupt 0 subroutine was used, the register of Port 1 was inspected to confirm that "pin 0" was cleared.

---

# 14. Conclusions and Recommendations

Although one of the main objectives of the project, "to make it easier to understand the internal operation and architecture for a first time student", was only partially fulfilled, the acquiring of an in depth knowledge of various aspects the micro-controller operation and application made the execution of this project very worthwhile. For a first time micro-controller student, the complexity of a micro-controller system, together with the intricate program development process, were more than enough to occupy the student's concentration completely. Sufficient time must be given to get acquainted and "used to" all the new concepts. To add the complexity of this system to all the rest, was just too much to expect. Most of the first time micro-controller students left the board alone until much later in the course. The more advanced students, who only had to adapt a little further to understand the possibilities and features of the board, found it quite helpful and easy to use.

In view of the above findings, it is recommended that: "The board should only be made available to students in a more advanced micro-controller course". It would still provide an immediate channel to realise and test various user applications, without having to go through the time-consuming process of designing and building his own system. The adaptability of the system will allow him to complete various projects during the course, varying from very simple to more complex applications. A big advantage is that those simple applications, which does not really deserve the trouble of making a special board for testing, can easily be put together on a blank

IBM prototype board and tested thoroughly. The existence of the basic minimum system, together with the ease of programming, testing and modifying the software, require only the additional user specific circuits to be added to the system.

Additional benefits that came from the execution of this project are the in depth knowledge acquired in various aspects of the operation of the micro-controller, and also of the hardware and software design. A great deal was also learned about the approach, planning and execution of a complex project in general.

For the micro-controller specifically, the use of the interrupts had to be studied and tested in detail, especially the interrupt trigger levels, priorities and the exact behaviour of instruction execution sequences. The Single Step, Break Point and 'Escape' break features were entirely based on the unique reaction of the micro-controller on an interrupt request.

The operation and application of the Field Programmable Logic Array, and the entire circuit development process from an ORCAD-based design entry schematic drawing had to be studied and applied. This included the conversion of the schematic drawing to a netlist, the placing and the routing of the final LCA, the selection of the best layout, and the conversion of the configuration into a bitstream. Various techniques had to be tested and "re-invented" to convince and enable the *Xilinx* software to map all the required hardware into the small 2064 package and place specific I/O pins at specific locations to suit the PC Board layout. Where the placing and routing of an LCA normally take a PC about

twenty minutes to complete, the final process for this layout took 36 hours! The configuration modes of the LCA had to be assessed and the most suitable selected for the seeding process and the normal operation. The exact behaviour of the special configuration control pins had to be studied to design the additional hardware to control the configuration processes to do precisely what they were is expected to do, with the minimum of external components.

Valuable experience in the use of the "Borland-C++" (for PCs) was gained in the compilation of the SEED.EXE program. This includes the reading of the configuration and Operating System data files from disk, and the conversion of the data to synchronous serial data trains for transmission through the parallel printer port of the PC.

A set of 25 production Printed Circuit Boards were made professionally, which gave valuable experience in using the computer package "TANGO" to do the preparation of CAM (Computer Aided Manufacturing) files according to the production house specifications. These include Gerber Plot and Number Control Drill (NC Drill) files. Although the final fifteen boards built was not a large production run, it still gave valuable insight into possible variations and modifications to ease production that should be considered and build into a board right from the initial stages. Repeatability, ease of construction and faultfinding, and also the maintenance of the final product had to be kept in mind.

The package "ACAD" was also used to draw the hardware schematic diagram (external to the LCA), and an unconventional method of schematic diagram

drawing has been developed. This method specifically caters for the schematic drawing of micro-controller type circuits. They usually have a multitude of parallel running busses and connections. All the integrated circuit blocks were drawn as long narrow rectangular boxes next to one another. Each connection, or *net*, between the various ICs, was then drawn as a separate horizontal line through all the blocks. Connections made to a specific pin of a block were shown as semicircles, placed on the crossing between the lines and the block border. The correct pin numbers were also placed next to the semi-circles. The next net was placed below the first, so that each horizontal line represents one net. This circuit diagram can then be used with great ease to directly draw up a netlist for the PC board layout by hand.

In this type of layout, it is also much easier to follow the connection path through to the various ICs than in conventional schematic drawings. Connecting lines on these drawings have numerous 90° bends and cross-overs to other connections. Again by using the "ACAD" drawing package, modifications and the shifting the various connections around for the clearest and most logic drawing layout, were made very easy. The final schematic drawing of the external hardware, using this technique, is included in the appendix.

The attempt to reduce the total current dissipation of the prototype board proved to be very valuable exercise and had a profound influence on the design and approach to any other projects since: "To aim for the minimum possible current dissipation without any loss of performance, right from the initial planning and design stages of a project!"

Similar innovations were made on the software side:

The development of the Broad Based or Operating Diagrams and the Single Line Flow Diagrams from the conventional flow diagram standards, were so useful in clarifying the flow of a program, that it has since been used in quite a few consequent projects. It was also imported into a few formal courses in the programming of micro-controllers. By using the computer package "ACAD", all the flow diagram design work was done on the computer screen, which made changes and modifications easy and allowed neat printouts to be made to base the rest of the development on.

Due to the stack limitations set by the objectives, careful track had to be kept on the stack depth when using subroutines within other subroutines. It was also important to know what preset values were required when entering a subroutine and what outputs would result when returning. Equally important were which registers and variables would be corrupted when a specific subroutine was executed. These requirements forced the development of the *good* habit: "To include a summary at the beginning of each subroutine, stating the name, a short description of what is done, the input requirements, and also what output will result and where it could be found. It also states which registers will receive new data that would corrupt old values, which other subroutines would be called from within the subroutine, and the stack depth used". Some subroutines even have various possible stack depths, depending on which options were chosen from within the subroutine.

To be able to predict the placement of Assembler and "Franklin C-Compiler" modules by the L51 Linker program, a vast number of test modules were written in both source languages. These modules were assembled and compiled, and tested in various combinations with various placement directives, to find which gave the best and most consistent results. The suggestions for writing user programs were all based on these results. At the same time, a fair amount of knowledge was accumulated about the use and general compilation process of a C-compiler and the L51 Linker program.

In general, the use of all the various existing computer packages for various aspects of the project, was in itself a valuable experience. A brief summary of all the packages used and their various applications, follows:

* **PCTYPE.EXE:** A word processing package, used for writing the source files for the Operating System, the Seeding Driver, and various test files. Further, it was used for the inspection of computer generated error files, listing files and Hex files, and the preparation of symbol files for the simulator program. Finally, it was used to compile the draft copy of the documentation.

* **W51.EXE:** The WordPerfect word processing package for the preparation of the final documentation.

* **ASM51.EXE:** The Assembler program, used for assembling the Operating System and the various assembler test files.

* **AVSIM.EXE:** The 8051 Simulator, used to test and debug the logic and flow of the Operating System and test program modules.

* **L51.EXE:** The Relocator-Linker program, used to convert and link together relocatable object files of assembler modules and modules written in C into absolute object files.

* **C51.EXE:** The Franklin C-Compiler for Micro-Controllers, for compiling and investigating the placement of main programs and interrupt subroutines, written in C.

* **BC.EXE:** Borland C++ Compiler for writing and compiling the seeding driver program, SEED.EXE.

* **DRAFT.EXE:** The ORCAD schematic capture package for drawing the LCA design entry circuits for the general decoding and support of the micro-controller, as well as the Seeding Shift Register and the Address Counter.

* **PLOTALL.EXE:** The ORCAD plot driver program to convert the schematic drawings of ORCAD to DXF files, for loading into the drawing package ACAD to generate neat and proper printouts.

* ***Xilinx* Package:** A set of approximately 10 programs used to test, convert, map, place, route and check the internal configuration of the LCA, and to produce the final bitstream Hex files, ready for loading into the LCA.

* **PCB.EXE:** The TANGO PCB layout for the design and testing of the netlist integrity and Design Rule Check (minimum spacing), of the external circuit diagram of the board, as well as to generate Gerber Photoplot and NC Drill files for production.

* **ACAD.EXE:** A general purpose drawing package, for the drawing of the Operating and Flow Diagrams of the Operating System, the drawing of the external schematic diagram of the board, the printout of the ORCAD schematic diagrams, and the preparation of the sketches and drawings for the documentation. The drawings were converted and stored in the DXF format for transfer to the WordPerfect package.

* **GRAPHCNV.EXE:** Conversion program for a DXF format drawing to the WPG format, required by the WordPerfect package.

* **ASEASY.EXE:** A spreadsheet package, for the simulation and testing of the decoding circuits of the LCA and the generation and testing of the final PCB netlist, with the aid of macros.

* **HG.EXE:** Harvard Graphics for the generation of the sketches and drawing of the notes for the micro-controller course.

Without the aid of the computer and applicable CAD packages, an engineering project like this would probably be very difficult or impossible to execute effectively!

---

# 15. Future Modifications and Extensions

Throughout the building, testing and application of the final production board, a few items were noted which should be modified in future circuits. They are as follows:

The auto-off routine and hardware circuits worked well, as long as the battery voltage was sufficient and no undue short circuits were placed on the supply. This can easily happen when an IBM prototype card or other circuit is plugged into the board. It sometimes happens that the configuration data in the Code RAM gets corrupted and causes the LCA to not configure properly. It will then stay in the configuration mode. If the seeding cable and a PC are handy, it will be simple to re-seed the board and continue. If not, however, there is not an easy way to switch the power to the board off, except to remove and disconnect the battery. It is recommended to place two connections on the board that are accessible through the side openings. One can then, for example, use a coin to short out two pins to switch the power off manually.

If the View Angle Adjust potentiometer is moved slightly away on the PCB layout from under the display, it would be possible to place the potentiometer on the Display Module side of the board. The adjustment shaft can then be extended out through the face plate. This will make the adjustment of the view angle much easier.

The jumper and track cutting modifications to the address lines **A12** and **A14** should be build into the PCB layout of the board. It may require a new pin

allocation of the general I/O pins of the LCA, and another "place and route" run for the LCA. The additional micro-controller Reset pull-down resistor should also be incorporated into the PCB layout.

If it is possible, another circuit configuration for the software controlled *power off* circuits should be designed. The supply break should not be made in the line between the circuit ground and the battery or the external supply ground. They should be connected permanently. A common ground line would be a great advantage for experimental circuits that require more current than the onboard regulator can supply, and that uses the external supply directly though its own regulating circuits.

The spacing of the jumper connections should be such that adjacent jumpers can be mounted using a single jumper strip, with only the pin between the two jumpers removed or cut off. To mount and line up one longer strip only, is much easier than two short strips next to each other.

The seeding and RS232 serial download cables are never used together. They could be made into a single cable with two different sets of plugs and sockets at the ends.

An attempt was made to use the board as an in-circuit emulator for another micro-controller system. This was done by extending the pins of the micro-controller though the edge connector and a multi-core cable to a 40-pin IC plug. The 40-pin plug was plugged into the IC socket of the micro-controller of the other board to emulate its micro-controller and take over the control of the hardware. The

emulator system will then have the advantage of direct downloading of modified programs from the PC, and being able to Single Step or use Breakpoints to test and debug the hardware and software of the other board. The PSEN and oscillator connections to the user board were disabled and the Interrupt 0 pin from the user board was re-routed to the IRQ pin of the system through the multi-core cable.

These emulator experiments were only partially successful on the specific system tested. It was possible to access and test the memory and I/O circuits of the user board without any problems. The *Single Step* and *Execute* running modes did not work very successfully. It could have been more the result of the complexity of the tested system itself. The system was a burglar alarm that would be battery powered if a power mains break was detected. Special emphasis was therefore placed on power conservation, and the system was placed in the Power Down or Idle modes for most of the time. It was only "woken up" by a hardware reset or interrupt request when required. The routing of the critical Reset and Interrupt 0 request lines through the LCA was probably the cause of the problems.

Further tests to use the board as an in-circuit emulator on simpler application circuits may prove more successful.

---

# 16. Other Project Developments as a Direct Result of this Project

As the direct result of the background knowledge acquired in the execution of this project in the basic requirements of a minimum micro-controller system, some other micro-controller projects were developed. The first two projects described below, led up to two distinct and very successful student training systems: One was a simpler 8051 system, and the other for the PIC16C84 micro-controller.

## 16.1. A Simple EPROM Emulator

The external data RAM and the smart socket of this Training board were used to emulate an EPROM for another system. This was done in an effort to steer clear of the time-consuming process of having to erase and program EPROMs. The user program was loaded into the Code RAM of the Training board through the RS232 link, then "Block Moved" into the External Data RAM. The RAM, together with its *smart socket*, was then removed and simply plugged into the EPROM socket of the user board. This system works very well as an EPROM emulator.

