Internationally Convertible and Compatible Software

Spectrum Complete Machine Code Package

48K 128K Spectrum +2 +3

ROYBOT

ROYBOT CONVERTIBLE AND COMPATIBLE MACHINE CODE PACKAGE

by Roy Longbottom C.Eng., M.I.E.R.E., M.B.C.S.

CONTENTS

ROYBOT, 45 Hullbridge Road, Rayleigh, Essex, SS6 9NL

ROYBOT INTERNATIONALLY CONVERTIBLE AND COMPATIBLE MACHINE CODE SYSTEM

The following four programs are supplied:

Assembler BASIC driver	Assembler machine code
Test BASIC driver	Test machine code

The drivers are written in BASIC to provide the internationally convertible and compatible features, whereby any information appearing on the screen can be easily converted to a different language and any output can be directed to any channel that can be driven via BASIC. Special characters can be constructed by user defined graphics, as shown in the Spectrum manual, or by using the assembler: a range of French, German and Scandanavian characters are predefined in the software and can be entered in the program by using Graph Shift a to s. Machine code is used where speed is required, such as on assembly, disassembly, testing and output formatting. The software is driven by simple menu selections and prompted input, making it very easy to use.

The microdrive and disk versions have an additional auto-run program which has facilities for user defined character design.

ASSEMBLER SUMMARY

The assembler has been written particularly to suit beginners but it is also suitable for professional software, such as that produced by ROYBOT. Assembler input is in the lower case format given in Spectrum manuals and is entered in BASIC REM lines, with multiple instructions per line, if required: this means that there are no new complex editing procedures to be learnt, input files can be prepared on a Spectrum without needing to load special software and summary printed listings can easily be produced. An example of the similarity to BASIC, using line number labels, is:

```
10 LET a=10:GOSUB 20:GOTO 30  =  10 REM ld a,10;call L20;jr L30
20 LET a=a+100:RETURN            20 REM add a,100;ret
```

Alternatively, named labels can be used and comments added: this allows easier line renumbering and built in documentation:

```
 1 REM NUMBER 1;defb 10
 2 REM NUMBER 2;defb 100
10 REM START;ld a,(@NUMBER 1);ld b,a;ld a,(@NUMBER 2);call @ADDEM;jr @NEXT
20 REM ADDEM;add a,b;ret
30 REM NEXT;Comments can be added as this
```

Various menu controlled utility procedures are provided for merging and deleting input lines, saving assembled code, erasing and cataloguing files for tape, disks, microdrives or RAM disks.

The assemble menu allows all or selected lines to be assembled, assembly with or without listing and listing or label addresses to screen or printer. The code is always assembled to address 53000 onwards, but this is only used for relative addressing purposes, the final address being selected as a menu option. Long assembly input codes can be split into two or more sections and assembled as one through the process of merge lines, assemble, save code, delete lines, merge new lines, continue assembly, save new code and so on. The various sections of code can then be loaded to the real addresses and saved as one code file.

Lines 1 to 2000 are used for assembler input lines. An example of size is 400 input lines, with an average of 6 instructions per line, could occupy 20,000 bytes and generate 4000 bytes of code. This would be assembled in about 75 seconds. The memory is used as follows:

Below 53000	BASIC driver and input lines
53000 - 57249	Assembled machine code
57250 - 61251	2000 label addresses
61252 - 65367	Assembler variables and machine code
65368 - 65535	User defined characters

TEST SUMMARY

When the TEST program is loaded, a display appears, showing the computer's register contents, stack, flags, selected memory, last and next instructions with addresses. Underneath the menu options are shown. In this mode, the program is ready to accept assembler instructions, typed in one at a time for the beginner to learn all about machine code, in conjunction with the manual supplied. The menu options are:

1 Exit - this goes to a second menu, allowing code to be loaded from tape, disks, microdrives or RAM disks and also to disassemble it.

2 Step - this allows the code to be stepped and executed one instruction at a time, showing what effect it has on the registers, flags, stack etc. Also, at any time, new assembler instructions can be typed in to change the values.

3 Run slow - this steps through the code automatically at about 5 instructions per second, showing changes in register contents etc. dynamically.

4 Run fast - this steps at about 30 instructions per second but only displays instruction addresses until a stop is reached or the space bar pressed, when the full display is given.

5 Run CLS - this is the same as 4, without address display, and is provided to test programs producing screen output.

6 Backtrack - this allows the last 5 instructions and their effects to be replayed.

7 Dec/Hex - this switches all the displays between decimal and hexadecimal format.

8 Binary - this allows the contents of one selected register/pair to be displayed as binary.

9 Start address - allows selection of a new start address for the next instruction to be executed. It can also be used to step one instruction at a time without execution and showing disassembled instructions.

10 End address - slow or fast trace will stop when the selected end address is passed.

11 Breakpoint - tracing will stop when an instruction exactly at the breakpoint address comes up next for execution.

12 Memory display address - this can be for any 8 bytes of memory. Initially, it displays a buffer area showing the code produced by instructions typed in directly.

13 Real address - initially, the program can be used for testing or disassembling code at a real address e.g. the 48K Spectrum ROM. Selecting 13 will switch to virtual mode, so that code loaded can be regarded as being at a different address e.g. code assembled to start at address 30000, 40000 or whatever, can be tested as though it was loaded to the real address.

14 Step calls - normally subroutines are stepped at the slow rates, but selecting 14 can cause subroutine calls to be executed at the full rate of up to nearly 1 million instructions per second. Using real addresses, any depth of calls will be executed correctly. Using virtual addresses, only the first call can be executed correctly as address conversion cannot be carried out for subsequent ones.

15 Print - this sends the output to the printer and shows register contents, instruction address and mnemonic code as programs are stepped or when a trace stops.

The memory is used as follows:

Below 32800	BASIC driver and variables
32800 - 40949	test and disassembler machine code
40950 - 44999	assembler for direct input
45000 - 65367	machine code to be tested
65368 - 65535	user defined characters if required

Code to be tested can be loaded below 45000, as low as 40950 but, if this is selected, direct instruction input is disabled.

LOADING THE SOFTWARE

The tape version has the test software (test.bas & test.bin) on one side and assembler (ass.bas & ass.bin) on the other. To load and start, reset the computer and use the usual LOAD "". Program test.bas can be loaded via an assembler menu but to load the assembler always reset or type CLEAR 52999 first.

The microdrive and disk versions have auto-load programs "run" and "disk". For the former, enter NEW (select BASIC on 128K) and enter RUN. On the +3, select "Loader" to load "disk". The main software can the be loaded by selecting menu options 1 or 2.

COPYING THE SOFTWARE

On receiving the software, a copy should be made for normal use and the original stored in a safe place. In line with the design objectives of putting the user's requirements first and easy conversion for other disks etc., the software provides facilities for making the copies. Please do not abuse this user friendly facility for making illegal copies for others: as the purchaser, you are only licensed to make extra copies for your own use.

To copy the TEST Basic and code, on loading, select menu option 1 EXIT (press ENTER), then 4 SAVE ROYBOT from the second menu. Tape, RAM disk or drive can the be selected, as appropriate, the drive number or letter being requested for the latter.

To copy the assembler Basic and code, on loading, select menu option 4 SAVE, then 2 ROYBOT code/BASIC, followed by tape etc., as above.

The microdrive loader can be saved by selecting menu option 3. On the disk version this option copies all the software to another disk.

CONVERSION

All variables that are displayed are declared at the start of the BASIC drivers (TEST lines 20-450, ASSEMBLER lines 2020 to 2560, LOADER lines 20 to 250). These can be changed by the user to terms that he understands better or to a different language. To assist in the latter, Graph Shift characters a to s have been defined to represent special French, German and Scandanavian characters at the end of the assembler code: these can be changed as shown in the example in the assembler section or via the drive auto-load program. The code (65368 to 65535) can be saved and loading arranged for use in the TEST program (automatic with auto-load) but care must be taken that the area is not overwritten by machine code that is being tested. The length of the words displayed is limited by the dimensioned arrays, which cannot be changed (see TEST line 1600, ASSEMBLER 4380, LOADER 990).

If it is necessary to initialise a channel, send special characters to a printer etc., it can be arranged by selecting INITIALISE from the secondary TEST menu or the main ASSEMBLER menu: these GO TO 460 and 2580 respectively, where additional BASIC statements can be included.

All accesses to tapes, disks etc. follow the variables, at the start of the programs, and can be changed for other devices. The BASIC lines are:

	Tape	Drive	RAM Disk
TEST loading code	490	500	510
saving ROYBOT software	540	550	560
start line to load test code	580	590	600
ASSEMBLER loading BASIC program	2640	2620	2630
merging input lines	2670	2650	2660
saving assembled code	2700	2680	2690
saving ROYBOT BASIC and code	2740	2720	2730
saving all BASIC	2760	2770	2750
start line to load code	2790	2800	2780
CATALOG		2830	2840
ERASE file		2860	2870

For drive identification, n$ is used for a letter or VAL n$ for a number.

For the drive autoloader, lines 270 to 300 and 350 to 390 can be changed.

Printing is carried out using LPRINT at lines 620 to 630 for TEST and 2900 to 3080 for ASSEMBLER. These can be changed to print statements peculiar to certain interfaces.

