How to fully use the 1K ZX81

Introduction
The ZX81. We all grew up with it.
In the years that the ZX81 was on sale all kinds of books were released to get the most out of your ZX81. These books all reacted on coding ON the ZX81 itself.

This might have worked in the early years but even in the 1980's many games were not developped on it or for a single system. The books were for the homebrewcoders but not for the profesionals. Now even homebrewcoders have access to larger computers that can be used to code your game for your favourite ZX81. With a powerfull computer at your site you can get the most out of your ZX81. Even in 1K.

BASIC vs machinecode
To get the most out of the 1K ZX81 bytesaving tricks in BASIC were common. This small booklet will not tell you how to save bytes in BASIC. The simple fact why bytesaving in BASIC is not the way to get the most out of your 1K ZX81 is that:

1) BASIC is slow to code, your program will be slow.
2) The bytes saved in BASIC slow down the code even more.
3) BASIC on a ZX81 is a byte intensive language
4) For byte saving code you need to start coding in machinecode.

– BASIC is slow to code, your program will be slow.
For games with not too much interaction BASIC can be used, but a game with more movement or calculations BASIC will be too slow.
BASIC is slow due to the fact that each line is interpreted by the ROM how to exeute the line. Not once, but each time the line is run.

– The bytes saved in BASIC slow down the code even more
The byte saving 'tricks' might give you a few bytes but your code becomes slower LET A=40000 is faster than LET A= VAL “40000”. When you use a lot of byte saving tricks your code will run slower than the speed you want the code to run at. Adding speed to BASIC is almost impossible.

– BASIC on a ZX81 is a byte intensive language
In BASIC on the ZX81 you can only set 1 statement per line.
Each line holds 4 bytes for the linenumber and the linelength.

Then the statement follows. A simple line like
10 LET A=10 already holds 16 bytes in memory
20 LET A=A+1 holds 17 bytes
With just 2 statements we have already used 33 bytes and that is before we have run the code. After running the variable will be held somewhere in memory too at the cost of bytes. So to get the most out of the 1K ZX81 the only conclusion can be:

For byte saving code you need to start coding in machinecode.
The 2 statements from above will give the insight why after more than 40 years it is time to make that step into coding machinecode.

The lines from above
10 LET A=10
20 LET A=A+1

can be coded in machinecode in 3 bytes:

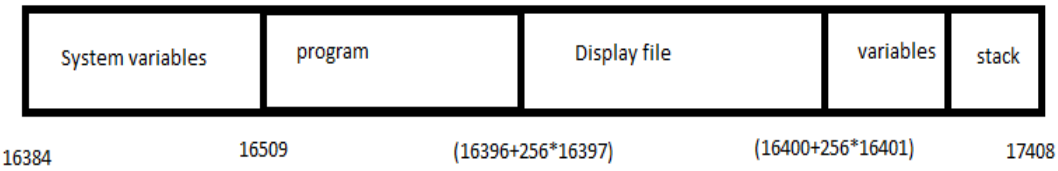
```
LD    A,10   ; 62 10
INC   A      ; 60
```

Behind the instructions the memory values of the instructions are shown.
A compiler will, unlike the BASIC-interpreter, translate the instructions into executable code.

The size is just 3 bytes? Why is that? Both the linenumber and the length of the line are not stored in memory. The address in memory is in fact the linenumber. Second the computer stores the value of A in just 1 byte. For numbers above 256 but smaller than 655536 you only need 2 bytes. BASIC stores a number in a floatingpoint notation. In most games there is no need to use floatingpoint numbers. A score is in points, not in decimals. Even if you would display decimal points or time less than a second other methods can solve this problem.

With machinecode the size of a progam will reduce and will create room to expand your games!

The memory layout of a 1K ZX81
To get the most out of the 1K ZX81 we need to know how the memory of a 1K ZX81 is organized.