## 16.2. A Simple 8051 Emulator

Soon the plugging and unplugging of the EPROM Emulator above, also became labourious and another solution was sought. If the RAM IC, the micro-controller and an address latch were placed on a separate board and the micro-controller pins extended underneath the board, it can be used as an in-circuit emulator. All that

was still required was to find some easy way to load the RAM directly from the PC, independent of the Training Board. This board can then be plugged into and left in the micro-controller IC socket of the user board. It will act as an in-circuit emulator with an easy re-programmable "piggy-back" RAM for the Code memory.

The final system required one more address latch for the high address byte and an octal latch to load the RAM directly through the parallel printer port of the PC. This system did not require an existing Operating System on the board that would receive and store the data. A logic voltage level converter, such as one would require if the RS232 serial port were used, was not required. Data from the PC will be downloaded directly to the board in a *nibble* format. The *nibble* format consists of four parallel data and four control lines.

The system operated as follows: The high nibble of the high address byte would be loaded into four registers of the 8-bit D-flip-flop with a clock control signal. Then the low nibble would be loaded into the same four flip-flop registers, while the high nibble is parallel shifted to the other four registers. The outputs of the octal D-flip-flop would then contain the whole high address byte. The high address is then stored in the high address latch through one of the control lines. The outputs of the high address latch are directly connected to the high address pins of the RAM. The low address byte would be loaded in a similar two-part operation and latched into the low address latch. The low address latch outputs are connected to the low address lines of the RAM. Finally, the data byte will be loaded into the D-flip-flop, of which the outputs are also connected to the data pins of the RAM. The fourth control line will write this data into the RAM.

During the loading process, the micro-controller is placed in the ONCE or "On Circuit Emulation" Mode, where all activity on the controller is suspended. Once loaded, the cable to the PC can be removed, the high address latch and D-flip-flop disabled, and the micro-controller taken out of the ONCE mode to run the program. The *smart socket* became superfluous because it was not necessary to remove the RAM from the constant supply of the board. Forty DIL pins out on the underside of the board can be extended to plug directly into the micro-controller socket of a user board.

## 16.3. A Simpler 8051 Training Board for First Time Students

Eight LEDs for outputs, and four push-buttons for inputs were added to the above board to make it a very good training tool for first time micro-controller students. The ease of programming and portability of the system, and the low cost of the components placed it within the limited budget of a student. Each can construct his own board to take home to practice at will. A small conversion program converted a standard Hex file into a so-called "Nibble" file, which could be downloaded through the parallel printer port using the standard DOS **COPY** command. No additional plug-in cards or sophisticated programming software was required from the driving PC. That made the programming system portable to any PC with a parallel printer port. The board could initially be used as is, to teach the operation of the micro-controller and simple programming techniques. It was also possible to add external hardware to the board though extension connectors. The extension pins out on the underside of the board, allowed the board to be used as an in-circuit emulation of the micro-controller of another system as well.

As a basic training board for first time micro-controller students, and as an 8051 Emulator, this circuit worked most successfully.

## 16.4. The PIC Development System

As a direct *descendant* of the Simplified 8051 Training Board, a similar system was developed for the PIC family of micro-controllers. The PIC16C84 was chosen as the template for learning about all the other members of the PIC family, because it was the only micro-controller with easy programmable EEPROM code memory. The rest uses EPROM memory only. The code memory is internal to the controller and no external memory or address latches were required. It resulted in a much smaller and simpler board. The mid-range of PIC micro-controllers can be programmed by two serial lines while still plugged into the applications circuit. A small programmer was developed which plugs into the board and is driven from a PC through the parallel printer port. The 12.5 volt programming voltage required, was derived from a converter on the Programmer Board. It draws its supply directly from the 5 volt of the Basic Board. The programmer was kept on a separate board so that it could also be used to program PIC controllers in other circuits.

Again the portability to any PC with a printer port, and the simplicity and ease of re-programming were maintained, which was indeed also one of the main objectives of the original "parent" project.

==========

# 17. Bibliography

Chapman WA. **Mastering C Programming.** 1991. Macmillan Education Ltd., Houndmills, Bastingstoke, Hampsire, RF232XS. ISBN 0-333-49842-9.

Dallas Semiconductor, Inc. **Product Data Book.** 1992-1993. Dallas Semiconductor Corporation, Dallas, Texas, 75244-3292.

Franklin Software, Inc. **C51 Compiler Reference Manual.** Rev. 10.90, 1990. Franklin Software, Inc., San Jose, California, 95129.

General Instrument, Inc. **General Instruments Optoelectronic Products.** 1995. General Instrument Optoelectronic Division, Palo Alto, California, 94304.

Goldstein LJ & Gritz L. **Hands-on Turbo C.** 1989. Brady, Simon & Schuster, Inc. Gulf & Western Building, One Gulf & Western Plaza, New York, 10023. ISBN 0-13-383704-1.

Intel Corporation. **Embedded Microcontrollers & Processors.** Volume I, 1992. Intel Literature Sales, Mt Prospect, Illinois, 60056-7641. ISBN 1-55512-140-3.

Intel Corporation. **Memory Products.** 1992. Intel Literature Sales, Mt Prospect, Illinois, 60056-7641. ISBN 1-55512-117-9.

Intel Corporation, Inc. **MCS-51 Macro Assembler User's Guide for DOS Systems.** 1986. Intel Corporation, Santa Clara, California, 95051.

Kernigham BW & Richie DM. **The C Programming Language.** Second Edition, 1988. Prentice Hall, Englewood Cliffs, New Jersey, 07632. ISBN 0-13-110370-9.

Kochan SG. **Programming in C.** Revised Edition, 1988. Hayden Books, 11711 North College, Suite 141, Carmel, Indiana, 46032, USA. ISBN 0-672-48420-X.

National Semiconductor Corporation. **CMOS Logic Databook.** 1988. National Semiconductor, Santa Clara, California, 95052-8090.

National Semiconductor Corporation. **Linear 1 Databook.** Rev.1, 1988. National Semiconductor, Santa Clara, California, 95052-8090.

National Semiconductor Corporation. **Memory Databook.** Rev.1, 1988. National Semiconductor, Santa Clara, California, 95052-8090.

Optrex Corporation. **Dot Matrix LCD Module User's Manual.** Optrex Corporation, 3-14-9, Yushima, Bunkyo-ku, Tokyo 113, Japan.

Waite M et al. **C Primer Plus.** 1984. Howard W. Sams & Co., Inc., 4300 West 62nd Street, Indianapolis, Indiana, 46268, USA. ISBN 0-672-22090-3.

Xilinx, Inc. **User Guide and Tutorials.** Vol. I & II. 1991. Xilinx, Inc., San Jose, California, 95124.

Xilinx, Inc. **Design Implementation Reference Guide.** 1992. Xilinx, Inc., San Jose, California, 95124.

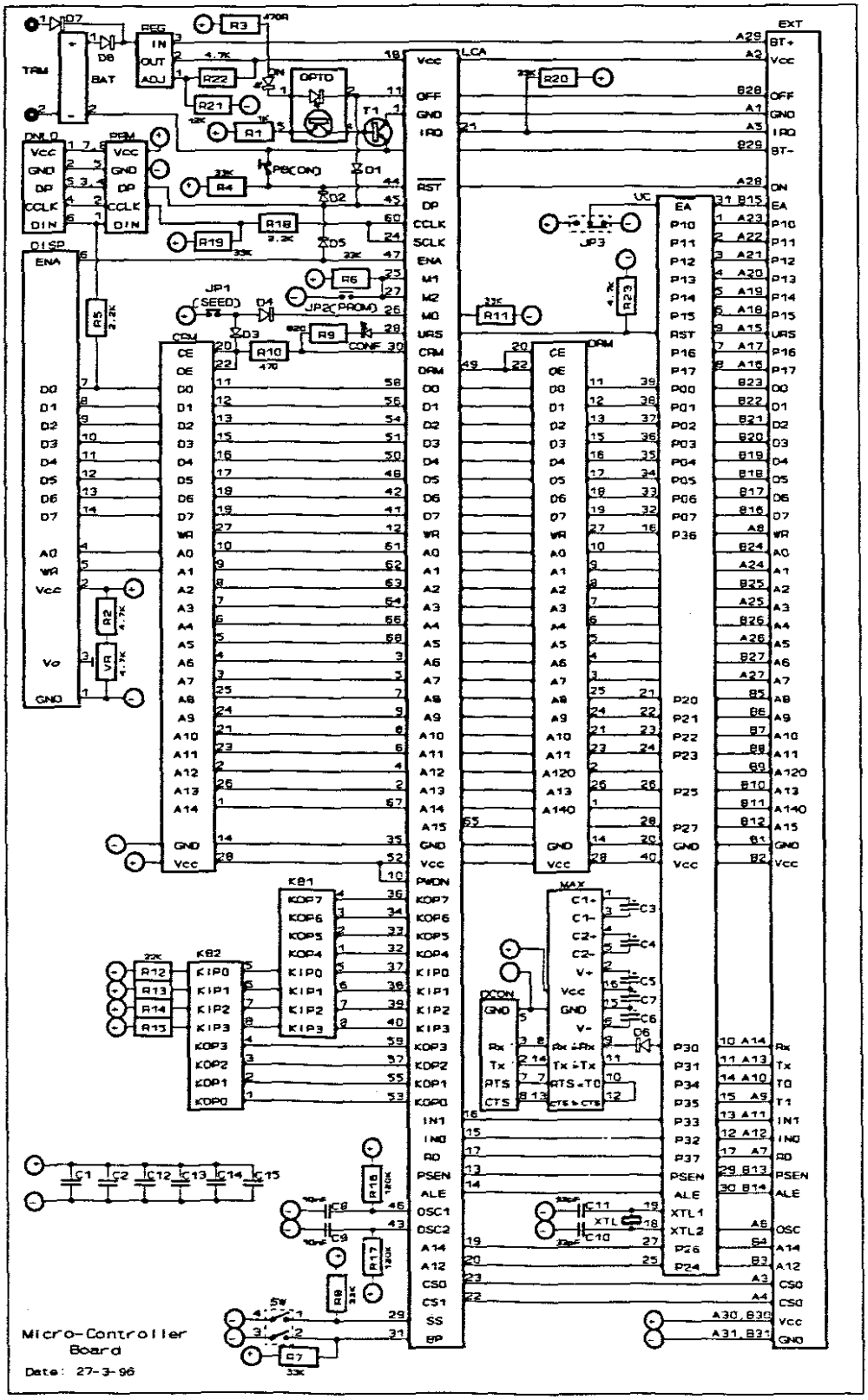
Xilinx, Inc. **ORCAD Schematic Design Interface Reference Guide.** 1991. Xilinx, Inc., San Jose, California, 95124.

Xilinx, Inc. **The Programmable Logic Data Book.** 1993. Xilinx Inc. San Jose, California, 85124.