THE PROCESSOR

The heart of the Spectrum is a microprocessor chip known as a Z80. This has its own internal fast memory elements, known as registers. Registers a,b,c,d,e,h,l can store one 8 bit byte of value up to 255 (see Spectrum manual Binary and Hexadecimal). Some of these can be combined to give 16 bit words of value up to 65535: they are bc, de and hl. Other registers, ix, iy and sp, can only be used for words. The size of the words limits the amount of main memory that can be accessed by the processor to 65536 (0 to 65535) bytes. 128K systems work by switching this range of accessing connections to different blocks of memory.

The Z80 has over 700 different machine code instructions. Most of these are quite simple, loading bytes or words between registers or registers and memory, manipulating individual bits, GOSUB and GOTO type instructions and simple arithmetic operations.

The 8 bits of a special register (f), which can also be combined with a to give af, are used as flags. These give an indication of the effect of certain operations, such as result of zero produced.

The processor requires an area of memory to be set aside as a stack, the address being contained in the sp register. Words are inserted at the top of the stack by using push instructions and the last entry removed by pop. The stack is also used automatically to provide the return address on calling subroutines.

TESTING FACILITIES

Load "test.bas" and follow the following examples, which demonstrate how to use the facilities. On loading, all the registers, except IY, are set to zero and displayed at the top left of the screen. IY is given the value normally used by the Spectrum ROM of 23610 or the address of system variable ERR NR (see Spectrum manual). The bc, de and hl register contents are shown for the register pairs and b,c,d,e,h,l separately. At the top right, the stack contents are shown for the last 3 entries, where the whole word and each of the two bytes are shown. The stack pointer (SP) value is the one used by the ROYBOT software.

The status of the flags is shown underneath the registers. Under these "was" shows the address and assembly input code for the last instruction executed and "next" the one to be executed next. Finally, across the middle of the screen, 4 words (8 bytes) of memory are displayed.

The following demonstrate use of the software. Follow the instructions carefully until you are used to using the software as it is so easy to cause the system to crash or NEW itself when using machine code.

HEXADECIMAL

Pressing 7 and ENTER switches the displays between decimal and hexadecimal format (see Spectrum manual for explanation).

BINARY

Initially, the binary values of the af register pair are displayed at the bottom right of the screen. Entering 8 provides a prompt for a different pair of registers: type iy and ENTER, noting the changes then repeat to reselect af.

DIRECT INSTRUCTION INPUT

Except when in the middle of one of the menu option procedures, the software will accept and execute instructions typed in directly. On loading, the memory display shows a buffer area where the machine code numbers are inserted for instructions entered from the keyboard. Typing in the following, one at a time with ENTER, demonstrates the displays. The beginner should go to the section "TEACH YOURSELF MACHINE CODE ON-LINE" after trying these.

Instruction	Code	Result
dec hl	93	hl becomes 65535, h and l each 255
push hl	229	the hl values appear on the stack, No. on stack = 1
pop de	209	de becomes the same as hl, stack is empty
add hl,de	25	hl becomes 65534 and the carry flag is set
ld a,65	62 65	a becomes 65, the flags are not changed

BACKTRACK

Enter 6 to playback the above, observing the same effects, pressing ENTER for the next instruction.

START ADDRESS

Enter 9, then 16 in response to the request for the start address. Note that the next instruction to be executed is at address 16. This is a restart routine (rst 16) in the Spectrum 48K ROM which displays the character in the a register.

STEP

Enter 2 and observe that the instruction has executed and becomes "Was". Next press ENTER 4 or 5 times and note the effects of further executions.

RUN SLOW

Enter 3 to trace through the code slowly. After about 45 seconds the letter A will be formed in the bottom left hand corner of the screen. The program will stop with an "INVALID OP." message, meaning there was no return address on the stack to go to. Enter 3 new instructions - ld a,65 ld bc,16 push bc -. Select 9 and enter start address 16 again, then 3 Slow trace. This time the A is displayed, followed by another character as the return picks up address 16 for a second pass. If some other value had been left on the stack, it is possible that the system would crash.

RUN FAST

Select 9 and enter address 16 again, then 4 to run fast. Repeat but press the space bar when execution has started, noting that the full display is given on stopping. Enter 4 again to complete the routine.

RUN CLS

Select 9 but for the address input H10 (hexadecimal for 16). Then select 5 to run through the code with the screen blank.

BREAKPOINT

Select 11 and enter address 2800. Select 9 and address 16 again, then 4 for fast trace. Enter 4 again (or 2,3 or 5) and the program executes until the breakpoint is reached again: then 4 again to go to the end.

END

Change the breakpoint address back to 0, 10 End to 5676, start address to 16 and 4 to run fast. The execution stops with an end message. This time selecting 2,3,4 or 5 does not lead to any further instructions being executed. Change the end address to 65535, then enter 4 to continue.

MEMORY DISPLAY

Select 12 and enter 23635. This is the address of the system variable PROG, the display showing the start address of the BASIC program. The fourth word shows the end of variables E-LINE.

STOP AND CLEAR

In order to restore the memory display to its original value or to reset all displays, select 1 Exit then 0 Stop. Type RUN to restart.

ENTERING A PROGRAM IN TEST MODE

If you know machine code you can type in a short program in the TEST routine. Type in ld ix,45000 ld (ix),205 ld (ix+1),100 ld (ix+2),234 and note that the "Next" instruction gives call 60004. Then type ld (ix+3),201 (return) ld (ix+4),43 (dec hl) ld (ix+5),201 ld a,201 ld (60004),a (puts return at 60004). Next select 2 and step through the program, noting that it calls 60004 and returns.

RELATIVE ADDRESSING

Following entering the above program, select 9 and enter address 60000 then option 13 to change the real address from "YES" to "NO": this resets all displays and the next instruction will show call 60004 at address 60000. Select 2 and step through the program, noting that it executes the dec hl instruction that was inserted at address 45004.

The above demonstrates the relative addressing capability where code can be loaded to address 45000 onwards and tested as though it was at another address. One exception is where programs being tested refer to an address below 16384, in the ROM, where no adjustment is made and the real address used.

The address offset calculation is carried out when real address "NO" is selected, assuming that whatever is at the selected start address is really at address 45000. Once the offset is calculated, the start address can be changed for testing to start anywhere within the code.

LOADING CODE TO TEST (AND CRASHING)

To load code, select 1 Exit, 3 Load code and menu selections according to whether the code is on tape, drive or RAM disk. As an experiment, load the assembler machine code "ass.bin", selecting address 45000 when prompted. This code is normally loaded to address 61300 so select this as the start address, then 13 for real address "NO" (or to "YES" and back to "NO"). Then select 2 and step through the subroutine call. If it is wished to demonstrate a system crash, continue stepping, noting that the bc register is loaded with a value greater than 60000. This is used as the length of the ldir copy instruction and executing this will move most memory contents up one location. If this is done, the system will have to be reset and software reloaded.

LARGER MACHINE CODE PROGRAMS

Code for testing can be loaded below 45000, down to 40950, that is about 24000 bytes. However, this disables the direct instruction input capability.

To use the relative addressing facility, it is necessary to calculate the offset from 45000. If code is loaded to 40950, the offset is 45000 - 40950 or 4050. As an example, if the real code address was 30000, select start address and input 34050 (real address + offset), then 13 for real address "NO". Selecting a new start address of 30000 will display "Next" as the first instruction loaded. For larger programs see DISASSEMBLING GAMES PROGRAMS.

STEP CALLS

Normally, when a subroutine is called, it is stepped or traced as the main code. When testing code is loaded to real addresses, subroutines can be executed at full processor speed. To demonstrate, select 14 to change step calls to "NO" then, as for the earlier demonstration, input start address 16 and 2 to step through the code: this time the end will be reached after 12 steps, without stepping through the call. Similarly, instructions typed in directly will execute at the full rate: try call 16. Select 14 to change to step calls "YES". If relative addressing is used, any subroutine called must not call others as the address cannot be calculated.

ERRORS

The error display appears in the middle of the screen under the memory display. Besides giving an indication when breakpoint or end address is reached, the main message on testing is "INVALID OP". Assembler errors are covered later. Certain instructions will not be executed e.g. halt, ld sp,NN, reti, im N or ret and pop, if the stack is empty. The software will not execute a conditional return if the stack is empty, even though the return is not appropriate. To overcome this, the address of a return instruction can be put on the stack (e.g. type in instructions ld hl,82 and push hl). An example to demonstrate this is a ROM routine which finds the start address of any BASIC line - select start address H196E then type in ld hl,20 (the line number), then select "Run slow". The program will stop with both hl and de registers pointing to the start address of the first line. Reselect the start address, put 82 onto the stack and 20 into hl, then repeat the trace. This time, de points to the start of line 10 and hl to the start of line 20.

PRINTING

If a printer is available, select 15 to change print to "YES" and repeat the step address 16 procedure with step calls "NO". The state of the registers etc. should be printed after each step. Repeat with run fast and printing should occur when the trace stops.

If the printing does not work see CONVERSION as it may need initialising or different print statements.