Any program for a ZX81 must load a part of the systemvariables and the program itself. Each program also holds a displayfile. Variables are not always saved with a program.

The program starts loading at 16393. The systemvariables between 16384 and 16392 are used by the machine. The same goes for the memory part called the stack. When your program is too large it will load over the stack and loading will fail. On a 1K ZX81 you can load 949 bytes without problems. Any larger program might not load, some will, some don't.

According to the layout a ZX81 program would start at 16509. This would indicate that ANY program would already have 125 bytes lost to the systemvariables. This would be the case if we would code on a ZX81 itself, but to win bytes a model that compiles into executable code can save and (re)use as many bytes as possible.

By using the model the BASIC-editor in the ROM can't be used anymore. This means that after loading a program the program should autorun and go to machinecode without returning to BASIC.
The model will win already 65 bytes from the systemvariables by simply erasing the calculator area and the printerbuffer.

S33	16444	PRBUFF	Printer buffer (33rd character is NEWLINE).
SN30	16477	MEMBOT	Calculator's memory area; used to store numbers that cannot conveniently be put on the calculator stack.
S2	16507	not used	

Start coding outside the ZX81
Many tools exist to code programs outside the ZX81. The output of these tools can be loaded back into the ZX81. ZX BASIC Editor and TEXT2P allow writing BASIC-programs in a simple editor and export to a format that can be loaded into a ZX81 or a ZX81 emulator, either with new hardware like a ZXPAND-interface or by sound when playing the content as a WAV-file into the ear-plug. Screens can be drawn with ZXDRAW.

To code a machinecode program outside the ZX81 an editor for the program and an assembler to turn that code into a program is needed SJASM is such a compiler and NOTEPAD can be the editor. This is not the only solution.
Many other assemblers are available. It is up to the user self to determine which editor and assembler is the most easy to use. On each platform (Unix, PC, Apple) a solution is available.

C-compiler
It is possible to code in C and compile to ZX81-Z80 machinecode.
Coding in a higher language will in the compilingproces always create overhead in the code. Overhead has 2 disadvantages:
1) it consumes space
2) extra code will slow down the program

BASIC-compiler
Like a C-compiler a BASIC-compiler will also create overhead. For the same reasons using a BASIC-compiler is not wanted.

Assemblermodel
The model here below is the base for any program to assemble.
It is optimized in the reduction of overhead bytes and all initial set up is placed over the systemvariables to reduce even more bytes,

The model

```
; 12 bytes from #4000 to #400B free reuseable for own code

           org #4009

; in LOWRES more sysvar are used, but in this way the
; shortest code over sysvar to start machinecode.
; This saves 11 bytes of BASIC

; DO NOT CHANGE AFTER BASIC+3 (=DFILE)
basic      ld h,dfile/256        ; highbyte of dfile
           jr init1
```

```
            db 236                ; BASIC over DFILE data
            db 212,28             ; GOTO USR 0
            db 126,143,0,18 ; shortest FP notation of #4009

eline       dw last
chadd       dw last-1
            db 0,0,0,0,0,0        ; ? not useable
berg        db 0                  ; ? before loading
mem         db 0,0                ; x OVERWRITTEN ON LOAD

init1       ld l, dfile mod 256   ; low byte of dfile
            jr init2

lastk       db 255,255,255
margin      db 55

nxtlin      dw basic              ; BASIC-line over sysvar

flagx       equ init2+2
init2       ld (basic+3),hl       ; repair dfile pointer
            ld l,vars mod 256     ; end of screen lowbyte
            db 0                  ; x used by ZX81
            db 0                  ; x used by ZX81
            ld h,vars / 256       ; end of screen highbyte

frames      db #37                ; after loading LD (HL),N
            db #e9                ; set JP (HL) as end of
                                  ; screenmarker
            xor a
            ex af,af'
            jp gamecode           ; YOUR GAMECODE, can be
                                  ; everywhere

cdflag      db 64
; DO NOT CHANGE SYSVAR ABOVE!

; free codeable memory
gamecode    jr gamecode

; the display file, Code the lines needed.
dfile       db   118
            db   "D"-27,"E"-27,"M"-27,"O"-27,118

vars        db 128                ; becomes end of screen
last        equ $
```

```
end
```