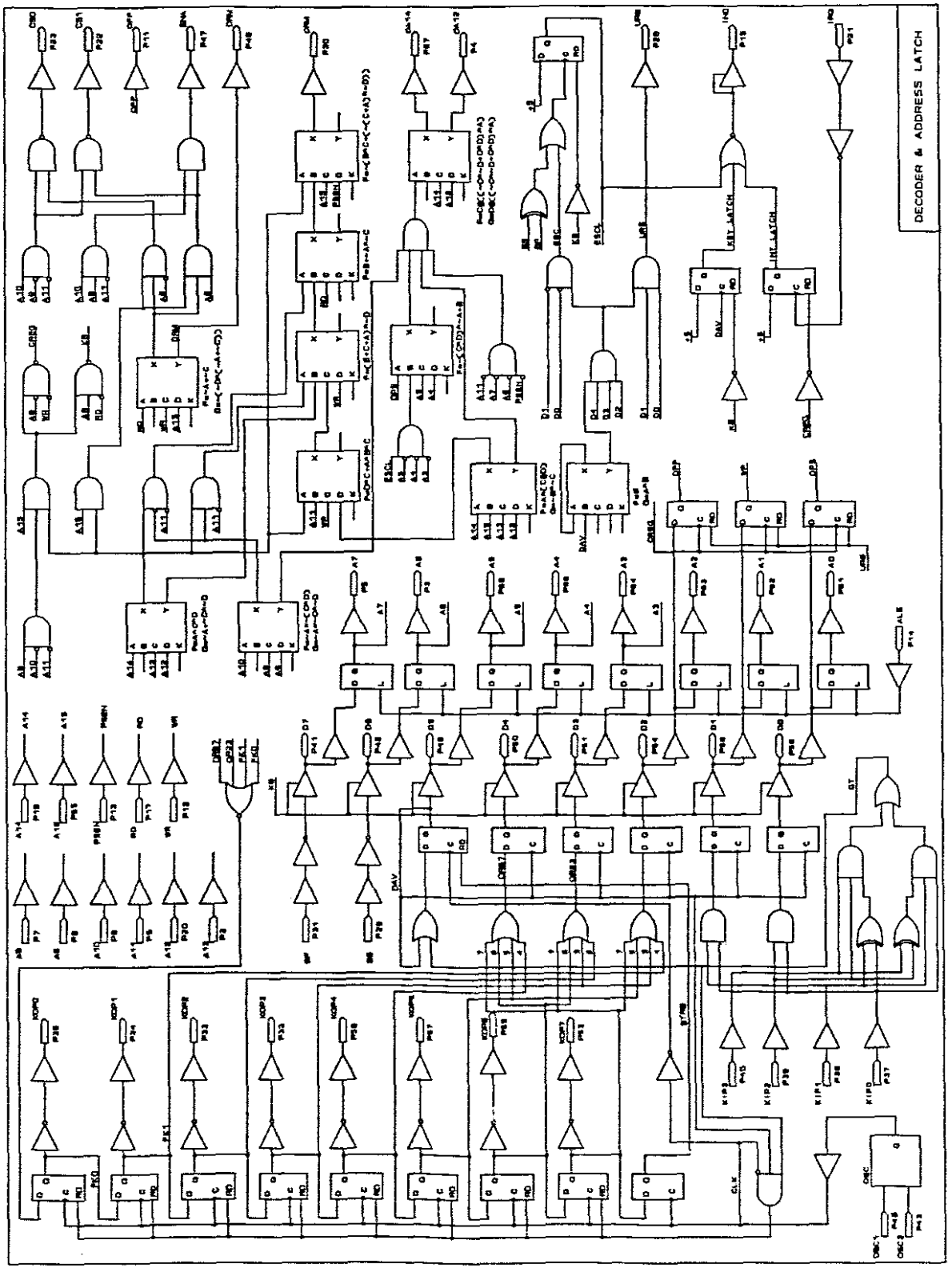
# Table of Contents to the Appendices


| | | *Page* |
|---|---|---|
| A: | The Complete External Board Schematic Diagram | 271 |
| B: | The Complete DECODER Schematic Diagram | 272 |
| C: | The Complete SHIFT REGISTER Schematic Diagram | 273 |
| D: | Numbers and Functions of the Extension Edge Connector Pins | 274 |
| E: | Memory Map Allocations and Bulk Storage Addresses | 227 |
| F: | The Pinouts and Functions of the XC2064 PLCC68 LCA Package | 280 |
| G: | The Seeding Serial Data Download Cable | 283 |
| H: | Listing of the Seeding Driver Program: SEED.EXE | 284 |
| I: | Useful Subroutines Already Existing in the Operating System | 291 |
| | Wait for a Programmable Period Subroutine | 292 |
| | Reset and Clear the LCD Display Module | 292 |
| | Display a 16-Character Message on the Display Module | 293 |
| | Wait for and Decode a Key from the Keyboard | 293 |
| | Select BAUD Rate Subroutine | 294 |
| | Binary Nibble to ASCII Conversion | 294 |
| | ASCII to Binary Conversion | 295 |
| J: | The LCD Display Module and Control Commands | 296 |
| K: | The Binary and Hex Format Files | 305 |
| L: | Component List for the Training Board | 308 |
| M: | Component Overlay and Connections | 312 |
| N: | Micro-Controller Board Netlist | 313 |
| O: | Costing and Suppliers of the Components of the Board | 318 |

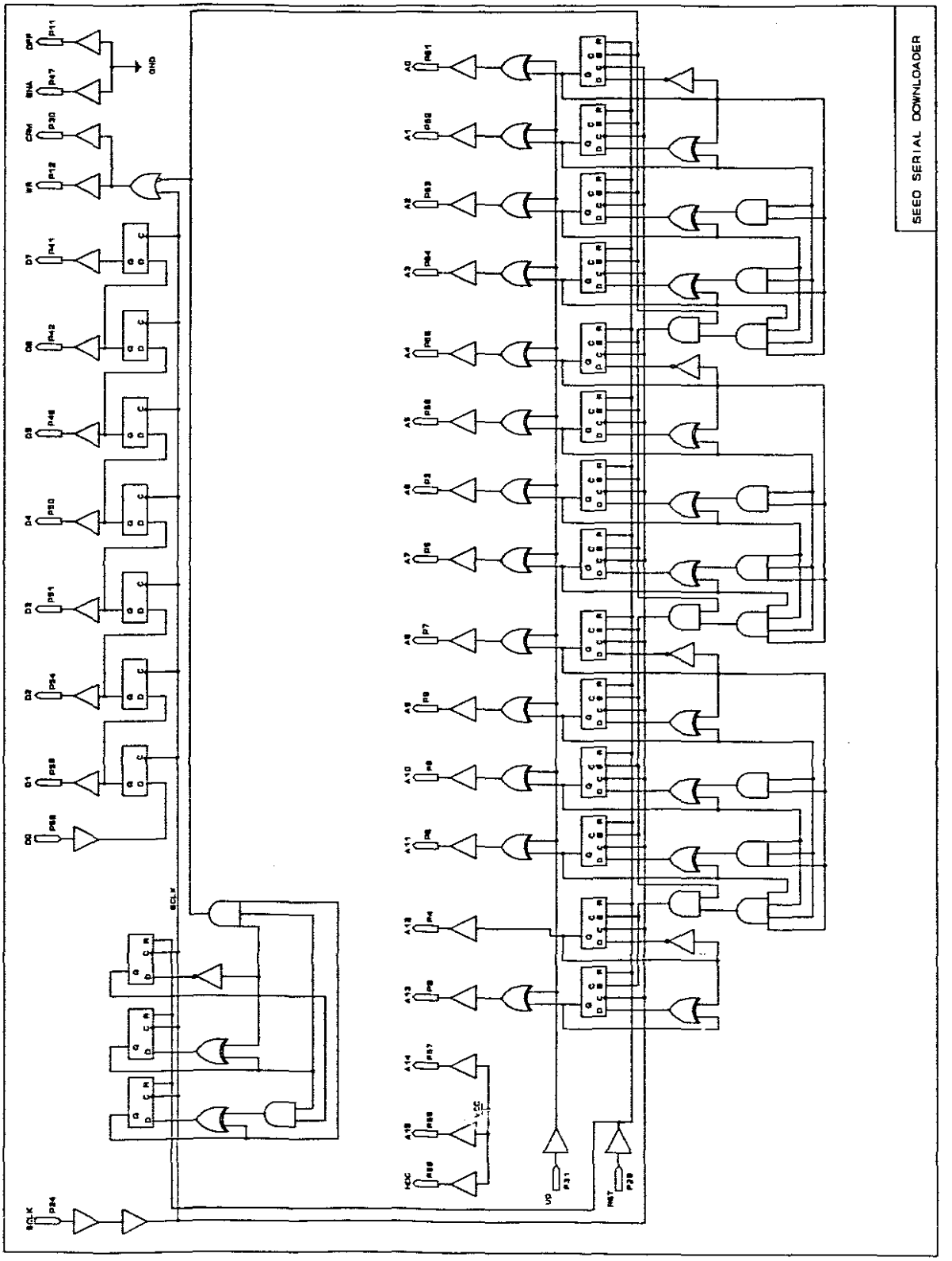
# Appendix A: Complete External Schematic Circuit Diagram



# Appendix B: The Complete DECODER Schematic Diagram



# Appendix C: The Complete SHIFT REGISTER Schematic Diagram



# Appendix D: Numbers and Functions of the Extension Edge Connector Pins

The extension edge connector is the size of a standard IBM motherboard bus extension edge connector, and blank IBM prototype circuit cards can be used to add circuitry to the Training board. The pin numbers run from A1 to A31 from the bottom to the top on the side that faces the user, and B1 to B31 are on the underside. All the pins of the micro-controller are directly connected to the pins of the extension socket, plus a few that may be useful for external circuits.

**Top Side:** (Facing a user)

| Pin | Name | Description |
|---|---|---|
| A1 | GND | Circuit Ground (NB: Not the battery ground!) |
| A2 | Vcc | +5 V circuit positive (NB: Not the battery positive!) |
| A3 | $\overline{CS0}$ | External $\overline{\text{Chip Select 0}}$. Addresses F400 to F4FF. (256 bytes. Further decoding can be done with addresses A0 to A7, RD, and WR) |
| A4 | $\overline{CS1}$ | External $\overline{\text{Chip Select 1}}$. Addresses F500 to F5FF. |
| A5 | $\overline{IRQ}$ | User $\overline{\text{External Interrupt 0}}$, in place of Port 3.2. A high to low transition on this pin will latch an interrupt to P3.2. The latch must be reset by a write instruction to the LCA control register. |
| A6 | OSC | Oscillator output from Pin 19 of the micro-controller |
| A7 | $\overline{RD}$ | $\overline{\text{Read}}$ control from Port 3.7 |

| Pin | Name | Description |
|---|---|---|
| A8 | $\overline{WR}$ | $\overline{Write}$ control from Port 3.6 |
| A9 | T1 | Timer 1 external counter input to Port 3.5 |
| A10 | T0 | Timer 0 external counter input to Port 3.4 |
| A11 | $\overline{IN1}$ | External $\overline{Interrupt\ 1}$ to Port 3.3 |
| A12 | $\overline{IN0}$ | External $\overline{Interrupt\ 0}$ to Port 3.2, used for a key or IRQ interrupt generation. |
| A13 | TX | Serial Transmit from Port 3.1 |
| A14 | RX | Serial Receive to Port 3.0 |
| A15 | URS | Micro-Controller Reset to pin 9 |
| A16 | P17 | To/from Port 1.7 |
| A17 | P16 | To/from Port 1.6 |
| A18 | P15 | To/from Port 1.5 |
| A19 | P14 | To/from Port 1.4 |
| A20 | P13 | To/from Port 1.3 |
| A21 | P12 | To/from Port 1.2 |
| A22 | P11 | To/from Port 1.1 |
| A23 | P10 | To/from Port 1.0 |
| A24 | A1 | Address 1 from address latch in the LCA |
| A25 | A3 | Address 3 from address latch in the LCA |
| A26 | A5 | Address 5 from address latch in the LCA |
| A27 | A7 | Address 7 from address latch in the LCA |
| A28 | ON | External ON control. (Connect to BT- to switch on) |
| A29 | BT+ | Unregulated positive supply to the extension card |

| Pin | Name | Description |
|---|---|---|
| A30 | Vcc | +5V circuit positive |
| A31 | GND | Circuit Ground |

**Under Side:** (Away from the user)

| | | |
|---|---|---|
| B1 | GND | Circuit Ground |
| B2 | Vcc | +5V circuit positive |
| B3 | A12 | Address 12 to the LCA for interrupt vector swapping |
| B4 | A14 | Address 14 to the LCA for interrupt vector swapping |
| B5 | A8 | Address 8 from Port 2.0 |
| B6 | A9 | Address 9 from Port 2.1 |
| B7 | A10 | Address 10 from Port 2.2 |
| B8 | A11 | Address 11 from Port 2.3 |
| B9 | A12 | Address 12 from Port 2.4 to the edge connector only |
| B10 | A13 | Address 13 from Port 2.5 |
| B11 | A14 | Address 14 from Port 2.6 to the edge connector only |
| B12 | A15 | Address 15 from Port 2.7 |
| B13 | $\overline{\text{PSEN}}$ | $\overline{\text{Program Segment Enable}}$ form micro-controller pin 29 |
| B14 | ALE | Address Latch Enable from micro-controller pin 30 |
| B15 | $\overline{\text{EA}}$ | $\overline{\text{External Code Enable}}$ to micro-controller pin 31 |
| B16 | D7 | Address/Data 7 from Port 0.7 |
| B17 | D6 | Address/Data 6 from Port 0.6 |
| B18 | D5 | Address/Data 5 from Port 0.5 |
| B19 | D4 | Address/Data 4 from Port 0.4 |

| Pin | Name | Description |
|---|---|---|
| B20 | D3 | Address/Data 3 from Port 0.3 |
| B21 | D2 | Address/Data 2 from Port 0.2 |
| B22 | D1 | Address/Data 1 from Port 0.1 |
| B23 | D0 | Address/Data 0 from Port 0.0 |
| B24 | A0 | Address 0 from address latch in the LCA |
| B25 | A2 | Address 2 from address latch in the LCA |
| B26 | A4 | Address 4 from address latch in the LCA |
| B27 | A6 | Address 6 from address latch in the LCA |
| B28 | OFF | OFF control from LCA (Goes positive to switch off) |
| B29 | BT- | To Negative Battery Terminal (NB: Not Circuit Ground) |
| B30 | Vcc | +5V circuit positive |
| B31 | GND | Circuit Ground |