DISASSEMBLY

To disassemble from the test screen select the start address (e.g. 0 for the start of ROM), then enter 9 again for start address and keep pressing ENTER to display sequential addresses and instructions as "Next" and "Was". To end this, enter an appropriate start address.

For a listing type disassembly, select the start and end addresses (e.g. 0 and 20), then 1 Exit and 2 Disassemble. If an end address is not selected, following the scroll indication, rather than pressing ENTER for the next screen, press BREAK or the space bar and type RUN.

To print the disassembly, select 15 Print before exit to the second menu. If hexadecimal format is preferred, select 7 Dec/Hex also.

DISASSEMBLING OTHER ROMs

Other ROMs can be disassembled by using the save routines provided in the ROMs. For example, on the Spectrum +2, exit from the test and select STOP. Then type save ! "rom" code 0,16384 then RUN. Select 1 Exit, 3 Load code, 3 RAM disk, name "rom", address 45000. Similarly, the microdrive Interface 1 ROM can be saved by save *"m";1;"rom" code 0,8192 (or one of the +3 ROMs by save "m:rom" code 0,16384) and loaded to 45000. The ROM can then be disassembled from address 45000 but note that relative addressing cannot be used to display the real addresses, as they are below 16384.

DISASSEMBLING GAMES PROGRAMS

Breaking into games programs is a science in its own right and the author of this software is not an expert on the subject. However, some experiments have been carried out to provide a starting point. Anyone attempting this will at least need to absorb the detail given in the Spectrum manuals on memory, system variables and using machine code. Following the detail of the demonstration of finding BASIC line numbers (see ERRORS) will also help.

The games start by loading a BASIC program, usually quite short. These usually set RAMTOP (using CLEAR address), load code above this address, often preceded by loading a screen by load SCREEN$ or load code to 16384. This may then be followed by a USR address statement to start running the machine code. The first steps for disassembly are to find the RAMTOP and USR addresses and to load the code without executing the USR statement. Sometimes it may be possible to load the BASIC program and stop the tape before the following load is started and delete the USR statement so that the system will stop after loading the code. In many cases the BASIC loaders are written to prevent them from being listed, machine code may be embedded to control non-standard tape loading, system variables poked to make the system crash and so on. Example programs, which will allow these BASIC loaders to be cracked are given later.

Once the code is loaded, it can be saved in two or more parts for disassembly later. The code (or data) usually starts at around address 24000, so two parts could be, say, 24000 to 48004 and 48000 (leave some overlap) to 65535. These can then be loaded in turn by the TEST program.

ASSEMBLER FACILITIES

Assembler input codes are entered in BASIC lines 1 to 2000, with a REM statement at the start, using the normal Spectrum editing facilities. The codes are entered in lower case format as given in Spectrum manuals. Multiple instructions can be entered in a line with semi-colons between them. The codes must be exactly the correct format without extra spaces, or an error message will be given and the codes not assembled. The lines can be typed in with or without the assembler being loaded.

One byte variables, N or DIS, and 2 byte variables, NN, can be positive (no sign) decimal or hexadecimal numbers or characters. The offset, D, in certain ix or iy instructions can be decimal or hexadecimal, in the range -128 to +127. Hexadecimal numbers have the prefix H and letters must be capitals. Characters have a prefix C.

Load the assembler, select 0 to stop and enter the following lines:

```
10 REM ld b,254;ld a,CX;ld hl,Cab;ld de,0;ld ix,20480
20 REM ld b,HFE;ld a,88;ld hl,H6162;ld de,H0;ld ix,H5000
30 REM ld a,(ix-128);ld (ix+127),H2A;ld (iy-H10),Cn
```

Rather than using real addresses for call and jump instructions or a one byte variable for jr DIS and djnz DIS (e.g. call 54200;jp 55123;jr z,34;djnz 250), a BASIC line number can be used as a label with a prefix L. The assembler determines the address or, in the case of DIS, the displacement values (-126 to +129 from the instruction start address). Similarly, a line number can be used as a NN variable. Add the following to the lines typed in:

```
40 REM call L10;jp nz,L20;jr nc,L30;djnz L40;ld hl,L10;jp (hl)
```

The latter give the same effect as jp L10.

Besides standard Z80 instructions, a number of additional ones are provided to preset variables in memory. These are defb to define a byte, defl to define a byte in binary, defw to define a word, defc to define a string of up to 255 characters and defs to define a number of bytes of memory, initially zeroised. Note defb, l and w can be typed in during TEST to display the value for conversion purposes. Enter:

```
50 REM defb 65;defb Ca;defw 1234;defw HFFFF;defc "String"
60 REM defs 100;defl 10101010
```

A further function, defL, is available for defining memory addresses to be referenced by a machine code program. These can be used for various purposes and have a format as shown in the following examples. The first example gives an unused line number label the same address as an existing line to enable successful assembly or to allow the code to be tested before code for the new line is written. The second two examples can be used to define frequently used addresses which may be changed, for example when a long program is assembled as a number of parts.

```
70 REM defL100 L10;defL1 45328;defL2 HA6FE
```

It is usual to include the defL statements at the start of program, defining lines which are otherwise unused.

The advantage of using line number labels is that it is easy to find the code referred to. The disadvantage is that lines cannot be renumbered without checking whether there is a reference to them. Named labels can be used instead of or as well as line number labels, the advantages being that line renumbering is easier and, along with comments, they provide built in documentation.

Named labels and comments must start with a capital letter and can be up to 9 and 15 characters respectively. If they are greater than this, only the first 15 will be displayed. Labels must be placed as the first entry on a line, otherwise they will be treated as comments. They can be on a line with no other instructions, otherwise separated by a semi-colon.

References to named labels must be preceded by a @ symbol. Examples are:

```
 80 REM Memory;defs 250;Output area
 90 REM Count 1;defw 250
100 REM START
110 REM ld hl,@Memory;ld bc,(@Count 1)
120 REM Loop;ld (hl),0;inc hl;dec bc;ld a,b;or c;jr nz,@Loop;Zeroise
130 REM This is a valid line for documentation purposes.
```

ASSEMBLY

Assuming the assembler is loaded for the example lines to be typed in, enter RUN then select 1 Assemble from the main menu. To check that the typing is correct, select 6 Start assembly. The address 53000 will flash at right of the chosen entry. Press ENTER to continue or the space bar and ENTER to cancel the request. If all are correct, two passes of the assembler will be executed with the addresses and line numbers being displayed. If an error is found, an automatic listing is given from that point. For a full listing, before assembly, select 3 List on assembly, then 1 to cancel, 2 to display or 3 to print.

A selection of lines can be assembled by changing the first and last line numbers (options 1 and 2). The program will not allow a second line number to be selected which is less than the first.

LABEL ADDRESSES

Following assembly, label and line number addresses can be displayed or printed via assemble option 4. The label information stored is the start address of each line used. Named labels are not stored but are picked up from the input code lines as the display is given. The addresses are stored until a new start assembly option is chosen but, if the input program has been deleted, the label names will not be displayed.

ASSEMBLER RELATIVE ADDRESSING

The code is always assembled to address 53000 onwards but the start address can be selected according to where the code is finally to be loaded. The assembler calculates label addresses, used by calls and jumps, and code addresses for listing as the final ones. This can be demonstrated by selecting different start addresses, assembling with listing and displaying label addresses.

ASSEMBLING LARGE PROGRAMS

Programs, where the input lines are in two or more parts, can be assembled and combined with the second or subsequent parts calling or jumping to code in earlier parts, providing different line numbers are used for each part or, at least, those referenced are unique.

After the first part has been assembled, the continue address is calculated as the starting point for the next code: this can be changed, if required, by selecting option 7 and entering the new address. Selecting 8 continue assembly will assemble the next part, calculate the new label addresses and pick up references to lines already assembled. Where only line number labels are used for the cross references, the procedure is automatic.

As named labels are not stored forward, they can be redefined at the start of subsequent parts: for example, a routine called FIND at line 1200 in part 1 can be referred to as 10 REM FIND;defL10 L1200. If 1200 REM FIND were used again, the address would be redefined with a wrong value.

The ROYBOT assembler is assembled in two parts and has 8 named label references from part 2 to part 1. The following demonstrate how it was assembled, as an indication of how to deal with large programs:

1) Merge "ass11", select assemble, start address 61300, select start assembly, print label addresses (for reference by BASIC), select save code, save as code1 then delete lines 1 to 2000.

2) Merge "ass12", select assemble, note continue address, select continue assembly, print labels for second part, save code as code2.

3) Select stop, manually load "code1" 61300 and load "code2" to continue address then save "ass.bin" 61300,4236.

INPUT LINES

Lines 1 to 2000 are reserved for the REM input lines and 2001 to 5000 for the BASIC driver. Lines greater than 5000 can be used for other purposes, such as a short BASIC program for testing machine code just assembled.

MERGE LINES AND DELETE

The input lines can be entered without the assembler being loaded and incorporated later by selecting option 3 merge lines. After assembly, some or all of these can be deleted by selecting 5 delete, then 3 delete selected lines. If it has been necessary to modify the input lines, different options are available for saving them.