An explanation of the model
The ZX81 needs a setup of the systemvariables to load a and execute a game from tape. After loading the ROM will continue with the execution of the BASIC line mentioned in the systemvariable NXTLIN. When a BASIC-program is running the next line to execute is set here. So the ROM will now run the BASIC-line at the address set in this variable.
The address here is “basic”.

The ROM expects a line that complies to the layout of a BASIC-line. First a linenumber (2 bytes) must come, followed by 2 bytes for the length of the line and then the BASIC-statement, ending with a newline. This is how a line is entered from the BASIC-editor and the syntax is checked.
However in an artificially created line checking the syntax is not needed and a lot more can be coded in less bytes.

The shortened BASIC-line is in the model at address “basic”.

```
basic       ld h,dfile/256          ; highbyte of dfile
```
The ROM is expecting a linenumber here. A linenumber can have a value between 1 and 9999. The linenumber is stored as a 2 byte value, high byte first. The BASIC can be fooled with a piece of machinecode pretending to be a linenumber as long as 2 bytes together form a number less than 10.000.
The savest method is to keep the first byte less than 39 (38*256+255= 9983). In the model the first byte is the opcode LD H,N which happened to have the value 38!

```
            jr init1
```
After the linenumber the length of the line is stored. This is needed to calculate the next line in BASIC to set in NXTLIN. However the program
only uses 1 BASIC line. Only a start to machinecode. The next line is irrelevant to the program. So we can set ANY value as length.
We set a JR to skip te real BASIC statement.

```
            db 236                  ; BASIC over DFILE data
            db 212,28               ; GOTO USR 0
            db 126,143,0,18         ; shortest FP of #4009
```
This part is the start of the machinecode. The BASIC-interpreter from the ROM will now execute the command GOTO USR 0, where the 0 is only a 0 for the syntax. The actual address where the USR will go to is stored in the floating

point notation on the last line. The 126 is a marker that a floatingpoint number will follow. After that normally a 5 byte floatingpoint number is coded. The last 2 bytes however only add a value after the integer value, the number behind the decimal point. The number routine from USR will skip these 2 bytes and only accept the integer value. For the start it is therefore irrelevant which values follow. The bytes are read, but irrelevant. so we don't need to add the bytes and simpy use what follows in the sytemvariables. The ZX81 will now start the machinecode at address #4009. Our BASIC-start into machinecode is fully coded over the systemvariables. We don't lose bytes to code a line AFTER the sytemvariables, like the ZX81 would normally do.

If a program would return to BASIC the interpreter would expect a newline, however the program will stay in machinecode. The endmarker is therefore not needed. An endmarker (linefeed) which would be added automatically when typed on a ZX81.

The machinecode initialisation
The machinecode is now called. What are the next steps before the game starts? Again the ZX81 is at address “basic” but now the machinecode is run.
So first the highbyte of the screendisplay is set in H

```
basic       ld h,dfile/256        ; highbyte of dfile
            jr init1
```

Next step is a smal relative jump over the real BASIC.

```
init1       ld l, dfile mod 256         ; low byte of dfile
            jr init2
```

The lowbyte of the screen is set. The setting is done in 2 parts
since a single LD HL,dfile did not fit the available space.