==========

# Appendix E: Memory Map Allocations and Bulk Storage Addresses

## The 64K Byte Code Memory Map: (Read Access through PSEN)

| | |
|---|---|
| 0000 - 4FFFh | Available for User Programs (20K bytes) |
| 5000 - 6FFFh | Operating System (8K bytes) |
| 7000 - 7FFFh | Not available as code (4K bytes): System Variables, I/O Mapping and LCA Configuration. |
| 8000 - FFFFh | Code Addresses do not exist. |

## The 64K Byte External Data Memory Map: (Access Using RD & WR)

| | |
|---|---|
| 0000 - 7FFFh | External Data - available to the user. |
| 8000 - CFFFh | User Programs (20K bytes) - mirror image of Code |
| D000 - EFFFh | Operating System (8K bytes) - write protected |
| F000 - F2FFh | System Variables, Internal Data and SFR store |
| F300 - F3FFh | As 1 WRITE byte to the Control Register, CREG, in the LCA, overlaid by a READ/WRITE memory byte to enable the reading of the current settings. |
| F400 - F4FFh | External Chip Select, CS0 (256 bytes) |
| F500 - F5FFh | External Chip Select, CS1 (256 bytes) |
| F600h | Display Module Command Instruction WRITE address |
| F601h | Display Module Data WRITE address |
| F602h | Display Module Status READ address |
| F603h | Display Module Data READ address |
| F604 - F6FFh | Display Module mirror image addresses |

| | |
|---|---|
| F700 - F7FFh | 1 READ byte for the Keyboard and the Running Mode Switches. |
| F800 - FFFFh | LCA Configuration (2K bytes) - write protected |

## System Variable and Bulk Storage Address Locations

| Normal System Variables | | Bulk Storage: |
|---|---|---|
| F000 to F03F | Break Point Table | F080 to F0BF (64 bytes) |
| F040 to F041 | Table End (0000H) | F0C0 to F0C1 (2 bytes) |
| **Pre-Interrupt Storage:** | | **System Variable Storage:** |
| F042 to F043 | Program Counter | F0C2 to F0C3 (2 bytes) |
| F044 | Acc | F0C4 |
| F045 | PSW | F0C5 |
| F046 | DPL | F0C6 |
| F047 | DPH | F0C7 |
| F048 | B | F0C8 |
| F049 | SP | F0C9 |
| F04A | IE | F0CA |
| F04B | IT | F0CB |
| F04C | TMOD | F0CC |
| F04D | TCON | F0CD |
| F04E | TL0 | F0CE |
| F04F | TH0 | F0CF |
| F050 | TL1 | F0D0 |
| F051 | TH1 | F0D1 |

| | | |
|---|---|---|
| F052 to F069 | Internal Data | F0D2 to F0E9 (24 bytes) |
| F06A | Cursor position | F0EA |
| F06B to F07A | Displayed Characters | F0EB to F0FA (16 bytes) |
| F07B to F07F | (Not used) | F0FB to F0FF (5 bytes) |

**Other SFR Registers** (Not stored as Pre-Interrupt values):

| | |
|---|---|
| Port 0 | F100 |
| Port 1 | F101 |
| Port 2 | F102 |
| Port 3 | F103 |
| PCON | F104 |
| SCON | F105 |
| T2CON [80C32] | F106 |
| RCAP2L [80C32] | F107 |
| RCAP2H [80C32] | F108 |
| TL2 [80C32] | F109 |
| TH2 [80C32] | F10A |
| Future use | F10B to F117 (28 bytes) |
| Internal Data | F118 to F1FF (top 231 bytes) |

The current lower 24 bytes are not stored, because they will be corrupted by the Operating System and the stack operations during the Internal data and System Variable storage and retrieval operations of Modes 3 and 4. The contents of these 24 bytes for a User Program are stored together with the Pre-Interrupt register values.

==========

## Appendix F: The Pinouts and Functions of the XC2064 PLCC68 LCA Package

| Pin | Sym. | Normal Function | I/O | Mast. Par. Conf. | Serial Slave Conf. |
|---|---|---|---|---|---|
| 1 | GND | Ground | - | Ground | Ground |
| 2 | A13 | Address 13 | I | A13 (O) | <High> |
| 3 | A6 | Address 6 | O | A6 (O) | <High> |
| 4 | A12O | Address 12 | O | A12 (O) | <High> |
| 5 | A7 | Address 7 | O | A7 (O) | <High> |
| 6 | A11 | Address 11 | I | A11 (O) | <High> |
| 7 | A8 | Address 8 | I | A8 (O) | <High> |
| 8 | A10 | Address 10 | I | A10 (O) | <High> |
| 9 | A9 | Address 9 | I | A9 (O) | <High> |
| 10 | PWDN | Power Down | I | Power Down (I) | Power Down (I) |
| 11 | OFF | Off Control | O | <High> | <High> |
| 12 | WR | Write | I | <High> | <High> |
| 13 | PSEN | Code Read | I | <High> | <High> |
| 14 | ALE | Address Enable | I | <High> | <High> |
| 15 | IN0 | Interrupt 0 | O | <High> | <High> |
| 16 | IN1 | (not used) | - | <High> | <High> |
| 17 | RD | XData Read | I | <High> | <High> |
| 18 | Vcc | +5 volt | - | +5 volt | +5 volt |
| 19 | A14 | Address 14 | I | <High> | <High> |
| 20 | A12 | Address 12 | I | <High> | <High> |
| 21 | IRQ | User Interrupt | I | <High> | <High> |
| 22 | CS1 | Chip Select 1 | O | <High> | <High> |

| | | | | | |
|---|---|---|---|---|---|
| 23 | CS0 | Chip Select 0 | O | <High> | <High> |
| 24 | SCLK | (not used) | I | <High> | <High> (Ser.Clk) |
| 25 | M1 | (not used) | I | M1 = 1 (I) | M1 = 1 (I) |
| 26 | M0 | (not used) | I | M0 = 0 (I) | M0 = 1 (I) |
| 27 | M2 | (not used) | I | M2 = 1 (I) | M2 = 1 (I) |
| 28 | URS | 80C31 Reset | O | HDC (O) | HDC (O) |
| 29 | SS | SS Switch | I | <High> | <High> (Reset) |
| 30 | CRM | Code RAM Ena | O | LDC (O) | LDC (O) |
| 31 | BP | BP Switch | I | <High> | <High> (Up/Down) |
| 32 | KOP4 | Keybrd Out 4 | O | <High> | <High> |
| 33 | KOP5 | Keybrd Out 5 | O | <High> | <High> |
| 34 | KOP6 | Keybrd Out 6 | O | <High> | <High> |
| 35 | GND | Ground | - | Ground | Ground |
| 36 | KOP7 | Keybrd Out 7 | O | <High> | <High> |
| 37 | KIP0 | Keybrd In 0 | I | <High> | <High> |
| 38 | KIP1 | Keybrd In 1 | I | <High> | <High> |
| 39 | KIP2 | Keybrd In 2 | I | <High> | <High> |
| 40 | KIP3 | Keybrd In 3 | I | <High> | <High> |
| 41 | D7 | Data 7 | I/O | D7 (I) | <High> |
| 42 | D6 | Data 6 | I/O | D6 (I) | <High> |
| 43 | OSC2 | Oscillator 2 | I | <High> | <High> |
| 44 | RST | LCA Reset | I | Reset (I) | Reset (I) |
| 45 | PD | Program | I | Done (O) | Done (O) |
| 46 | OSC1 | Oscillator 1 | I | <High> | <High> |

| | | | | | |
|---|---|---|---|---|---|
| 47 | ENA | ENA Display | O | <High> | <High> |
| 48 | D5 | Data 5 | I/O | D5 (I) | <High> |
| 49 | DRM | Data RAM Ena | O | <High> | <High> |
| 50 | D4 | Data 4 | I/O | D4 (I) | <High> |
| 51 | D3 | Data 3 | I/O | D3 (I) | <High> |
| 52 | Vcc | +5 volt | - | +5 volt | +5 volt |
| 53 | KOP0 | Keybrd Out 0 | O | <High> | <High> |
| 54 | D2 | Data 2 | I/O | D2 (I) | <High> |
| 55 | KOP1 | Keybrd Out 1 | O | <High> | <High> |
| 56 | D1 | Data 1 | I/O | D1 (I) | <High> |
| 57 | KOP2 | Keybrd Out 2 | O | <High> | <High> |
| 58 | D0 | Data 0 | I/O | D0 (I) | DIN (I) |
| 59 | KOP3 | Keybrd Out 3 | O | (DOUT) | <High> |
| 60 | CCLK | (not used) | I | (CCLK) | CCLK (I) |
| 61 | A0 | Address 0 | O | A0 (O) | <High> |
| 62 | A1 | Address 1 | O | A1 (O) | <High> |
| 63 | A2 | Address 2 | O | A2 (O) | <High> |
| 64 | A3 | Address 3 | O | A3 (O) | <High> |
| 65 | A15 | Address 15 | I | (A15 (O)) | <High> |
| 66 | A4 | Address 4 | O | A4 (O) | <High> |
| 67 | A14O | Address 14 | O | A14 (O) | <High> |
| 68 | A5 | Address 5 | O | A5 (O) | <High> |

I = Input, O = Output, <High> = High Impedance (weak pull-up)

==========

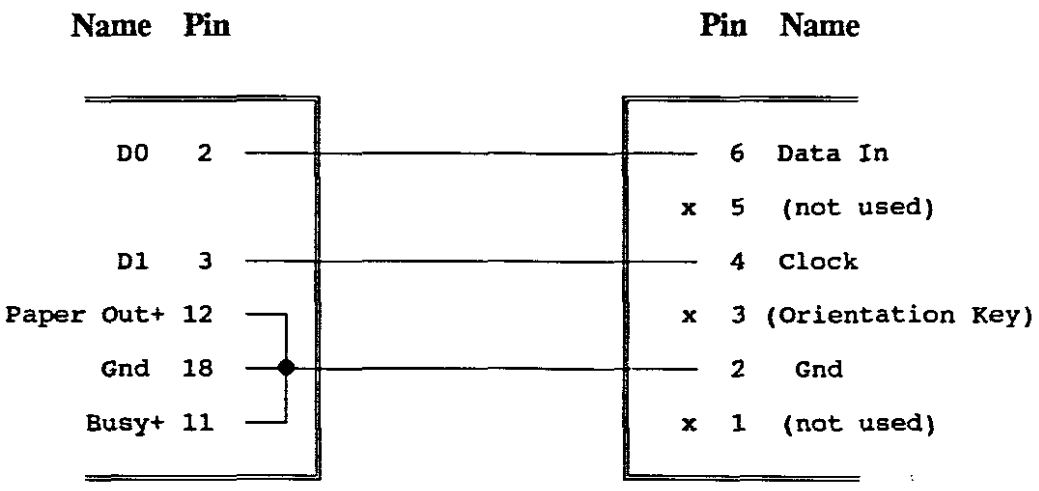
# Appendix G: The Seeding Serial Data Download Cable

(On the PC Side)

**25-Pin D-Connector** (Male)

(On the Board Side)