If it is decided to keep the input lines as separate independent programs, select 5 delete then 4 ROYBOT lines. The ROYBOT BASIC will be deleted and the system will stop with a "Nonsense in BASIC" message. Insert a new line (>5000) to save and verify the lines and, initially, to erase the original, if necessary. Then type RUN to save the program. The new line will be included in the save and on merging, loaded for future use. DO NOT SAVE WITHOUT ENTERING RUN OR CLEAR OR THE SYSTEM WILL CRASH ON MERGING.

SAVE AND LOAD

One of the save options is to save all BASIC lines. This can be used as an alternative approach, saving the assembler BASIC and input lines together. If a long input program is produced, it will be observed that merging is very slow. When the save all BASIC option is used, the save will incorporate a restart line of 2010, the start of the assembler. To reload and automatically run the software, after the assembler and machine code has been loaded initially, select 2 load new and input the name of the combined assembly BASIC and input lines. This method is much faster than using merge if microdrives, disks or silicon disks are available.

The other save options are to save the assembled code from address 53000 and the ROYBOT code and BASIC (see COPYING THE SOFTWARE). Load new can also be used to load the TEST software "test.bas".

UTILITIES

Options 6 and 7 from the assembler main menu are provided for producing catalogs and erasing files from disks, microdrives or RAM disks as it has been found that these functions are used frequently when new machine code programs are being developed. For details of option 8 initialise, see CONVERSION.

128K SYSTEMS

When using 128K systems, particularly those relying upon tape input, the software and input lines can be loaded and saved temporarily in the RAM disk.

ASSEMBLER ERRORS

The following errors are detected and indicated on assembly:

1 Syntax error - wrong characters typed, extra spaces etc.
2 Number out of range e.g. >255 for a 1 byte variable
3 Distance for jr instructions too far or invalid (jump to itself)
4 Line number or named label not found
5 D value in ix or iy+D too large
6 Label name used previously

Certain Spectrum systems will hang if adding BASIC program lines attempts to cause the available memory space to be exceeded. The software checks for this and issue a warning message "too many input lines". If this occurs, the program should be split into two. On assembly, the software also checks that the machine code will not be assembled to a real address greater than 57249: if this is attempted, the system will stop indicating "Out of memory". Assembling defs 4250 will demonstrate this.

If a printer does not work, refer to the section CONVERSION.

AUTO-LOADER PROGRAM

The auto-loader program, supplied for disk and microdrive versions, provides menu selection for loading the assembler or test software, copying the software and designing user defined characters.

On loading, the assembler code, containing predefined special characters, is also loaded. These will be in memory when TEST is selected. When character design is selected, the special characters are displayed along with a to s, which are used for selection purposes. Entering a letter displays a large version of the character and a menu with 0 Exit and:

1 Design - a cursor can be moved around and dots made white or black.

2 a normal or graph shift character can be displayed for comparison.

3 transfers the character (even blank) in place of the selected one.

If anything is changed another menu is displayed, where 0 Cancel rereads the original code, 1 allows another character to be selected and 2 resaves "ass;bin" with the changes included.

TEACH YOURSELF MACHINE CODE ON-LINE

Load TEST as described earlier and read sections "THE PROCESSOR" and "TESTING FACILITIES" before starting the following exercises.

REGISTER LOADING

The first group of instructions to consider are those which load registers with a constant. These are of the general format ld r,N where N is between 0 and 255, and ld rr,NN where NN is between 0 and 65535. Type in the following instructions in turn, pressing ENTER to cause them to execute. Note that, as the instructions are entered, the code appears in the memory display e.g. 62 1 for the first one:

```
ld a,1  ld b,2  ld c,3  ld d,100  ld e,75  ld h,254  ld l,255
```

Note that the value in the bc register pair is 256*b+c or 515 and similarly for de and hl. To confirm this enter the following. It can be observed that the NN value in the machine code is the opposite way round to that in the registers, also that the ix and iy loads are the same as hl, except being preceded by 221 or 253:

```
ld bc,515  ld de,25675  ld hl,65279  ld ix,65279  ld iy,65279
```

The next group of 49 instructions copy the values of any one of the single letter registers a,b,c,d,e,h,l to any others and itself. Try:

```
ld a,a ld a,b ld a,c ld a,d ld a,e ld a,h ld a,l ld b,a ld c,d ld h,l etc.
```

There is no direct equivalent for copying 2 letter registers but, for bc, de and hl, this can be achieved by two loads e.g. ld h,d and ld l,e. For another method, particularly for ix and iy, see STACK PUSH AND POP.

LOADING MEMORY CONTENTS

All double length registers can be interchanged with 2 adjacent memory bytes. Note that the memory display starts at address 35923 and, with instructions being 1 to 4 bytes long, addresses 35927 to 35930 can be used to demonstrate memory transfers. In the following examples note that the numbers in memory are again in the reverse order:

```
ld hl,1234  ld (35927),hl  ld de,(35927)  ld ix,(35927)  ld (35929),iy
```

WARNING - these instructions can easily overwrite important memory addresses, such as System Variables, and cause a crash.

The only single byte register that can be used in this way is a:

```
ld a,255  ld (35927),a  ld a,(35928)
```

INDIRECT ADDRESSING

Rather than using a number for loading memory contents, the address can be loaded into registers which are used for indirect loading. The hl register can be used in conjunction with a 1 byte number or any one of registers a,b,c,d,e,h,l. The following demonstrate indirect addressing of the attributes of the first character on the screen:

```
ld hl,22528  ld c,(hl)  ld a,79  ld (hl),a  ld (hl),249  ld (hl),c
```

The bc and de registers can be used, but only in conjunction with a:

```
ld bc,22528  ld de,22529  ld a,(bc)  ld (de),a  ld a,79  ld (de),a
```

INDEXED ADDRESSING

Indexed addressing is similar to indirect addressing but uses the ix and iy registers with a displacement in the range -128 to +127. The facilities available are identical to those for indirect addressing with hl and, except for an initial byte of 221 or 253, the machine code is the same. The first of the following examples again deals with attributes and the second loads System Variable RAMTOP into hl:

```
ld ix,22656  ld b,(ix-128)  ld (ix-128),249  ld (ix-128),b  ld a,(ix)
ld iy,23610  ld l,(iy+120)  ld h,(iy+121)  - Note reverse order
```

STACK PUSH AND POP

A push instruction stores a 2 byte register on the stack by subtracting 1 from the stack pointer SP, storing the first byte, resubtracting 1 then storing the second byte. The instructions can be used for copying from one register to another or for temporary storage purposes. Note the last in first out effects:

```
push bc  pop af  push iy  push hl  push de  pop hl  pop de  pop ix
```

The Spectrum software uses a number of stacks and, as in the ROYBOT software, private stacks can be created by saving the stack pointer (e.g. ld (35927),sp) then loading sp in various ways. The latter are not implemented in the TEST software (sp to memory is) as they would cause a crash, but they are of the following general format:

```
ld sp,hl  ld sp,ix  ld sp,iy  ld sp,NN  ld sp,(NN)  ld (NN),sp
```

The stack is another dangerous area to play with - push or pop too many and something will be overwritten: pop too few and a crash is likely (see also JUMP, CALL AND RETURN).

EXCHANGE REGISTERS

The CPU chip has a second set of af,bc,de,hl registers which can be exchanged. Some may be used by Spectrum ROM routines so it may not be a good idea to use them. They are implemented in the ROYBOT software by storing the values in memory.

```
ld a,123  ex af,af'  ld a,255  ex af,af'  ld de,12345  ld hl,9999
ld bc,1000  exx  ld hl,0  ld de,0  exx  exx  exx
```

Another instruction allows the hl and de registers to be exchanged and others exchange the contents of hl, ix or iy with the two bytes at the top of the stack:

```
ex de,hl  push de  ex (sp),hl  ex (sp),ix  ex (sp),iy  pop iy
```

I AND R REGISTERS

There are two other registers that can only be used in conjunction with the a register. These are the interrupt page register i and the memory refresh register r. Loading values into these registers should be avoided. The instructions are - ld a,i ld a,r ld i,a ld r,a

BLOCK TRANSFER AND SEARCH

These are some of the most powerful and useful instructions available. Any memory area, within the 64K addressable range, can be copied to any other as a block transfer. The hl register has to be loaded with the source address, de with the destination and bc with the number of bytes to copy.

There are two ways of copying blocks of memory to ensure that data is not overwritten when the blocks overlap. To copy a block to a lower address it is necessary to start at the bottom, copy the first byte then increment the addresses (ldir). To copy a block to a higher address, start at the top, copy the last byte then decrement the addresses (lddr). The first example copies rubbish to the screen (watch the screen on pressing ENTER for ldir). The second moves the attributes of the top line. Note: with bc too large or Ø or addresses being wrong, the system will crash. The third example uses the single byte copy instructions ldi and ldd, moving the instruction code and first 3 attribute bytes into the memory display.

```
ld hl,Ø  ld de,16384  ld bc,6144  ldir  ld de,22527  ld bc,6144  lddr
ld hl,22529  ld de,22528  ld a,(de)  push af  ld bc,32  ldir  ld e,31
ld l,3Ø  ld bc,31  lddr  pop af  ld (de),a
ld hl,35923  ld de,35927  ldi  ldi  ld hl,2253Ø  ldd  ldd  ldd
```

For block search instructions hl is loaded with the start address, bc with the number of bytes to search and a with the character to be found. The search stops when bc reaches Ø or a match is found. For the latter, the z flag will be switched on and hl points to the address following the match. The single byte searches cpi and cpd compare a byte and increment hl:

```
ld a,Cr  ld hl,Ø  ld bc,2  cpir  cpir  cpdr  cpi  ld e,(de)  cpi  cpd  cpd
```

INPUT OF CHARACTERS, HEXADECIMAL AND BINARY NUMBERS

As used in the previous example, characters can be loaded to registers using the prefix capital C. In order to understand many of the other instructions, a knowledge of hexadecimal and binary is necessary and the appropriate sections in the Spectrum manual should be studied. Hex numbers can be loaded using the prefix H (e.g. ld a,HFF) or addresses in the same format. As described under TESTING FACILITIES, selecting option 7 switches between decimal and hex format for addresses and register contents. Also, selecting option 8 enables binary values to be displayed.