Finally the screenpointer is set over the earlier used systemvariables.
During an intrupt the screen can now be displayed.

```
init2       ld (basic+3),hl ; repair dfile pointer
```

However..... the screen defined in this example is too short for a full screen. According to the manual the smallest compressed screen would hold
25 linefeeds. A screen can be further compressed with a single JP (HL) as final byte on the screen. This address is calculated like dfile before.

```
            ld l,vars mod 256     ; end of screen lowbyte
            db 0                  ; x used by ZX81
            db 0                  ; x used by ZX81
            ld h,vars / 256       ; end of screen highbyte
```

The value #E937 is lowered by 1 after loading making it opcode LD (HL),#E9.
The second byte must have bit 7 set to keep frames work correctly.
The XOR A and EX AF,AF' is added to delay the intruptroutine.
Often not needed, but can be used with a compressed screen.
Finally the end of the screen is set and your machinecode program is called.

```
frames     db #37                ; after loading LD (HL),N
           db #e9                ; set JP (HL) as end of
                                 ; screenmarker
           xor a
           ex af,af'
           jp gamecode           ; YOUR GAMECODE, can be
                                 ; everywhere


frames     db 255,255       ; after loading CP 255, skipped
           ld (hl),#e9      ; JP (HL) = end screenmarker
           jp gamecode      ; GOTO YOUR CODE.
```

This model will show the text “DEMO” on the screen when compiled and loaded. You can code your code at the line starting with GAMECODE.

Compressed screen
As mentioned, according to the ZX81 manual the screen can be compressed. On a 1K ZX81 the screen will always be compressed. An empty screen would be 25 newlines in memory. While coding games I was thinking if it would be possible to compress a screen even further. The ZX81 uses a trick to determine if a byte is data or code to execute on the screen. The hardware determines if bit 6 of a screenbyte is set, If so then the program will execute the code otherwise it will work out together with the ULA which character must be displayed.
The display of a character is done in 4 tstates. The ROM knows the end of a line with the command HALT. This command has bit 6 set, so it is executed and the ZX81 will wait for an intrupt. Due to this waiting the ZX81 can use shortened lines on the screen. When you don't have a HALT at the end of a line the ZX81 is capable of displaying 34 characters on a line, allthough the 34th character is not capable of displaying inverse characters.
JP NN als has bit 6 set so this command could be used to jump back to a previous HALT. In theory this should then fill the rest of the screen.
A JP-command however uses 10 tstates. Due to this timing not all shortened screens gave a good display. The command JR uses 12 tstates but doesn't have bit 6 set so this will not run as executed code. The command JP (HL) has bit 6 set and executes within 4 tstates, the same time needed for a character. It also is just 1 byte in size so it can be used IF the register HL has a value that can be

used to jump to. HL holds the address of the start of each new line. Placing JP (HL) as first “character” on a line will make the ROM executing the backjump to the start until an intrupt occurs and the intruptroutine continues with the next line, which keeps starting with a jump to the startaddress. The rest of the screen is filled without the need of more HALTs. The use of JP (HL) already saves a byte when your game uses 23 lines, Less lines on the screen will save more bytes. Only when you use a full screen with 25 HALTS (=24 screenlines) the use of JP (HL) at the end has no effect. If your game has a size of a checkboard your screen would be defined like this

```
dfile       db #76                  ; first HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
            db #E9                  ; JP (HL) fills the rest of the screen.
```

In this example your screen doesn't need 16 HALT's to fill the screen, just a single JP (HL). Your screen is even more compressed than sir Clive defined. You save another 15 bytes.

The model above will set the final JP (HL). The model will do this over the destination of VARS. Since we will not return to BASIC no VARS are needed. By wrtiting over the endmarker from VARS (needed as last byte to load a game) we don't even lose a byte when we have full screen.

When you place a JP (HL) over the first HALT the entire screen will go blanc. The screen is still displayed but as a full blanc screen. You could use this to alter your screen and after changing your screen repair the first HALT and your screen will be shown at once.