**6-Pin Header** (Female)




==========

# Appendix H: Listing of the Seeding Driver Program: SEED.EXE

```
#include <stdio.h>
#include <stdlib.h>
writebits(char filename[13], char port[5], char hilo);
main(void)
{  char prt,ch1,ch2;
   char port[6]="LPTn:",filename[13];
   FILE *ss, *cs, *os, *pr, *op, *inp;
   system("cls");
   puts("\t\t\tSerial-Parallel SEEDING Program.");
   puts("\n\tThis program is used to place some intelligence on a new");
   puts("\tcontroller board with a blank or corrupted Operating System or");
   puts("\tconfiguration for the Xilinx IC, in the Code RAM.");
   puts("\n\tThe Xilinx IC in the \"Serial Slave Mode\", is first configured to");
   puts("\tbe a Shift Register and an Address Counter through this program and");
   puts("\tthe file \"SHIFTREG.MCS\".");
   puts("\n\tThe configured Shift Register is then used to convert a clocked");
   puts("\tserial data train from a PC to load the Configuration data for");
   puts("\tthe Xilinx chip for normal operation, from the file DECODER.MCS");
   puts("\tinto the top of the Code RAM.");
   puts("\n\tThen the code of the Operating System from the file OPSYS.HEX, is");
   puts("\tloaded into the bottom of the Code RAM.");
   printf("\n\tPress ENTER to Continue...");
   getchar();
   system("cls");
   puts("\tThe loading of the above files, or the \"seeding\" process, is done");
   puts("\tthrough the Xilinx download cable from a parallel printer port of");
```

```
puts("\ta PC to the keyed 5-pin header plug on the controller board. A");
puts("\tmade up cable from the printer port to either a 6-pin header");
puts("\tsocket to the 8-pin DIL socket on the controller board can be");
puts("\tused. The specification for such a cable is in the documentation.");
puts("\n\tThe seeding process is controlled by this program, in conjunction");
puts("\twith the slide switches on the Controller Board. The jumper on the");
puts("\tboard's component side, marked \"SEED\" must be closed to initiate");
puts("\tseeding. Switch 1 resets the address counter when OFF, and Switch 2");
puts("\tcontrols the direction of the address counter: OFF = From the top");
puts("\tdownwards, and when switched ON = From the bottom, upwards.");
printf("\n\tPress ENTER to Continue...");
getchar();
system("cls");
puts("\t\t\tStep-by-Step Seeding Procedure:\n");
puts("\t 1. Connect the Controller Board to a parallel port of the PC.\n");
puts("\t 2. Close the \"SEED\" jumper on the Controller Board.\n");
puts("\t 3. Apply power to the board.\n");
puts("\t 4. Press the ON/CONF button.  The Red LED on the");
puts("\t    Controller Board should light up.");
if((ss=fopen("shiftreg.mcs","r"))==(FILE *) NULL)
{ puts("\n\nUnable to find SHIFTREG.MCS in current directory!\a");
  exit(0);
}
if((cs=fopen("decoder.mcs","r"))==(FILE *) NULL)
{ puts("\n\nUnable to find DECODER.MCS in current directory!\a");
   exit(0);
}
if((os=fopen("opsys.hex","r"))==(FILE *) NULL)
```

```
{ puts("\n\nUnable to find OPSYS.HEX in current directory!\a");
  exit(0);
}
fclose(ss);
fclose(cs);
fclose(os);
puts("\n\t 5. Select the appropriate printer port of the PC:");
puts("\n\t\t1. LPT1:\n\t\t2. LPT2:\n\t\t3. Quit.");
do
{ printf("\n\t\t(1,2,3):");
  scanf("%c",&prt);
  if(prt=='3') exit(0);
  if(prt=='2') port[3]='2';
  if(prt=='1') port[3]='1';
}
while (prt<'1'||prt>'3');
if((pr=fopen(port,"w"))==(FILE *) NULL)
{ printf("\n\nParallel Port %s may not be ready!\a",port);
  printf("\nCheck the cable and if power is available on the board.");
} fclose(pr); system("cls"); printf("\t\tPort %s is selected!",port);
puts("\n\n\t 6. Press ENTER to start the Shift Register configuration.");
printf("\t    Or Press Q to Quit..");
ch1=getchar();
if(ch1=='Q'||ch1=='q') exit(0);
system("cls");
printf("\n\n\t Configuring the Xilinx to a Shift Register.....");
writebits("SHIFTREG.MCS",port,'1');
puts("\aDone!");
```

```
    puts("\n\t 7. Open the \"SEED\" jumper. (Done with Shift Register configuration.)");
    puts("\n\t 8. Set Switch 1 OFF and then ON again, to reset the internal");
    puts("\t address counter. Switch 2 stays OFF to select loading from");
    puts("\t    the top, downwards.");
    puts("\n\t 9. Press any key to continue loading the configuration data for");
    printf("\t normal operation, into the top of the RAM, or Q to quit:");
    ch1=getchar();
    if(ch1=='Q'||ch1=='q') exit(0);
    system("cls");
    printf("\n\t    Loading the configuration data to the RAM.....");
    writebits("DECODER.MCS",port,'h');
    puts("\aDone!");
    puts("\n\t10. Set Switches 1 OFF then both 2 ON. Switch 1 is the address counter");
    puts("\t reset. Switch 2 selects the upwards count direction.");
    puts("\n\t11. Press any key to continue loading the code of the operating");
    printf("\t    system into the bottom of the RAM, or Q to quit:");
    ch1=getchar();
    if(ch1=='Q'||ch1=='q') exit(0);
    system("cls");
    printf("\n\t    Loading the Operating System....");
    writebits("OPSYS.HEX",port,'h');
    puts("\aDone!");
    puts("\n\t12. Remove the download cable, set all switches to OFF, and press");
    puts("\t    the ON/CONF button of the Controller Board.");
    puts("\n\t13. Now the Controller Board is Seeded and ready!");
    return(0);
}
```

```
writebits(char filename[13], char port[5], char hilo)
{       char ch1,ch2;
  int i,j,no_bytes,startln,endln,byte_val,bitv,parv,bt1,bt2;
  FILE *inp,*pr;
  inp=fopen(filename,"r");
  pr=fopen(port,"a");
  putc((char)255,pr);     /* Set all Data bits = 1 */
  do
  {       startln = 0;
    endln = 0;
    do
      ch1 = getc(inp);   /* read until ':' is found.*/
      while(ch1!=':');
    ch1=getc(inp);       /* read number of bytes in this line */
    ch2=getc(inp);
    bt1=(int)ch1-48;
    if (bt1>9) bt1=bt1-7;
    bt2=(int)ch2-48;
    if (bt2>9) bt2=bt2-7;
    no_bytes=bt1*16+bt2;
    for(i=1;i<7;i++)          /* read 6 ch Address & control */
    { ch1=getc(inp);
    }
    if(ch1=='2')       /* 2 = start line */
      startln=1; /* do nothing, go back to : */
    if(ch1=='1')
      endln=1;          /* 1 = end of data */
```

```
for(i=no_bytes;(i>0)&&(startln==0)&&(endln==0);i--)
{  ch1=getc(inp);
   ch2=getc(inp);
   bt1=(int)ch1-48;
   if (bt1>9) bt1=bt1-7;
   bt2=(int)ch2-48;
   if (bt2>9) bt2=bt2-7;
   byte_val=bt1*16+bt2;
   for(j=1;j<9;j++)
   {     if (hilo=='l')
         bitv=byte_val % 2;
      else
         bitv=byte_val / 128;
      parv=180+bitv;    /* Clock = 0 */
      if((char)parv<'┤'||(char)parv>'╖') exit(0);
      putc((char)parv,pr);
      if (hilo=='l')
         byte_val=byte_val/2;
      else
         byte_val=2*(byte_val % 128);
      parv=parv+2;      /* Set Clock */
      if((char)parv<'┤'||(char)parv>'╖') exit(0);
      putc((char)parv,pr);
   }
  }
}
```

```
    while(endln==0);
    putc((char)255,pr);
    fclose(inp);
    fclose(pr);
    return(0);
}
```

==========

# Appendix I: Useful Subroutines Already Existing in the Operating System

When a user program is run on the training board from address 0000H, the subroutines already in the Operating System, are directly available and can be used in user applications. The user only has to use a subroutine CALL to the correct address in his program. To prevent the unwanted corruption of existing data, summaries of the register usage and stack requirements, have been included on the next page, so that provision can be made by the user. The number of stack bytes includes the stack bytes required by the subroutine, and other subroutines called from within. The starting addresses of the subroutines in the Operating System are also included.

Some of these subroutines make use of other subroutines, but those CALL addresses are already contained in the code. When subroutines are used that also use interrupt subroutines, the user must include an appropriate interrupt subroutine in his program. A single RETI instruction will be sufficient as a user interrupt subroutine in all the cases.

When a user program that uses some of these routines is tested on the simulator, and error would be flagged if these routines do not exist on the simulator. One may either type out and include the routines together with the user program, or a single "RETURN" statement and a breakpoint can be placed at the appropriate address. The result of the subroutine can then be simulated by typing in the expected value into the appropriate registers of the simulator.

## Wait for a Programmable Period Subroutine

**Description:** Goes into the Idle Mode and waits for a programmable period, form 0.2 msec up to 13.1 seconds, in steps of 0.2 msec.

**Usage:** Call subroutine with the loop counter in the DPTR. Calculate time as (65536 - DPTR) x 0.2 msec.

**Corrupt:** Timer 0, DPTR, Acc and 4 stack bytes.

**Subroutines Used:** Timer 0 Interrupt (RETI only)

**Starting Address:** C:604Dh

**(Wait for 1 second:** C:604Ah)

(Listing on page 225)

## Reset and Clear the LCD Display Module

**Description:** Executes the Reset Display command, and waits a bit more than the prescribed minimum 1.64 msec.

**Usage:** Call subroutine.

**Corrupt:** Timer 0, Acc and 6 stack bytes.

**Subroutines Used:** Timer 0 Interrupt (RETI only).

**Starting Address:** C:6073h

(Listing on page 226)

## Display a 16-Character Message on the Display Module

**Description:** Displays a user defined 16-character message on the LCD display module.

**Usage:** Place 16 ASCII characters as Data Bytes somewhere in the code memory, set the DPTR to this address and call the subroutine.

**Corrupt:** Timer 0, DPTR, Acc, B, C, 8 stack bytes.

**Subroutines Used:** Timer 0 Interrupt (RETI only)

**Starting Address:** C:607Fh

(Listing on page 226)

## Wait for and Decode a Key from the Keyboard

**Description:** Wait in the Idle mode until a key is pressed. Decode the key and return. When entered, it will wait until a previous key has been released and the new key pressed.

**Usage:** Return with the key value, 00h to 1Fh in the Accumulator. The three MSBits have been masked out and cleared.

**Corrupt:** DPTR, Acc, B, C, EX0=1 and 7 stack bytes in SS or BP modes, or 4 stack bytes in Exe mode.

**Subroutines Used:** Operating System interrupt 0 in SS & BP, or User Interrupt 0 in EXE mode (RETI only).

**Starting Address:** C:60CBh

(Listing on page 227)

## Select BAUD Rate Subroutine

**Description:** Uses the LCD to display and select a baud rate together with the arrow and the Enter keys. The Help file is available and can be called by the Help button.

**Usage:** Return with the BAUD RATE set, but not running.

**Corrupt:** Both Timers, DPTR, Acc, R4 to R7 and 13 stack bytes.

**Subroutines Used:** External Interrupt 0, Timer 0 (RETI only)

**Starting Address:** C:6185h

(Listing on page 176)

## Binary Nibble to ASCII Conversion

**Description:** Converts the least significant nibble in the Accumulator to its ASCII equivalent (0h - Fh, to '0' - 'F').

**Usage:** Enter with value in Accumulator, mask out the MS Nibble and exit with the ASCII value in the Accumulator.

**Corrupt:** Acc and 2 stack bytes.

**Starting Address:** C:620Ch

(Listing on page 228)

## ASCII to Binary Conversion

**Description:** Converts and ASCII number form '0' to 'F' to its binary value.

**Usage:** Enter with ASCII number in Accumulator, and exit with the binary value in the Accumulator.

**Corrupt:** Acc and 2 stack bytes.

**Starting Address:** C:6218h

(Listing on page 228)

==========

## Appendix J: The LCD Display Module and Control Commands

The LCD display module is controlled by its own dedicated onboard micro-controller and communication to and from the board is done through 4 or 8 data- and 3 control lines. The display module can almost be treated as a simple peripheral device, except that the prescribed reset sequence and timing requirements must be strictly adhered to. The same internal controller and software are used for similar but bigger display units, such as 20, 24, 32- and 40-characters by 1-line, or the same number of characters by 2-lines. During the initialization sequence of the display module, the internal controller must be told which type of display it is controlling.

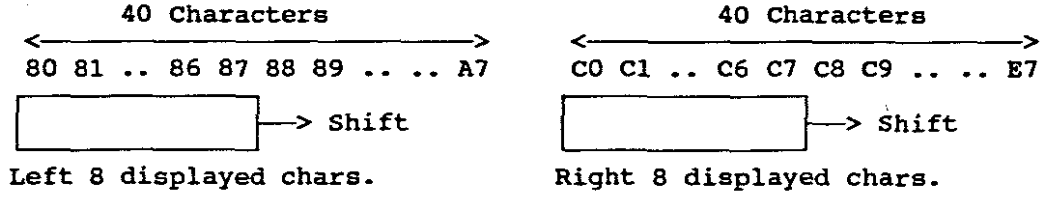
The control lines are a "Command/Data" line, a "Read/Write" control and a positive active "Enable" control. The data lines and the two command lines must be stable before the Enable control can go positive. It must stay active for more than 450 nsec before it can be pulled low again. After a normal instruction to the display controller, a waiting period of at least 40 $\mu$sec must be allowed before the next instruction.