Hexadecimal and binary conversions can also be carried out using defb, defw and defl functions by direct input (see ASSEMBLER FACILITIES). The hexadecimal digits declared can be 1 or 2 for defb and 1 to 4 for defw. Binary defl declarations must always be 8 bits. Examples of def functions displaying numbers in the memory display are:

```
defb HA  defb 1Ø  defb HAB  defw H1DE  defw H9A8F  defl 1111ØØØØ
```

EIGHT BIT ARITHMETIC AND LOGIC

Most of these instructions are in association with the a register (accumulator) and involve a variable, other registers, indirect addressing or indexed addressing. The first group of instructions are add, add with carry, subtract and subtract with carry. The carry flag is switched on where an add gives a value greater than 255 or a subtract gives a negative result. Load appropriate values in register and memory locations and input some of the following:

add a,a add a,b add a,c add a,d add a,e add a,h add a,l add a,(hl)
add a,(ix+1) add a,(iy-123) add a,23 adc a,a adc a,b adc a,99 etc.
sub a sub b sub c sub d sub e sub h sub l sub (hl)
sub (ix+100) sub (iy+45) sub 77 sbc a,a sbc a,b sbc a,5 etc.

The next group of instructions are comparisons with the a register. Where the value being compared is equal to that in a, the zero flag z is set and, where it is greater, the carry flag c is set.

cp a cp b cp c cp d cp e cp h cp l cp (hl) cp (ix+46) cp 255

Instructions are available for adding 1 (increment) or subtracting 1 (decrement) from any register or memory location (be careful to only change known locations):

inc a inc b inc c inc d inc e inc h inc l inc (hl) inc (ix+1)
dec a dec b dec c dec d dec e dec h dec l dec (hl) dec (ix+1)

Three logic instructions, AND, OR and XOR are available, operating on binary patterns in the a register. The results of carrying out the operations on each individual bit are:

0 AND 0 = 0 0 OR 0 = 0 0 XOR 0 = 0 0 AND 1 = 0 0 OR 1 = 1 0 XOR 1 = 1
1 AND 1 = 1 1 OR 1 = 1 1 XOR 1 = 0 1 AND 0 = 0 1 OR 0 = 1 1 XOR 0 = 1

AND can be used for selecting bits, OR for combining and XOR for inverting. Load various numbers into registers via the a register, such at 170, 15, 240 and 255, noting the binary patterns. Then carry out a selection of the following instructions.

and a and b and c and d and e and h and l and (hl) and (iy-22)
or a or b to or (iy+100) xor a xor b to xor (iy-120) etc.

An important function of these instructions is that they clear the carry flag; "and a" and "or a" leave the a register unchanged; "xor a" will also zeroise the a register. The or function is often used for checking a double byte register for zero.

FLAGS

The flags can be all switched on or off through using push/pop (ld e,255 push de pop af ld e,0 push de pop af). The bits in f are:

0 Carry flag (c) - see subtract above.

1 Add/subtract (n) - this is switched off for adds and on for subtracts and is used with decimal arithmetic.

2 Parity/overflow (p) - this is set by and, or, xor, if the number of bits is even and reset if odd. It is also used to indicate overflow when dealing with positive numbers (see s flag) where carry is not set (try - ld a,128 and a (no parity) add a,64 and a (parity) ld a,127 and a (reset p) add a,1 (overflow).

4 Half carry (h) - this is similar to carry except it is associated with with the lower 4 bits. It can be set by "and" and reset by "or". The main use is in decimal arithmetic (see daa).

6 Zero (z) - this is switched on by an operation giving a result of zero or an equal comparison. As other flags, it is not changed by loads.

7 Sign (s) - rather than using a byte to count up to 255 it can be treated as containing positive numbers 0 to 127 or negative numbers -1 to -128, where -1 is 255 and -128 is 128 (2's complement). The sign bit is switched on when an operation sets bit 7 (e.g. 128) - try ld a,0 add a,1 sub 129 add a,127 inc a.

Bits 3 and 5 are indeterminate.

GENERAL PURPOSE OPERATIONS

Decimal Adjust Accumulator (daa) - this is used in conjunction with binary coded decimal adds and subtracts and uses the n and h flags. It is difficult to understand but essentially assumes that all numbers are decimal e.g. 117 or H75 is decimal 75. Try ld a,H75 add a,H16 daa (answer H91) add a,H10 daa (answer 1 + carry for 101) sub H33 daa (answer H68).

Complement Accumulator (cpl) - this inverts the bits in the a register. Try ld a,1 cpl (answer HFE or 254).

Negate Accumulator (neg) - this gives the 2's complement value (see sign flag). Try ld a,1 neg (answer HFF or 255 or -1).

Complement Carry Flag (ccf) and Set Carry Flag (ccf) - note that there is no clear carry instruction but this is done with "and", "or" or "xor". Try scf and a ccf ccf ccf.

SIXTEEN BIT ARITHMETIC

These have the same operation codes as for 8 bit instructions but there is a more limited range. The codes are:

```
add hl,bc  add hl,de  add hl,hl  add hl,sp  add ix,bc  add ix,de
add ix,ix  add ix,sp  add iy,bc  add iy,de  add iy,iy  add iy,sp
adc hl,bc (or de, hl, sp)  sbc hl,bc (or de, hl, sp)
inc bc  inc de  inc hl  inc ix  inc iy  inc sp  dec bc etc.
```

Note two inc sp instructions drop a word from the top of the stack but one gives invalid displays on TEST. The add sp instructions can be useful for obtaining a copy of a word on the stack: push 3 different values and copy the third to de by - ld hl,4 add hl,sp ld e,(hl) inc hl ld d,(hl)

JUMP, CALL AND RETURN

There are two kinds of jump instructions jr (relative) and jp (absolute). The first has a 1 byte displacement (DIS), which is forward 0 to 127 or backwards -1 to -128 (255 to 128) from the start of the next instruction. The second kind have a 2 byte address (NN). When relative jumps are used the machine code can be relocated anywhere in memory but has the disadvantages that the jump distances are not very great and lead to many assembly errors. As shown in ASSEMBLER FACILITIES, programs are written referring to line number or named labels. The instructions can be typed in under TEST (e.g. jr 127 jr z,128 - note addresses) but do nothing other than display the codes in the memory display.

Jumps can be unconditional (as BASIC GOTO) or conditional on the state of one of the flags. The instructions are:

jr DIS jr c,DIS jr nc,DIS (no carry) jr z,DIS jr nz,DIS (non zero)
jp NN jp c,NN jp nc,NN jp z,NN jp nz,NN jp pe,NN (parity even)
jp po,NN (parity odd) jp m,NN (sign negative) jp p,NN (sign positive)

Next there are three indirect jumps jp (hl), jp (ix) and jp (iy), where the register holds the address to jump to (e.g. ld hl,60000 jp (hl) is the same as jp 60000).

A special instruction, decrement b register and relative jump non zero (djnz DIS) is provided for loop control. It does the same as dec a and jr nz,DIS. Enter ld b,0 djnz 123 noting b goes to 255 (loop count 256).

The next group are call and ret, for subroutines, which are the same as BASIC GOSUB and RETURN. The instructions can be unconditional or with the same conditions as jp instructions. The call instructions are 3 bytes long and the following address is pushed onto the stack for the return. The following will demonstrate this - call 10 ld a,1 and a call nz,20 (called) call z,30 (not called) ret (pop) ret z (no pop) ret nz (pop)

Note that the return addresses on the stack prevent other items from being popped in a subroutine. However, the return address can be popped into a register temporarily. The instructions are:

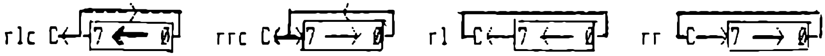
call NN call c,NN call nc,NN call z,NN call nz,NN also pe, po, m, p
ret ret c ret nc ret z ret nz also pe, po, m, p

The final two instructions in this group are return from interrupt or non-maskable interrupt - reti retn

ROTATES AND SHIFTS

There are again a large number of instructions which are used for rotating and shifting bits in any register or memory location. They can be used for multiply or divide (by 2, 4 etc.), moving data from one byte to the next one bit at a time and for counting or checking bits within a loop. The following diagrams indicate the bit flow and can be demonstrated by using those associated with the a register and observing the binary values and carry flag. They are of the following general format:

op a op b op c op d op e op h op l op (hl) op (ix+D) op (iy+D)

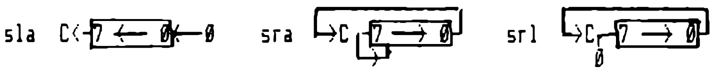


Rotate Left and Right circular rlc and rrc - these circulate the bits around a byte setting the carry flag as a 1 bit is moved from one end to the other. An example is rotating the first attribute byte to see which bits are set - ld hl,22528 repeat rrc (hl) 8 times.