Bytesaving technics in machinecode

Like BASIC uses bytesaving code we can do similar tricks in machinecode. Most of them are quite logic. Use JR not JP to save a byte. As long as your jump is within 127 bytes this will work. If your jump is too large see if you have a jump to the same location between your jump and the destination. If so do a JR to that address and from there the program will JR to the destination. XOR A for LD A,0 but only when flags may change. Other tricks are to only set the lowbyte of a register when the highbyte is already right.

These are just a few suggestions. Depending the code you write many optimizations are possible in the context of the game. If, for instance, you have a game where you swap between 2 players. You can define player1 as 1 and player 2 as 2. By defining them as 0 and not 0 you can set A to player 1 with XOR A and to player 2 with LD A,H when H is always filled with ANY value but 0. This trick is used in the 2016 version of Chess 1K.

Coding in machinecode
This book is not a book to learn machinecode. Many books handling the Z80 can be used to learn the base of coding the Z80. This book will provide several useable routines that are shortened to use in your code.

Before you start coding it is wise to form a plan of your game. A game is a piece of code that allows interaction from a player to an action by the computer. This defines about all games possible, wether it is a boardgame, a puzzle, an arcadegame or whatever the game should do.

Each game wil have the following parts to make it a game.

Start, wait for startkey to be pressed
Initialisation like resetting score, set level, set lives
A game loop with
– Displaying progress on screen
– Reading input from player
– Scoring
– Test end of game or continue game loop
End of game routine
Restart

Since “restart” is the same as “start” you can alter the order like this.

End of game routine
Start, wait for startkey to be pressed
Initialisation like resetting score, set level, set lives
A game loop with
– Displaying progress on screen
– Reading input from player
– Scoring
– Test end of game or continue game loop

In this order a jump to the startroutine is saved. As previously mentioned in the model there is a jump to the startcode, so this can be START without the

cost of a few bytes.

With the order of the routines set you can think what your game should do.
If your game has multiple options to start, the startroutine must be able to handle all starts and the initialisation must set the parameters for each option.
If you have a single game with ie “PRESS S FOR START” you only check for a S-key and then reset scores, set lives etc.

After the setup the gameloop will start. The computer shows something on the screen and the player must act on it. With the right action you score something, a wrong action will cost a live or ends the game. When a game ends the endgameroutine follows. This routine could be a check if a highscore has been reached and set a new highscore. After that the game can restart.

When each part is carefully defined what to do you can code your game.
To make a game attractive your game must have an incentive to play.
A highscore is such an incentive. Each time you play you want to beat that score. A game can become boring when progress doesn't result in better playing by the computer. This could be speeding up the game, more enemies, less time to solve a task or better artificial intelligence (AI) in next levels

The AI in a game can be:
– Fixed
– Based on situation/level
– False playing

An example of a fixed AI is my game of Othello.
The computer will try to place stones in a fixed order.

Based on the situation/level could be that a computeropponent would do better moves towards a destination. My game WORLDBAR used this technic.
The AI will pick a random number. If the number is larger than a testvalue the waiter runs in a random direction. If the number is smaller the waiter runs 1 step towards the destination. Each level the testvalue has a different value making the waiter run more directly towards the table where he needs to go.

A game that uses an AI where the computer isn't playing fair is the game BACKGAMMON on the ZX Spectrum. On a higher AI-level the values of the die will be the values for the best move the computer can make. No random throw but false play by the computer and almost impossible to beat.

THE ROM
When coding a game in 1K the use of ROM-routines would help you with

saving bytes in your program. Although the ROM has several useable routines it is not usefull to use the ROM. You could use the CLS-routine in the ROM to clear a screen, however... when in less then 3,5K free memory the ROM compress your screen. When your screen has a fixed size, like a chessboard, you will lose the size and reading fields on the board becomes impossible.

Besides the poblems on the screen several routines do use systemvariables. Calling these routines makes using the systemvariable-area for other use impossible. Some routines can be used, the routine to translate a keypress into a keyvalue, but check your routine for the use of systemvariables.