To adhere to these requirements, the display module had to be decoded at separate reading and writing addresses on the Training Board. The module is decoded to only read form, or only write to the appropriate addresses, else the outcome may be indeterminate.

| | |
|---|---|
| The Command Write address is: | F700h |
| The Data Write address is: | F701h |
| The Status Read address is: | F702h |
| The Data Read address is: | F703h |

In the display module, there are two sets of RAM accessible to a user, the "Display Data RAM" or DDRAM and the "Character Generator RAM" or CGRAM. The DDRAM is for the storage of the ASCII code of the displayed character, and the CGRAM is used for the bit image of eight user definable characters. The DDRAM consists of two sets of 40 characters each, of which only 8 bytes of *each* set is displayed on the 16-character display. The addresses that contain the left eight characters are from 80h to A7h, and the right-hand eight characters are from C0h to E7h. The reset window for the displayed characters is at the lowest addresses of each section, i.e. 80h to 87h, and C0h to C7h.

**DDRAM Organization:**




The display windows can be shifted to cover any eight consecutive bytes in a continuous loop, i.e. 80h will follow A7h, and C0h will follow E7h. The windows of the two sections can only be shifted simultaneously. The position of the cursor determines at which address the next data byte will be placed. Initialization system commands will determine if the cursor position must increment or decrement when a character is written to the DDRAM, or that the cursor position must stay steady, and the displayed characters moves left or right past the cursor position.

Data can also be written to characters not currently displayed. They will then appear when the displayed characters are moved left or right. The two sections can only be shifted simultaneously, although the cursor is set to be in only one section.

A display reset command will clear the whole DDRAM. The ASCII number 32, a "space", will be written to all the addresses. The position of the display window will be placed over the first 8 bytes of each section, and the cursor will be placed on address 80h, the first left-hand character of the 16-character display.

With the display set to the cursor increment mode, a message can be written to the display by first placing the cursor on the address 80h by a "cursor placement" command to the command address. After the prescribed 40 $\mu$sec waiting period, the first character can be written in its standard ASCII code to the Data Write address. After the same waiting period, the next character can be written, and so on, for the first eight characters. Then the cursor must be placed on the first character of the second section, C0H, before the rest of the characters are written to the data address.

## The Character Generator RAM

All the standard ASCII characters are stored in the character generator ROM, plus a few lesser know graphic and Japanese characters. A complete list is included at the end of this section. It is also possible to generate eight user characters on a 5 by 8-bit map in the Character Generator RAM or CGRAM. The ASCII codes 0 to 7 have been allocated to display these user defined characters. Addresses 40h to 7Fh (8 sets of 8 bytes each) have been allocated to store the byte values of the user characters. Only the five least significant bits of each byte are used.

To define a user character, the required address of the first byte of one of the user characters must first be determined by the formula: 40h + n x 8, where n is the number of the user character, and also the ASCII code ('0' to '7') by which it must be called.

The bit image of the user character must be designed on a 5 by 8 grid, as shown below: A '1' will set the appropriate pixel and a '0' will clear the pixel. The byte values of the pattern must then be determined. The cursor is placed on the first byte address calculated above (40h to 7Fh) through the Cursor Place command. After that the group of data bytes can follow. They must be written to the Data Write address of the display module. The internal cursor address of the CGRAM counter will be incremented automatically after each data write operation.

## CGRAM Organization

| **Character 0:** | **Address** | **Byte Value** | **Bit values to display '↑'** |
|---|---|---|---|
| | 40h | 04h | x x x 0 0 1 0 0 |
| | 41h | 07h | x x x 0 1 1 1 0 |
| | 42h | 15h | x x x 1 0 1 0 1 |
| | 43h | 04h | x x x 0 0 1 0 0 |
| | 44h | 04h | x x x 0 0 1 0 0 |
| | 45h | 04h | x x x 0 0 1 0 0 |
| | 46h | 04h | x x x 0 0 1 0 0 |
| | 47h | 04h | x x x 0 0 1 0 0 |

| **Character 1:** | **Address** | **Byte Value** | **Bit values to display '↓'** |
|---|---|---|---|
| | 48h | etc. | etc. |
| | 49h | | |
| | .. | | |
| | 4Fh | (Last byte of character 1) | |

| Char. 2 to 7: | Address | Byte Value | Bit values to display |
|---|---|---|---|
| | 50h | etc. | etc. |
| | 51h | | |
| | .. | | |
| | .. | | |
| | 7Fh | (Last byte of character 7) | |

Whether a set of data is written to the DDRAM or the CGRAM, will depend on the address of the "last" cursor position selected. If it was between 40h and 7Fh, the CGRAM will be accessed, while if it was between 80h and A7h, or between C0h and E7h, the DDRAM will be accessed.

## Reading the Current Data or an Addresses from the Display Module

It is also possible to read either the data stored in the DDRAM or the CGRAM, or the cursor address counter value of either the DDRAM or the CGRAM.

The current cursor address is read from the Status (or Address) Read address and depending on which RAM the last cursor place command was addressed to, the return data will either be that of the DDRAM or the CGRAM. The most significant bit from either address counter will contain the "Busy Flag". When it is set, it will indicate that the display module controller is still busy with internal operations and any commands or data to the module will be ignored.

The stored data of the DDRAM or the CGRAM can also be read through the Data Read address, and the previous cursor place command will determine from which RAM and from which address the data will be returned. Reading data from either RAMs, will automatically increment the address counter for the reading of the next data byte immediately.

## Command Instructions of the LCD Display Module (In the Hex format)

**Initial Display Type Setup Instructions:** (40 $\mu$sec execution time)

20 = 4 Data bits, 1 Line Display Unit, 5 x 7 Character bit map

24 = 4 Data bits, 1 Line Display Unit, 5 x 10 Character bit map

28 = 4 Data bits, 2 Line Display Unit, 5 x 7 Character bit map

2C = 4 Data bits, 2 Line Display Unit, 5 x 10 Character bit map

30 = 8 Data bits, 1 Line Display Unit, 5 x 7 Character bit map

34 = 8 Data bits, 1 Line Display Unit, 5 x 10 Character bit map

38 = 8 Data bits, 2 Line Display Unit, 5 x 7 Character bit map (*)

3C = 8 Data bits, 2 Line Display Unit, 5 x 10 Character bit map

(* = Used for this device)

**Set Cursor or Display Movement from Here Onwards Commands.** (40 $\mu$s)

04 = Display will stay steady and Cursor position will Decrement.

05 = Display will move Right and Cursor position will stay steady.

06 = Display will stay steady and Cursor position will Increment.

07 = Display will move Left and Cursor position will stay steady.

**System Commands** (1.64 msec Execution time each)

01 = Reset display and cursor positions and clear DDRAM.

03 = Reset display and cursor *positions* only - no clearing.

**Show Cursor and Display Commands:** (40 $\mu$sec)

08 = Display OFF, Cursor OFF, Cursor character NOT blinking.

09 = Display OFF, Cursor OFF, Cursor character blinks.

0A = Display OFF, Cursor ON, Cursor character NOT blinking.

0B = Display OFF, Cursor ON, Cursor character blinks.

0C = Display ON, Cursor OFF, Cursor character NOT blinking.

0D = Display ON, Cursor OFF, Cursor character blinks.

0E = Display ON, Cursor ON, Cursor character NOT blinking.

0F = Display ON, Cursor ON, Cursor character blinks.

(When the Cursor is ON, the underscore character appears.)

**Shift Cursor or Display Commands:** (40 $\mu$sec)

10 = Cursor moves 1 character Left, display stays steady.

14 = Cursor moves 1 character Right, display stays steady

18 = Display moves 1 position Left, cursor stays on same character

1C = Display moves 1 position Right, cursor stays on same character

**Place Cursor and Select DDRAM or CGRAM Commands:** (40 $\mu$sec)

40 to 7F = Place cursor at CGRAM Addresses

80 to A7 = Place cursor at 1st window of DDRAM addresses.

C0 to E7 = Place cursor at 2nd window of DDRAM addresses.

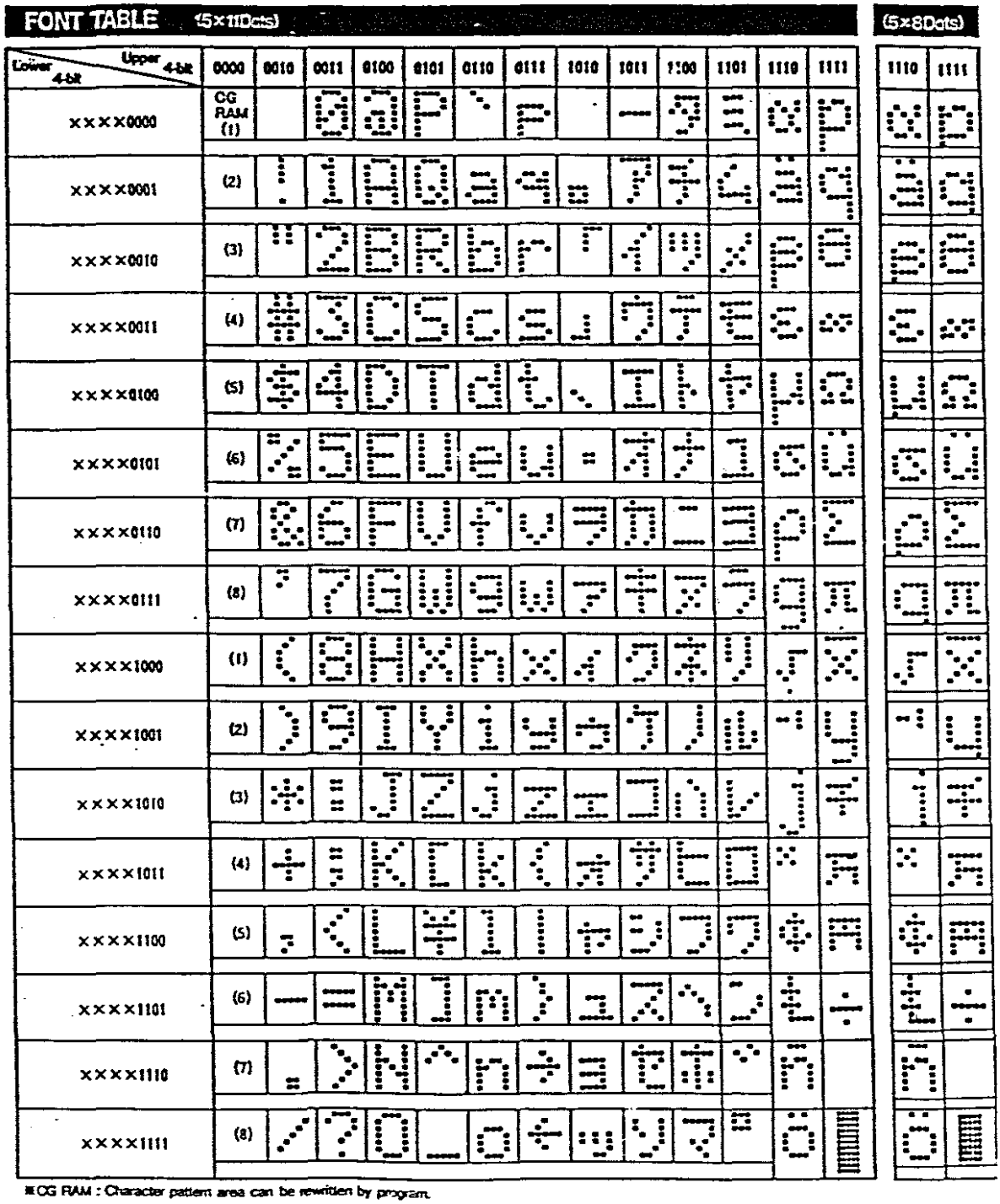
## The Display Module Initialization Sequence

1. Wait 15 milli-seconds after power up.
2. Write 38h to command address - select 8 data bit mode.
3. Wait 4.1 milli-seconds.
4. Write 38h to command address again - dummy byte in case the 4 data bit mode was active.
5. Wait 100 $\mu$-seconds.
6. Write 38h to command address - 8 data bits, 2 line, 5 x 7 font (This Module is considered to be a 2-line device.)
7. Wait 40 $\mu$-seconds.
8. Write 38h to command address - 8 data bits, 2-line, 5 x 7 font
9. Wait 40 $\mu$-seconds.
10. Write 01h to command address - reset module.
11. Wait 1.64 milli-seconds - delay for reset.
12. Write 06h to command address - Cursor Increment and display steady.
13. Wait 40 $\mu$-seconds.
14. Write 0Dh to command address - Display ON, Cursor OFF and cursor character blinks.

---

../The LCD Display Module Internal Character Set.

# The LCD Display Module Internal Character Set




※CG RAM : Character pattern area can be rewritten by program.