Rotate Left or Right rl or rr - these rotate through the carry bit, requiring 9 steps to return to the original value. An example of use can be seen by disassembling address 1480 to 1496, which is used for reading 8 bits from a tape. Register l is set to 1 so, within the loop, carry is set on the eighth rotate left. Before rotate, a compare sets the carry flag when a 1 bit is read and this is moved into the register with the rotate.

Rotate "a" register - instructions rlc a, rrc a, rl a and rr a are available in a different form which is quicker and only 1 byte long. They are - rlca rrca rla rra

The shift instructions are as follows:



Shift Left Arithmetic sla - the bits move left into carry, fill with zeros and can be used for multiplying by 2, 4 etc. over 1 byte or more with rl - ld c,1 sla c sla c sla c ld de,175 sla e rl d sla e rl d

Shift Right Arithmetic sra and Shift Right Logical srl - these move the bits right into carry, srl filling with zeros and sra with zeros, if bit 7 is "0" and with ones if bit 7 is "1". These instructions can be used for division. With bit 7 set the number can be regarded as negative (see Sign Flag) so sra can divide these. e.g. ld a,128 (-128) sra a (192 or -64).

Rotate Digit Left rld and Rotate Digit Right rrd - are for use with binary coded decimal and rotate a 4 bit digit in the a register with 2 digits at a memory location defined by hl.

These can be demonstrated by using the memory display. Enter ld hl,35927 ld (hl),1 ld a,H28 rld (a=H20 (hl)=H18) rld (a=H21 (hl)=H80).

BIT MANIPULATION

These represent the largest group of instructions (240 in all). They allow each bit of single byte registers or any memory location to be switched to "1" (set), reset to "0" (res) or tested (bit). A major use is for flags where 8 different conditions can be recorded in a byte. The test instructions set the zero flag if a particular bit is zero. The codes are:

set 0,a set 0,b to set 0,l set 0,(hl) set 0,(ix+D) set 0,(iy+d)
set 1 to set 7 res 0 to res 7 bit 0 to bit 7

The Spectrum ROM Software makes frequent use of these instructions in manipulating System Variables e.g. (iy+0) to (iy+3). Examples to try are - ld h,0 bit 5,h (z on) set 5,h bit 5,h (z off) ld ix,22528 (attributes) set 7,(ix+1) to flash res 7,(ix+1) for flash off.

RESTART GROUP

A set of 8 special instructions are available with the CPU chip which call subroutines at addresses 0, 8, 16, 24, 32, 40, 48 and 56. These have to be programmed for the functions required on a particular computer. In the 48K Spectrum ROM they are:

```
 rst 0  - causes a "NEW"
 rst 8  - stops the program with an error code (see Spectrum Manual).
          rst 8;defb 0 gives error 1, rst 8;defb 10 error B etc.
 rst 16 - displays the character in the a register (see TESTING examples).
 rst 24 - associated with scanning a BASIC line - fetch (CH-ADD) to "a".
 rst 32 - scanning again but increment (CH-ADD) first.
 rst 40 - used with floating point calculator.
 rst 48 - creates workspace.
 rst 56 - maskable interrupt routine called 50 times per second to scan
         the keyboard and increment the frame counter (see im 1).
```

INPUT AND OUTPUT

Input and output instructions are the same as BASIC IN and OUT so the appropriate chapters of the Spectrum manual should be studied. The port address is defined by a one byte variable N (data bits by a register) or the c register (data bits b register). Example programs, given later, show how some of the instructions can be used. The codes are:

```
   in a,(N)  in a,(c)  to  in l,(c)  out (N),a  out (c),a  to  out (c),l
```

Block input/output is provided, similar to block transfer and search. Register hl gives the data address, c the port and b the number of bytes to transfer. The codes are ini inir ind indr outi otir outd otdr.

MISCELLANEOUS CPU CONTROL

nop (code 0) does nothing, enabling unwanted codes to be poked with zeros.

halt - suspends operation until the next interrupt.

di ei - disable/enable interrupts, di inhibits normal keyboard scanning.

im 0 im 1 im 2 - interrupt modes. Normal operation is im 1 where rst 56 is executed automatically. Under im 0 external devices can execute instructions via the data bus. For im 2 an indirect call is made to an address defined by the i register and the I/O port.

SUMMARY OF FLAGS

When certain instructions are executed flags may be unchanged so the condition is preserved for later testing:

No Flags changed - ld (except ld a,i or r), 16 bit inc and dec, set, res
                   push, pop and exchange (except af), jump, call, ret

s,z,p unchanged - add hl/ix/iy, rla, rlca etc. (rl, sl etc. do), cpl, scf

c unchanged - inc, dec, rld, rrd, cpl, in, out, ldi, ldd etc., bit

c set to 0 - and, or, xor, to 1 - scf

h set to 0 - or, xor, rla, rl etc. rld, scf, in, ldi, to 1 - and, cpl, bit

n set to 0 - add, adc, and, or, xor, inc, rl, sla etc., scf, ldi,bit

n set to 1 - sub, sbc, dec, cp, neg, cpl, cpi, cpd etc.

TIMING

It is sometimes necessary to be able to determine precise timings in a program such as when writing tape, producing music or moving a large number of objects on the screen without flicker. The specification for the Z80-CPU includes timing in the form of number of clock pulses or T states for each instruction. On the Spectrum the clock is about 3.55 MHz, giving around 0.28 microseconds per T state. The shortest instructions take 4 T states and longest 23. Calculating timings based on T states is not particularly accurate and it is better to do it by program (see example program 1). Where timing is not too critical, an approximation of 2.5 microseconds per instruction can be used (400,000 per second).

EXAMPLE PROGRAMS

The following programs show how most of the various types of instructions can be used and particularly in conjunction with Spectrum facilities. They should be assembled to address 53000.

EXAMPLE 1 - Loops and instruction timing - For single instruction timing a double loop is required. The outer loop to line 20 is controlled by de at 4000 and the inner loop to 30 by b at 250, giving a total of 1 million passes. The system variable FRAMES, which is incremented at 50 times per second, is initially set to 0 and the time returned to BASIC via bc at the end. The program can be run by entering CLS: PRINT USR 53000/50 and will give an answer of about 3.78 (microseconds per loop). A suitable instruction or two for timing (not changing b) should be inserted at line 30, the program reassembled and run via PRINT USR 53000/50-3.78, for the time - ld a,b gives about 1.16 and ld hl,1234 2.88 microseconds.

```
10 REM ld hl,0;ld (23672),hl;ld de,4000
20 REM LOOP1;ld b,250;push de
30 REM LOOP2;Insert instruction to be timed here
40 REM djnz @LOOP2;pop de;dec de;ld a,d;or e;jr nz,@LOOP1
50 REM ld bc,(23672);ret
```

A variation can be used for timing longer activities such as copying a full screen of data (100 times). The time for 1 screen in milliseconds can be obtained by PRINT USR 53000/5, giving about 39.6. This indicates that, with data in the right order, the maximum rate of changing screens is about 25 per second.

```
10 REM ld hl,0;ld (23672),hl;ld a,100
20 REM LP;ld de,16384;ld bc,6144;ldir;dec a;jr nz,@LP;ld bc,(23672);ret
```