REM-line

Each book in the early '80s told you to make a REM-line to place your machinecode. Due to typing the code on the ZX81 you are bound by use of the editor. This will result in a REM-line with the following syntax
<linenumber><size><REM-command>< n positions in line><linefeed>
Size and linefeed are calculated and placed by the computet. This syntax is relevant when a BASIC-program is run and the REM-line is passed by the program. The REM-command itself is needed to have a correct syntax when entering the line. When your program uses the above model you will
1) you don't enter the line by the editor
2) never pass the REM-line when BASIC-runs
3) you never return to BASIC after starting machinecode

Due to not taking these 3 steps we can skip entering a REM-line at all.
The compiler can add your machinecode without

1) a line number	2 bytes
2) the line	2 bytes
3) the REM-opcode	1 byte
4) the linefeed	1 byte

Using the model from above will therefore save you 6 bytes from entering a REM-line.

Game layout

Before you start coding a game it is good to draw your gameplay and layout out. Determine where in memory you store data, the area of data created by the program can be used for one time routines and routines that you can copy over the systemvariables. You also define the size of your screen,
but also the size of your scoredisplay. Most of my games have a first line on screen that looks like this:

```
dfile    db       #76                                    ; first HALT
```

```
score   db      28,28,28,28,28                  ; 5 bytes size score
        db      0                               ; a space
        db      'N'+101,'A'+101,'M'+101,'E'+101 ; name of the game
        db      0                               ; a space
hisc    db      28,28,28,28,28                  ; 5 bytes hiscore
        db      #76                             ; end of line
```

The size of the score is important. Best is a score sized in the maximum score you can get in a game. The maximum score in bowling is 300, so 3 bytes will fit the score. If you doubt about the highest score adding 1 byte cost less room than testing your code on an overflow.

Initialisation
Each new game you need to wait for a startkey, reset score and lives//speedup (if needed).

Keycontrols are next to read but starting a game is mentioned seperately.
A simple start of a game is to read a startkey.

```
nostart     LD      A,(LASTK)
            SUB     191
            JR      NZ,nostart
```

This 7 bytes routine will loop until a key in the row HJKL-Newline is pressed. Most controls don't use this row so it is a good start. You can't press it by accident.

The result of this routine is that A after start holds the value 0. This is also the value of a space.We can use this to reset the score.

Resetting the score is writing only zero's over the displayed score.

```
            LD      HL,score
reset       LD      (HL),28         ; a visible “0” in ZX81 ASCII
            INC     HL              ; go to next field
            CP      (HL)            ; test this field for a space (A=0)
            JR      NZ,reset
```

Reading keys
Reading keys is one of the most important parts in a game.
By reading keys you can add controls to your game.
Be aware what kind of control you need. If your game only needs 1 direction per move you can use the controls from the intruptroutine from the ZX81.
If you need multiple directions (ie diagonal move with up and left) you need to

code your own keyboardreadingroutine.

Your game can be waiting for input from the keyboard or just scan the keyboard and continue the game. A program that awaits input could be a cursor moving over a board that only continues when you have made a move. A game like chess, checkers, backgammon. A game that only scans the keyboard and then continues could be “Space Invaders”, no move detected,
you stay still and will not fire. The enemies will move on.
In both programs you can use the systemvariable LASTK to check if a key is pressed. A simple routine to check if ANY key is pressed is:
You can use the label LASTK since the model above has defined the place in memory of LASTK.

```
wait4key    LD    A,(LASTK)
            INC   A                  ; test from #FF to #00
            JR    Z,wait4key
```