# Appendix K: Binary and INTEL Hex Format Files

The contents of a memory IC can be stored on disk, either in the Binary or the Hex formats:

In the Binary format, no address information is included in the file. The first byte of a Binary file starts at address zero, and each consecutive byte contains its true binary value. All undefined address spaces are filled with 00h or FFh, and the file may or may not end off with an "End of File" marker (ASCII 26d or 1Ah). The number of bytes contained in the file is defined in the directory of the disk. The file cannot successfully be inspected on a PC screen or printed out, because some values of the bytes may be unprintable characters. A surreptitious "End of File" marker, which can be a normal byte value, may cause an unexpected break.

In the INTEL Hex format, however, each program instruction or data byte value is represented as two hexadecimal digits from '0' to 'F'. Only bytes defined by the source program are included in the file. Open areas are not cleared to 00h and are not specified or included in the file.

The data bytes are grouped together in sixteen (or less) sets of two hexadecimal digits per line. It is assumed that all the bytes in one line will follow at consecutive addresses. When an open area is encountered, the previous line may end off with less than 16 data bytes, and the next line will start at the address where the next defined bytes continue.

The number of data bytes in a line, and the absolute starting address of the first data byte

of a line, are included at the beginning of each line. Each line of a hex file is treated as a separate entity, and lines need not be in a strict numerical address order. All the characters used in the Hex file are printable or readable characters between the ASCII numbers 32d and 127d.

Each line of data of an INTEL Hex file is compiled as follows:

```
:10123400112233445566778899AABBCCDDEEFF00CC
```

* A line always starts with a colon (:)
* The first two hexadecimal digits specify the number of data bits included in the line with a maximum of sixteen.
* The next four hexadecimal digits specify the 16-bit address of the first data bit, with ascending addresses for the following bytes in the same line.
* Two Control digits follow, indicating one of the following:

  00 = Normal data,

  01 = End of this section,

  02 = Start of a new section (Only used if unrelated data is stored in the same file or memory device)
* From 1 to 16 data bytes, made up of two hexadecimal digits each.
* A two hexadecimal digit checkbyte, which when added to the total of the values of each pair of digits in the line (carries ignored), will result in 00h.
* A Carriage Return/Line Feed character (ASCII 13d or 0Dh)

* The last line of a file is indicated by a line with zero data bytes, address 0000h, control digits 01, and a checkbyte:

  :00000001FF

* The file must be ended off by the End-of-file marker (ASCII 26 or 1AH) to close the file when loaded to the PC for disc storage.

==========

# Appendix L: Component List for the Training Board

| Ref Designator | Pattern | Type/Value |
|---|---|---|
| BAT | LED | PP3 (9V) |
| C1 | CAP200 | 100μF |
| C2 | CAP200 | 0.1μF |
| C3 | CAP200 | 10μF |
| C4 | CAP200 | 10μF |
| C5 | CAP200 | 10μF |
| C6 | CAP200 | 10μF |
| C7 | CAP200 | 10μF |
| C8 | CAP200 | 10nF |
| C9 | CAP200 | 10nF |
| C10 | CAP100 | 33pF |
| C11 | CAP100 | 33pF |
| C12 | CAP200 | 1μF |
| C13 | CAP200 | 1μF |
| C14 | CAP200 | 1μF |
| C15 | CAP200 | 0.1μF |
| CONF | LED | RED LED |
| CRM | DIP28 | 62256 |
| D1 | DO300 | Signal Diode |
| D2 | DO300 | Signal Diode |
| D3 | DO300 | Signal Diode |

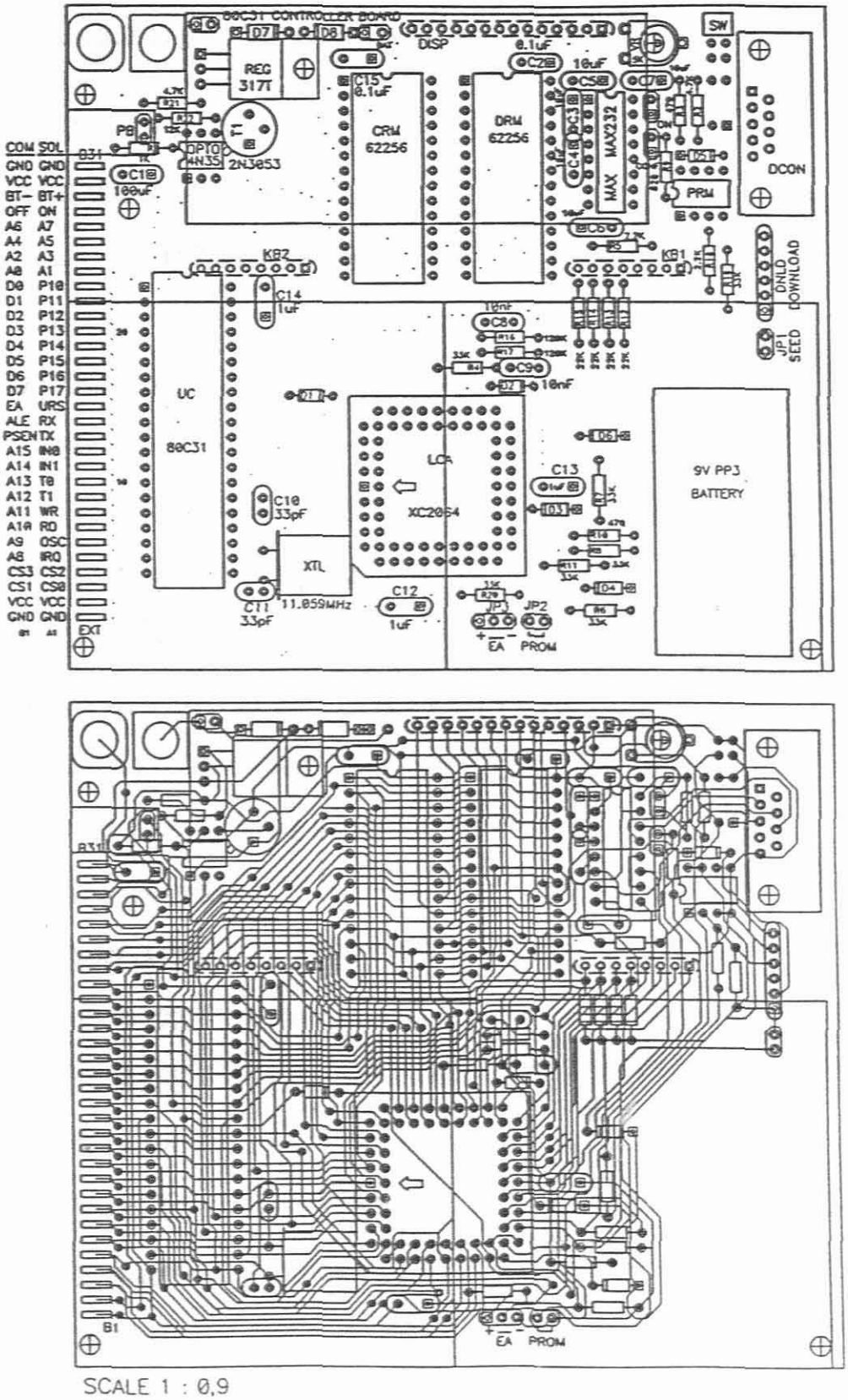
| Ref Designator | Pattern | Type/Value |
|---|---|---|
| D4 | DO300 | Signal Diode |
| D5 | DO300 | Signal Diode |
| D6 | DO300 | Signal Diode |
| D7 | DO350 | Rectifier Diode |
| D8 | DO350 | Rectifier Diode |
| DCON | D-9 | D-Connector |
| DISP | SIP14 | LTN111R-10 |
| DNLD | SIP6 | SIL6 |
| DRM | DIP28 | 62256 |
| EXT | IBMEDGE | 62P FEMALE |
| JP1 | SIP2 | SIL2 |
| JP2 | SIP2 | SIL2 |
| JP3 | SIP3 | SIL3 |
| KB1 | SIP8 | 16KEY# |
| KB2 | SIP8 | 16KEY# |
| LCA | PLC68 | XC2064 |
| MAX | DIP16 | MAX232 |
| ON | LED | Green LED |
| OPTO | OPTO | 4N35 |
| PB | SIP2 | SIL2 |
| PRM | DIP8 | XC1736 |
| R1 | RES400 | 1kΩ |
| R2 | RES400 | 4.7kΩ |

| Ref Designator | Pattern | Type/Value |
|---|---|---|
| R3 | RES400 | 470Ω |
| R4 | RES400 | 33kΩ |
| R5 | RES400 | 2.2kΩ |
| R6 | RES400 | 33kΩ |
| R7 | RES400 | 33kΩ |
| R8 | RES400 | 33kΩ |
| R9 | RES400 | 820Ω |
| R10 | RES400 | 470Ω |
| R11 | RES400 | 33kΩ |
| R12 | RES400 | 22kΩ |
| R13 | RES400 | 22kΩ |
| R14 | RES400 | 22kΩ |
| R15 | RES400 | 22kΩ |
| R16 | RES400 | 120kΩ |
| R17 | RES400 | 120kΩ |
| R18 | RES400 | 2.2kΩ |
| R19 | RES400 | 33kΩ |
| R20 | RES400 | 33kΩ |
| R21 | RES400 | 4.7kΩ |
| R22 | RES400 | 12kΩ |
| R23 | RES400 | 4.7kΩ |
| REG | REG | 317T |
| SW | DIP4 | DPDT |

| Ref Designator | Pattern | Type/Value |
|---|---|---|
| T1 | TO-5 | 2N3053 |
| TRM | TERM | Banana |
| UC | DIP40 | 80C31 |
| VR | TRMPOT | 4.7kΩ |
| XTL | XTAL | 11.059MHz |

==========

# Appendix M: Component Overlay and Connections




SCALE 1 : 0,9

# Appendix N: Micro-Controller Board Netlist

(Used for Faultfinding on the Board)