EXAMPLE 2 - Screen display and moving object. The example shows one way, with the bare essentials. Five characters are defined at lines 730 to 1170. These are copied to an area of memory defined at line 690 via line 10. If the characters were to be located in the user defined graphics area, line 10 would have ld de,65368 instead of @CHRS1. Lines 20 to 30 copy three of the characters to a memory area PIC1, defining a screen, with each occupying 8 lines. Lines 40 and 50 copy an equal number of bytes to an area ATR1, for attribute ink/paper/bright of white /blue/1 (7+8+64), red/white/0 (2+7*8), green/yellow/0 (4+6*8). 60-70 put the other 2 characters at the start of line 12 in PIC1. 80-110 set d as a delay count, hl as the address of line 12, ix as an area of memory, (ix)/(ix+1) as counters. 120-130 produce the display via 600 with hl, bc and de as defined above. The loop MVST repeatedly moves the object 241 bits across the screen, WAIT governing the speed. After each character bit slice, input at I of 6 to 9 or 0 stops the program.

```
  10 REM ld de,@CHRS1;ld hl,@CHARA 0;ld bc,40;ldir
  20 REM ld hl,@PIC1;ld a,0;call @FILL
  30 REM ld a,4;call @FILL;ld a,1;call @FILL
  40 REM ld hl,@ATR1;ld a,79;call @FILL
  50 REM ld a,58;call @FILL;ld a,52;call @FILL
  60 REM ld hl,@PIC1;ld de,384;add hl,de
  70 REM ld (hl),2;inc hl;ld (hl),3
  80 REM ld d,32
  90 REM START;push de
 100 REM ST2;ld hl,H4880;push hl
 110 REM ld ix,@POS;ld (ix),1;ld (ix+1),241
 120 REM ld hl,@PIC1;ld bc,@CHRS1;ld de,@ATR1
 130 REM call @DISPLAY
 140 REM pop hl
 150 REM MVST;dec (ix+1);jr z,@ST2;push hl;ld b,8
 160 REM MOVE;push hl;srl (hl);inc hl;rr (hl)
 170 REM inc hl;rr (hl);pop hl;inc h
 180 REM ld a,h;cp H50;jr c,@DJM;ld h,H48
 190 REM ld a,l;add a,H20;ld l,a
 200 REM DJM;djnz @MOVE;pop hl;pop de
 210 REM xor a;rl (ix);jr nc,@I;inc hl;ld (ix),1
 220 REM I;ld bc,HEFFE;in a,(c);cpl;and H1F;jr nz,@J
 230 REM WAIT;ld b,d;push de;ld e,0
 240 REM DL1;dec e;jr nz,@DL1;djnz @DL1
 250 REM jr @MVST
 260 REM STOP;ret
 270 REM J;jr @STOP;Temporary line
 580 REM FILL;ld (hl),a;push hl;pop de;inc de
 590 REM ld bc,255;ldir;inc hl;ret
 600 REM DISPLAY;push de;ld de,H4000
 610 REM Eachc;push bc;push hl;ld l,(hl);ld h,0
 620 REM add hl,hl;add hl,hl;add hl,hl;add hl,bc
 630 REM push de;ld b,8
 640 REM Dch;ld a,(hl);ld (de),a;inc hl;inc d
 650 REM djnz @Dch;pop de;inc de;ld a,e;and a
 660 REM jr nz,@Nextc;ld a,d;add a,7;ld d,a;cp H58
 670 REM Nextc;pop hl;inc hl;pop bc;jr nz,@Eachc
 680 REM pop hl;ld bc,768;ldir;ret
 690 REM CHRS1;defs 40
 700 REM PIC1;defs 768
 710 REM ATR1;defs 768
 720 REM POS;defs 4
 730 REM CHARA 0
 740 REM defl 00100110
 750 REM defl 00110111
 760 REM defl 01111111
 770 REM defl 01111111
 780 REM defl 01111110
 790 REM defl 11111100
 800 REM defl 10001000
 810 REM defl 00000000
 820 REM CHARA 1
 830 REM defl 00000000
 840 REM defl 00010000
 850 REM defl 00101000
 860 REM defl 00010000
 870 REM defl 10010100
 880 REM defl 01011000
 890 REM defl 00110000
 900 REM defl 00000000
 910 REM CHARA 2
 920 REM defl 00001111
 930 REM defl 00011011
 940 REM defl 00111011
 950 REM defl 01111111
 960 REM defl 11111111
 970 REM defl 11111111
 980 REM defl 11111111
 990 REM defl 00111000
1000 REM CHARA 3
1010 REM defl 11110000
1020 REM defl 11011000
1030 REM defl 11011100
1040 REM defl 11111110
1050 REM defl 11111111
1060 REM defl 11111111
1070 REM defl 11111111
1080 REM defl 00011100
1090 REM CHARA 4
1100 REM defl 00000000
1110 REM defl 00000000
1120 REM defl 00000000
1130 REM defl 00000000
1140 REM defl 00000000
1150 REM defl 00000000
1160 REM defl 00000000
1170 REM defl 00000000
```

In order to control the screen properly, it is essential that the addressing is understood. The first character in the display memory starts at 16384 or H4000, the second at H4001, third at H4002 etc. The 8 horizontal strips of the first line start at H4000, H4100, H4200 to H4700 and end at H401F to H471F: this is the reason that inc d is used at 640 and inc h at 170. The second line is H4020 - H403F to H4720 - H473F so inc de or inc hl are used to step along characters and to the next line. This continues to the eighth line H40E0 - H40FF to H47E0 - H47FF. Lines 9 to 16 follow a similar pattern with starting with H4800, H4820 to H48E0 and strips H48, 49, 4A to 4F. Lines 17 to 24 are H5000 to H50E0 and strips H50 to H57. Changes after lines 8, 16 and 24 are dealt with at program lines 650 to 670 when de becomes H4100, 4900 or 5100. Attributes start at H5800.

The following additions demonstrate keyboard or joystick operation. Input is obtained via line 220, using port address HEFFE (61438) for keys 6 to 0 or Spectrum joystick 1. Addresses for other keys are given in the Spectrum manual (see IN). At 270, 0 or fire stops the program. Via 280, joystick left or key 6 slows down movement across the screen and right or 7 speeds it up by changing delay d. Down or 8 moves the object down and up or 9 moves it up within the bounds of the middle third of the screen.

```
270 REM J;cp 1;jr z,@STOP;ld c,3;bit 0,(ix);jr z,@J2;ld c,2
290 REM J2:cp 8:jr z.@R:cp 16;jr z,@L;push de;cp 2;jr z,@UP;cp 4;jr z,@DN
310 REM DL2;pop de;jr @WAIT
320 REM L;ld a,32;cp d;jr z,@WAIT;inc d;jr @WAIT
330 REM R;ld a,4;cp d;jr z,@WAIT;dec d;jr @WAIT
340 REM UP;ld a,l;cp H20;jr nc,@UP1;ld a,H48;cp h;jr z,@DL2
360 REM UP1;ld d,h;ld e,l;dec d;ld a,H47;cp d;jr nz,@UP2
370 REM ld a,e;sub H20;ld e,a;ld d,H4F
390 REM UP2;call @COPY;push hl;ld b,7
400 REM UP3;push bc;ld h,d;ld l,e;inc h;ld a,H50;cp h;jr nz,@UP4
410 REM ld a,l;add a,H20;ld l,a;ld h,H48
420 REM UP4;call @COPY;pop bc;djnz @UP3;call @BLANK;pop hl;jr @DL2
480 REM DN;ld a,l;cp HDF;jr nc,@DL2;ld de,H20;add hl,de;ex de,hl;ld b,8
490 REM DN1;push bc;ld h,d;ld l,e;dec h;ld a,H47;cp h;jr nz,@DN2
500 REM ld a,l;sub H20;ld l,a;ld h,H4F
510 REM DN2;call @COPY;pop bc;djnz @DN1;call @BLANK;jr @DL2
530 REM COPY;ld b,c;push hl;push de
540 REM CP1;ld a,(hl);ld (de),a;inc hl;inc de;djnz @CP1;pop hl;pop de;ret
560 REM BLANK;ld b,c;ld a,0
570 REM BL1;ld (de),a;inc de;djnz @BL1;ret
```

Sound can be included in the program by adding the following at WAIT, where the delay varies between about 40 and 5 milliseconds. Flipping the port after each delay gives sound in the range 12.5 to 100Hz. The border can be flashed black/white by changing AND 16 to AND 23 and deleting OR 7.

```
235 REM ld a,(ix+2);xor 255;and 16;or 7;ld (ix+2),a
245 REM out (254),a
```

EXAMPLE 3 - Reading tapes. The following programs can be used to assist in hacking software supplied on tape. They must not be used for illegal copying. The first program has three starting addresses: 53000 reads headers for interpretation by BASIC: 53004 loads BASIC programs and stops without executing any autostart LINE: 53008 loads BASIC as code for later interpretation. Headers are loaded to 54000 and code to 54100. In the program, ix defines the start address for loading and de the length.

All loading starts at ROM address H556 with the carry flag set and a=0 for a header or a=255 in other cases. BASIC and code are via H775 and H800

```
 10 REM ld a,1;jr @Start;Read headers
 20 REM ld a,2;jr @Basic
 30 REM Ldcode;ld a,3;Load as code
 40 REM Start;push af;ld ix,54000;call @Headr;pop af
 50 REM cp 1;ret z;ld a,(ix);cp 0;jr nz,@Ldcode
 60 REM Code;scf;ld d,(ix+12);ld e,(ix+11);ld ix,54100;call H800;ret
 70 REM Basic;ld bc,34;rst 48;push de;pop ix;ld (ix),0;ld (ix+1),255
 80 REM ld hl,(23641);ld de,(23635);scf;sbc hl,de;ld (ix+11),l;ld (ix+12),h
 90 REM push de;ld bc,17;add ix,bc
100 REM HeadB;call @Headr;ld a,(ix);cp 0;jr nz,@HeadB
110 REM ld (ix+14),128;No line no.;ld a,1;ld (23668),a;jp H775
120 REM Headr;push ix;ld de,17;xor a;scf;call H556;pop ix;jr nc,@Headr;ret
```

```
  5 REM the following runs the above to read BASIC and code headers
 10 LET bc=USR 53000: REM Read header, press BREAK to stop
 20 FOR i=54001 TO 54010: PRINT CHR$ PEEK i;: NEXT i: REM name
 30 IF PEEK 54000=0 THEN PRINT " B ";: GO TO 100: REM BASIC
 40 IF PEEK 54000=3 THEN PRINT " C ";:GO TO 70: REM Code
 50 PRINT: GO TO 10: REM 54000=1/2 are no./chara arrays, length 54011/12
 70 PRINT PEEK 54013+256*PEEK 54014;",";PEEK 54011+256*PEEK 54012
 80 GOTO 10: REM Prints start address, length (16384 = SCREEN$)
100 LET l=PEEK 54014: IF l=128 THEN PRINT " None";: GO TO 120
110 PRINT " ";l*256+PEEK 54013;: REM Automatic start line number
120 PRINT " ";PEEK 54011+256*PEEK 54012;" ";PEEK 54015+256*PEEK 54016
130 GO TO 10: REM above give length (PROG) to (E LINE) and (VARS)
```

Many games will give headers and a BASIC loader as shown on the right, loading a screen, attributes and code to fill up the memory. The program is started at line 10 on loading and starts the game with the USR. To copy, change the basic if required, e.g. to load from disk including names in loads and save by SAVE "basic" LINE 0. Change the loader to as it was delete the USR statement (and INKs etc.), add SAVE statements after the LOADs, with appropriate start addresses and lengths, and RUN to load the tape and save where ever. A variation of the above may have loading addresses with LOAD CODE which are different to the headers.

```
basic  B 0 120 120
screen C 16384,6912
game   C 28000,37536
10 CLEAR 27999
20 LOAD "" CODE
30 LOAD "" CODE
40 GO TO USR 28000
```

Most programs set PAPER and INK on starting to make the listing invisible on normal loading. A variation which can prevent the above from giving a proper listing is to include control codes 0 to 31 in the BASIC lines. Type in a program 10 ::LET prog=PEEK 23635+256*PEEK 23636:PRINT prog. RUN to print the start address of the program, then POKE,prog+1,0 to make the line number 0. Listing will appear correct on a 48K system but, on the 128K, the line number will not be given and the line will appear several times. RUN and note that it still works. Then POKE prog+4,17:POKE prog+5,0 to change the two colons to PAPER 0. The listing is then blacked out, although the display may not change on the 128K system. In order to crack these, the program can be loaded and a loop typed in to peek and print from (PROG) to (VARS). In order to understand the format, the Spectrum manual should be studied. The games driver can then be reloaded and offending characters poked with 32 (space) and a suitable line number inserted to give a proper listing.

Headers as on the right indicate that BASIC variables are present. Assuming these are used in the program and the USR address is in the main code, the method used for the previous example may be satisfactory but GO TO should be used instead of RUN to avoid losing the variables. The last code is loaded to the printer buffer area which could cause a problem on 128K systems. If it does, it may be possible to transfer it to the screen memory until loading is finished: e.g. change the USR address to 16384 and add the code at CODE after the following new program which should be loaded to 16384 - ld hl,@CODE;ld de,23333;ld bc,20;ldir;jp 28000

```
basic B 0 234 122
code1 C 28000,37536
code2 C 23333,20
```

Programs with headerless loaders may indicate only a BASIC header and have a program essentially as shown here, indicating machine code in the variables area, or embedded in a BASIC line, with slightly different numbers. The address of (PROG) should be noted and a code file saved starting from this address with length as indicated in the header. This can then be loaded later for disassembly.

```
basic B 10 350 100
10 CLEAR 25000
20 PRINT USR 23950
```

In many cases the code is likely to use normal ROM routines around addresses H556 to H800 (1366-2048) for loading, using ix, a and de as described above, followed by a jump to the start address (or e.g. ld hl,35000;push hl;jp H556 to pick up the start by the return at the end of the loading routine). In this case it is quite easy to produce a new loader to read each section of tape in turn, to enable a copy to be made using normal save with headers.

Other loaders may use non standard code and have no ROM calls. These can be identified by having out (254),a instructions for flashing the border and ld a,127;in a,(254), or the in r,(c) equivalent for loading. Assuming standard saving techniques are to be used, it may not be necessary to determine how the loading works, but the ending procedures and any special variables used will have to be found. The loader code should be modified using BASIC POKEs, direct input with TEST or merging newly assembled code to make it interruptible so the real code can be loaded and copied.

In returning to a BASIC loader for different sections of code being copied, it must be ensured that overwriting does not take place of System Variables, the program and variables area, the stack (normally just below RAMTOP) or the printer buffer on +3 systems. Once in machine code, with no return to BASIC, these areas can be overwritten, if this is done in the original program. It should normally be possible to transfer the final parts of code via the display memory, as shown above.

A particular thing to watch out for in the machine code loaders are instructions which move the stack e.g. ld sp,hl or ld sp,23530. A return to BASIC cannot occur if this is done and these new stacks are often in the print buffer area. The creation of these stacks should be deferred to after the last stage of loading when they may again be handled via the memory display.

The final example is for dealing with long pieces of code e.g. with a header - prog C 23552,41984 indicating loading from the start of System Variables to the end of memory with the start address on a stack within the code loaded or in the printer buffer. Alternatively, a small amount of code may be first loaded to the top of memory and used to control the remaining loading.

This program copies loading code from ROM into the program area. Lines 40 to 48 are as in ROM but not copied due to the jp instruction. Rd1 is called and stops with a tape loading error message if reading is incorrect. The idea is to load code in parts the lengths defined by Len1/2. If Fl is 1 the code is loaded from 28000 to 65535 then to ROM addresses which are not changed. The first part of code can then be saved. With Fl=0 loading of the second part starts at 28000. The code should be assembled for address 27800 and saved. It is driven by the following BASIC program. The example shown is for code starting at 23552. The first part to be saved is up to address 27999 and the second to its real address of 28000. This can be loaded normally, but the first part may need to be via the screen display with code to transfer it added as shown earlier.

```
10 REM ld hl,H556;ld de,@Rd2
12 REM ld bc,116;ldir;call @Rd1
14 REM ret c;rst 8;defb 26
20 REM Rd1;ld ix,28000;ld a,255
22 REM ld de,(@Len1);and a;scf
30 REM Rd2;defs 116
40 REM Rd3;call H5E3;ret nc
42 REM ld a,HCB;cp b;rl l
44 REM ld b,HB0;jp nc @Rd3
46 REM ld a,h;xor l;ld h,a
48 REM ld a,d;or e;jr nz,202
50 REM ld de,(@Len2);ld a,(@Fl)
60 REM cp 0;jr nz,@Part2
62 REM ld ix,28000
70 REM Part2;jp H5A9
80 REM Len1;defw 1
82 REM Len2;defw 1
90 REM Fl;defb 0
```

```
10 LET fl=1: LET a=4448: LET b=37536: LET c=INT (a/256): LET d=INT (b/256)
20 POKE 27984,a-256*c:POKE 27985,c:POKE 27986,b-256*d:POKE 27987,d
30 POKE 27988,fl: PRINT USR 27800: STOP
50 CLEAR 27799: LOAD "rdbin" CODE 27800: STOP
60 REM fl=1 save CODE 28000,a: fl=0 save CODE 28000,b
```

EXAMPLE 4 - 128K RAM bank switching. The Spectrum manual describes memory bank switching. The following shows how it is done in machine code. The BASIC displays 8 screens, poking the number to BANKN and calling START where, for 0,1,3,4,6,7, a different memory bank is switched in and screen contents are copied along with attributes. The normal bank is then switched back in. Calling START2 (LET u=USR 40029) switches the memory banks in and copies the stored contents back to the screen: this is repeated 25 times. Calling START3 (USR 40065) switches the screen display between the normal one and the alternative in bank 7, without switching the bank within the normal addressing range: this is repeated 65536 times in 3.5 seconds. The program should be assembled for address 40000.

```
 10 REM BANKM;defL10 23388
 20 REM PORT1;defL20 32765
 30 REM BANKN;defb 0
 40 REM START;ld a,(@BANKN);cp 8
 50 REM ret nc;call @CHEK;ret z
 60 REM or 16;call @SWITCH
 70 REM ld hl,16384;ld de,49152
 80 REM ld bc,6912;ldir;jr @END
 90 REM START2;ld b,25
100 REM LOOP;push bc;ld a,8
110 REM L1;dec a;push af
120 REM call @CHEK;jr z,@A
130 REM or 16;call @SWITCH
140 REM ld de,16384;ld hl,49152
150 REM ld bc,6912;ldir
160 REM A;pop af;jr nz,@L1
170 REM pop bc;djnz @LOOP;jr @END
180 REM START3;ld de,0
190 REM LOOP2
200 REM ld a,24;call @SWITCH
210 REM ld a,16;call @SWITCH
220 REM dec de;ld a,d;or e
230 REM jr nz,@LOOP2
240 REM END;ld a,16
250 REM SWITCH;di;ld bc,@PORT1
260 REM ld (@BANKM),a;out (c),a
270 REM ei;ret
280 REM CHEK;cp 2;ret z;cp 5;ret
```

```
10 FOR I=0 to 7: CLS: FOR j=1 to 21: PRINT TAB i*3:PAPER 7-i;INK i;"****"
20 NEXT j: POKE 40000,i:LET u=USR 40001:NEXT i: STOP
40 CLEAR 39999:LOAD "ramsw.bin" CODE 40000
```