BAT+: TRM-1 D7-1

REG: D7-2 REG-3 EXT-A29 D8-2

OFF: LCA-11 OPTO-2 D1-1 EXT-B28

OF1: R3-2 ON-1

OF2: OPTO-1 ON-2

OF3: T1-2 OPTO-4

OF4: OPTO-5 R1-2

BAT-: BAT-2 TRM-2 PB-2 EXT-B29 T1-1

RST: PB-1 D2-2 R4-2 LCA-44 EXT-A28

DP: DNLD-5 D2-1 D1-2 LCA-45 PRM-3 PRM-4 D5-2

CCLK: LCA-60 LCA-24 R18-2

DCLK: DNLD-4 PRM-2 R18-1 R19-2

DIN: DNLD-6 R5-1 PRM-1

P10: UC-1 EXT-A23

P11: UC-2 EXT-A22

P12: UC-3 EXT-A21

P13: UC-4 EXT-A20

P14: UC-5 EXT-A19

P15: UC-6 EXT-A18

P16: UC-7 EXT-A17

P17: UC-8 EXT-A16

M1: LCA-25 LCA-27 R6-2 JP2-1

ENA: DISP-6 LCA-47 D5-1

EA: JP3-2 UC-31 EXT-B15

WR: CRM-27 LCA-12 DRM-27 UC-16 EXT-A8

D0: R5-2 DISP-7 CRM-11 LCA-58 DRM-11 UC-39 EXT-B23

A2: CRM-8 LCA-63 DRM-8 EXT-B25

A1: CRM-9 LCA-62 DRM-9 EXT-A24 DISP-5

A0: DISP-4 CRM-10 LCA-61 DRM-10 EXT-B24

CRM: LCA-30 R10-1 R9-2

CEC: R10-2 CRM-20 CRM-22 D3-2

DRM: LCA-49 DRM-20 DRM-22

M0: LCA-26 D4-2 R11-1

SEED: D3-1 D4-1 JP1-2

KOP7: KB1-4 LCA-36

KOP6: KB1-3 LCA-34

KOP5: KB1-2 LCA-33

KOP4: KB1-1 LCA-32

KOP3: KB2-4 LCA-59

KOP2: KB2-3 LCA-57

KOP1: KB2-2 LCA-55

KOP0: KB2-1 LCA-53

KIP0: KB1-5 KB2-5 R12-2 LCA-37

KIP1: KB1-6 KB2-6 R13-2 LCA-38

KIP2: KB1-7 KB2-7 R14-2 LCA-39

KIP3: KB1-8 KB2-8 R15-2 LCA-40

C1P: MAX-1 C3-1

C1N: MAX-3 C3-2

C2P: MAX-4 C4-1

C2N: MAX-5 C4-2

VP: MAX-2 C5-1

VN: MAX-6 C6-2

RX': DCON-3 MAX-8

BP: LCA-31 SW-2 R7-2

D1: DISP-8 CRM-12 LCA-56 DRM-12 UC-38 EXT-B22

D2: DISP-9 CRM-13 LCA-54 DRM-13 UC-37 EXT-B21

D3: DISP-10 CRM-15 LCA-51 DRM-15 UC-36 EXT-B20

D4: DISP-11 CRM-16 LCA-50 DRM-16 UC-35 EXT-B19

D5: DISP-12 CRM-17 LCA-48 DRM-17 UC-34 EXT-B18

D6: DISP-13 CRM-18 LCA-42 DRM-18 UC-33 EXT-B17

D7: DISP-14 CRM-19 LCA-41 DRM-19 UC-32 EXT-B16

VO: VR-2 DISP-3

VOR: VR-3 R2-2

A15: LCA-65 UC-28 EXT-B12

A14: CRM-1 LCA-67 DRM-1 UC-27 EXT-B11

A13: CRM-26 LCA-2 DRM-26 UC-26 EXT-B10

A12: CRM-2 LCA-4 DRM-2 UC-25 EXT-B9

A11: CRM-23 LCA-6 DRM-23 UC-24 EXT-B8

A10: CRM-21 LCA-8 DRM-21 UC-23 EXT-B7

A9: CRM-24 LCA-9 DRM-24 UC-22 EXT-B6

A8: CRM-25 LCA-7 DRM-25 UC-21 EXT-B5

RD: LCA-17 UC-17 EXT-A7

A7: CRM-3 LCA-5 DRM-3 EXT-A27

A6: CRM-4 LCA-3 DRM-4 EXT-B27

A5: CRM-5 LCA-68 DRM-5 EXT-A26

A4: CRM-6 LCA-66 DRM-6 EXT-B26

A3: CRM-7 LCA-64 DRM-7 EXT-A25

RXD: MAX-9 D6-2

RX: D6-1 UC-10 EXT-A14

TX': DCON-2 MAX-14

TX: MAX-11 UC-11 EXT-A13

RTS: DCON-7 MAX-7

T0: UC-14 EXT-A10

CTS': DCON-8 MAX-13

CTS: MAX-10 MAX-12

T1: UC-15 EXT-A9

INT1: LCA-16 UC-13 EXT-A11

INT0: LCA-15 UC-12 EXT-A12

PSEN: LCA-13 UC-29 EXT-B13

ALE: LCA-14 UC-30 EXT-B14

URS: LCA-28 UC-9 EXT-A15 CONF-1

CONF: CONF-2 R9-1

OSC1: LCA-46 R16-2 C8-1

OSC2: LCA-43 R17-2 C9-1

XT2: UC-18 C10-1 XTL-2 EXT-A6

XT1: UC-19 C11-1 XTL-1

SS: LCA-29 SW-1 R8-2

CS0: LCA-23 EXT-A3

CS1: LCA-20 EXT-B3

CS2: LCA-22 EXT-A4

CS3: LCA-19 EXT-B4

IRQ: LCA-21 EXT-A5 R20-2

REGA: REG-1 R21-2 R22-1

BT2+: BAT-1 D8-1

VCC: C1-1 C2-1 C5-2 C7-1 C12-1 C13-1 C14-1 C15-1 JP3-1 REG-2 LCA-18 LCA-52 R1-1 R2-1 R3-1 R4-1 R6-1 R7-1 R8-1 R12-1 R13-1 R14-1 R15-1 R16-1 R17-1 R20-1 DNLD-1 PRM-7 PRM-8 DISP-2 CRM-28 DRM-28 UC-40 EXT-A2 EXT-B2 EXT-A30 EXT-B30 JP1-1 MAX-16 LCA-10 R19-1 R21-1

GND: LCA-1 LCA-35 C1-2 C2-2 C6-1 C7-2 C8-2 C9-2 C10-2 C11-2 C12-2 C13-2 C14-2 C15-2 DNLD-2 PRM-5 VR-1 R11-2 DISP-1 CRM-14 DRM-14 UC-20 EXT-A1 EXT-B1 EXT-A31 EXT-B31 JP2-2 SW-4 SW-3 DCON-5 MAX-15 JP3-3 T1-3 R22-2

==========

## Appendix O: Costing and Suppliers of the Components of the Board

One of the objectives of the project was to make the board as cheap as possible, using only components that are readily available. The implementation of the Field Programmable Array reduced the possible component count, the board size and the number of IC interconnections to a minimum. The PC board itself was used as the main frame of the project. All the external components were mounted and connected to the board by machine screws and PC board mounted sockets. The final board size was not decided by the number of components that had to contain, but by the size of the keyboard and the display unit.

Quotes for the components from various suppliers in Cape Town were requested, and the final cost of the 15 production boards, at 1994 component prices, are shown below:

## Bill of Materials for the Controller Board. (MK 4)

**Supplier:** MicroSource

| Quantity | Type | Unit Price | Total |
|---|---|---|---|
| **Integrated Circuits:** | | | |
| 15 | 4N35 OPTO-Isolator | R 1.60 | R 2.40 |
| 15 | LTN111R-10 16 Ch X 1 LCD Display Module | R 40.00 | R600.00 |
| **Semiconductors:** (Transistors, LEDs and Diodes) | | | |
| 15 | 2N3053 (1 Watt) TOS | R 0.75 | R 11.25 |
| 15 | GREEN LED (small) | R 0.28 | R 4.20 |
| 15 | RED LED (small) | R 0.20 | R 3.00 |
| 90 | Signal Diodes 50 mA | R 0.06 | R 5.40 |
| 30 | Rectifier Diodes 1 A | R 0.07 | R 2.10 |

**Miscellaneous:**

| | | | |
|---|---|---|---|
| 30 | 16-KEY Rasterscan Keyboards | R 16.00 | R480.00 |
| 15 | 11.059MHz Crystal | R 2.25 | R 33.75 |

**Plugs, Sockets and Connectors:**

| | | | |
|---|---|---|---|
| 15 | PLCC68 Socket (Tulip) | R 2.75 | R 41.25 |
| 15 | 40p IC Socket (Tulip) | R 3.60 | R 54.00 |
| 15 | 16p IC Socket (Tulip) | R 1.25 | R 18.75 |
| 15 | 8p IC Socket (Tulip) | R 0.60 | R 9.00 |
| 15 | 6p IC Socket (Tulip) | R 0.50 | R 7.50 |

**Plugs, Sockets and Connectors:**

| | | | |
|---|---|---|---|
| 15 | 9P-DCON (F) PCB Mounted | R 1.25 | R 18.75 |
| 15 | 9P-DCON (M) Cable connection + cover | R 4.50 | R 67.50 |
| 15 | 25P-DCON (F) Cable connection + cover | R 3.80 | R 57.00 |
| 15 | 62-Pin IBM Edge Connector | R 2.10 | R 31.50 |
| 15 | PP3 BATTERY Connector | R 0.39 | R 5.85 |
| 10 | Right Angled Header Strip 20-pin Single Line | R 0.77 | R 7.70 |
| 45 | 2-pin Jumpers | R 0.10 | R 4.50 |

**Capacitors:**

| | | | | |
|---|---|---|---|---|
| 15 | 100µF | Elect. Radial, 25V | R 0.15 | R 2.25 |
| 75 | 10µF | Tantalum 16V (5mm pins) | R 0.55 | R 41.25 |
| 45 | 1µF | Tantalum 16V (5mm pins) | R 0.35 | R 15.75 |
| 30 | 0.1µF | Ceramic (5mm pins) | R 0.13 | R 3.90 |
| 30 | 10nF | Ceramic (5mm pins) | R 0.18 | R 5.40 |
| 30 | 33pF | Ceramic (2.5mm pins) | R 0.18 | R 5.40 |

**Resistors:**

| | | | | | |
|---|---|---|---|---|---|
| 30 | 120kΩ | 5% | ¼ watt carbon | R 0.02 | R 0.60 |
| 105 | 33KΩ | 5% | ¼ watt carbon | R 0.02 | R 2.10 |
| 60 | 22KΩ | 5% | ¼ watt carbon | R 0.02 | R 1.20 |
| 15 | 12KΩ | 5% | ¼ watt carbon | R 0.02 | R 0.30 |
| 30 | 4.7KΩ | 5% | ¼ watt carbon | R 0.02 | R 0.60 |
| 30 | 2.2KΩ | 5% | ¼ watt carbon | R 0.02 | R 0.60 |
| 15 | 1KΩ | 5% | ¼ watt carbon | R 0.02 | R 0.30 |
| 15 | 820Ω | 5% | ¼ watt carbon | R 0.02 | R 0.30 |
| 30 | 470Ω | 5% | ¼ watt carbon | R 0.02 | R 0.60 |
| | | | | | R 1544.90 |
| | | | | Add 14% VAT | 216.29 |
| | | | | | R 1761.19 |

**Supplier:** Communica (Pty)Ltd.

| | | | |
|---|---|---|---|
| 15 | 80C31BH Micro-Controller | R 12.15 | R 182.25 |
| 15 | MAX232 | R 7.74 | R 116.10 |
| 15 | LM317T TO-220 Regulator | R 2.55 | R 38.25 |
| 15 | 4.7KΩ Preset Potentiometer with shaft (CA9MV + REF004) | R 0.79 | R 11.85 |
| 15 | 2 Pole Slide Switches KTSA02 (Vertical Mounted) | R 2.98 | R 44.70 |
| | | | R 393.15 |
| | | Add 14% VAT | R 55.04 |
| | | | R 448.19 |

**Supplier:** Tarsus Technologies

| | | | | |
|---|---|---|---|---|
| 30 | 62256 | 32K Static RAM | R 15-80 | R 474.00 |
| 30 | 28-Pin | SmartSockets | R 41-10 | R1233.00 |
| | | | | R 1707.00 |
| | | | Add 14% VAT | R 238.98 |
| | | | | R 1945.98 |

**Supplier:** Saftec Sales

| | | | |
|---|---|---|---|
| 15 | Push Buttons MS 402 | R 5.44 | R 81.60 |
| 15 | Banana Sockets RED | R 2.12 | R 31.80 |
| 15 | Banana Sockets BLACK | R 2.12 | R 31.80 |
| 60 | 32-pin SIL IC socket strips | R 3.67 | R220.20 |
| | | | R 365.40 |
| | | Add 14 % VAT | R 51.16 |
| | | | R 416.56 |

**Supplier:** Sames (Stellenbosch)

| | | | |
|---|---|---|---|
| 15 | XC2064-50 PC68C LCA | R 65.00 | R 975.00 |
| | | Add 14 % VAT | R 136.50 |
| | | | R 1111.50 |

**Supplier:** Hamrads

| | | | |
|---|---|---|---|
| 32 | meter 5-Core Cable (4-core + screen) | R 2.70 | R84.40 |
| | (VAT Inclusive) | | |

**Supplier:** WH Circuit

| | | | |
|---|---|---|---|
| 25 | PCBoards | R 27.90 | R 697.50 |
| 1 | Initial Charge for photoplot | | R 240.00 |
| | | | --- |
| | | | R 937.50 |
| | | Add 14% VAT | R 131.25 |
| | | | --- |
| | | | R 1068.75 |
| | Subtract 10 boards @ R45 each | | - R 450.00 |
| | | | --- |
| | | | R 618.75 |
| | | | ========= |

**Supplier:** Cape Plastics

| | | | |
|---|---|---|---|
| 25+25 | Perspex Back and Display Covers | | R 40.00 |
| 3 m | 6mm PVC Tubing | | R 14.02 |
| | | | --- |
| | | (VAT Inclusive) | R 54.02 |
| | | | ========= |

**Supplier:** D Byl & Co.

| | | | |
|---|---|---|---|
| 90 | Mounting Screws & Nuts | (VAT Inclusive) | R 52.95 |

**Supplier:** Meltzer Agencies

| | | | |
|---|---|---|---|
| 15 | PP3 9-volt Batteries (6F22K) | R 2.69 | R 40.35 |
| | | Add 14% VAT | R 5.65 |
| | | | --- |
| | | | R 46.00 |
| | | | ========= |
| | | Total Costs | R 6546.04 |
| | | Cost per board | R 436.40 |
| | | | ======== |

**Note:** The cost of the labour for the construction and testing of the boards was not included, and could be calculated to be approximately three hours per board at the current labour rate.

**Possible Cost Savings per Board:**

Only one Smart Socket may be used for the CODE RAM. (-R46.85)

Leave out the 8-pin socket and JP2 (Future Serial PROM) (- R0.60)

A cheaper ON button (- R6.20)

Non-throughplated PC boards are available at R25.00 (-R20.00)

Own Serial Download Cable available (-R14.86)

The components can be supplied in kit form for self construction by the students to save on labour costs.

==========