This routine reads what key is pressed during the screenupdateroutine.
The A-register reads the port of the key that is pressed. The ZX81, the Jupiter Ace and the ZX Spectrum all use the same matrix to read the keyboard.

```
Port                                                  Port
    +---+---+---+---+---+ +---+---+---+---+---+
#F7 ! 1 ! 2 ! 3 ! 4 ! 5 ! ! 6 ! 7 ! 8 ! 9 ! 0 ! #EF
    +---+---+---+---+---+ +---+---+---+---+---+
#FB ! Q ! W ! E ! R ! T ! ! Y ! U ! I ! O ! P ! #DF
    +---+---+---+---+---+ +---+---+---+---+---+
#FD ! A ! S ! D ! F ! G ! ! H ! J ! K ! L ! NL! #BF
    +---+---+---+---+---+ +---+---+---+---+---+
#FE ! SH! Z ! X ! C ! V ! ! B ! N ! M ! . !SPC! #7F
    +---+---+---+---+---+ +---+---+---+---+---+
```

On the ZX81 when a key is pressed the portvalue is saved in the first address of LASTK. When no key is pressed this address will hold the impossible portvalue of #FF. In the second address (LASTK+1) info about the key is stored.

Both options need some delay. Even with the screendisplay done by the Z80 on the ZX81 without a delay both programs would run too fast to control the game. In the first option you can get the delay by checking if a key is pressed or not.

```
waitkeyup   LD    A,(LASTK)
            INC   A                  ; test from #FF to #00
```

```
                JR      NZ,waitkeyup        ; if not 0 any key is pressed

waitkeydown     LD      A,(LASTK)
                INC     A                   ; test from #FF to #00
                JR      Z,waitkeydown       ; if 0 no key pressed
```

These 2 routines together will wait until no key is pressed and then until any key is pressed. This routine doesn't tell you which key is pressed.

I told earlier that we can use a routine to decode which key is pressed.
To use this ROM-routine you need to give the BC-register the right value.
You can alter the routine from above like this to get a keyvalue.

```
waitkeyup       LD      A,(LASTK)
                INC     A                   ; test from #FF to #00
                JR      NZ,waitkeyup        ; if not 0 any key is pressed

waitkeydown     LD      BC,(LASTK)          ; get port in C and value in B
                LD      A,C                 ; copy port to A
                INC     A                   ; test from #FF to #00
                JR      Z,waitkeydown       ; if 0 no key pressed
                CALL    #7BD                ; ROM-routine to find keyvalue
```

The ROM-routine will now give back the value of the key pressed.
The values of the keys are shown in this table:

```
    +---+---+---+---+---+ +---+---+---+---+---+
key ! 1 ! 2 ! 3 ! 4 ! 5 ! ! 6 ! 7 ! 8 ! 9 ! 0 ! key
val !15 !16 !17 !18 !19 ! !24 !23 !22 !21 !20 ! val
    +---+---+---+---+---+ +---+---+---+---+---+
key ! Q ! W ! E ! R ! T ! ! Y ! U ! I ! O ! P ! key
val !10 !11 !12 !13 !14 ! !29 !28 !27 !26 !25 ! val
    +---+---+---+---+---+ +---+---+---+---+---+
key ! A ! S ! D ! F ! G ! ! H ! J ! K ! L ! NL! key
val ! 5 ! 6 ! 7 ! 8 ! 9 ! !34 !33 !32 !31 !30 ! val
    +---+---+---+---+---+ +---+---+---+---+---+
key ! SH! Z ! X ! C ! V ! ! B ! N ! M ! . !SPC! key
val ! X ! 1 ! 2 ! 3 ! 4 ! !39 !38 !37 !36 !35 ! val
    +---+---+---+---+---+ +---+---+---+---+---+
```

The SHIFT-key is not recognized as a single key press in the intruptroutine.
Only pressing the SHIFT-key will therefore give a 'nokey'-pressed return.
The keys have the values 1 to 39. We can use this in our advantage when we have a game that needs the keyvalue but must continue when no key is pressed.

A game could use this routine when not to wait for a keypress:

```
waitkeydown  LD    BC,(LASTK)      ; get port in C and value in B
             LD    A,C             ; copy port to A
             INC   A               ; test from #FF to #00
             CALL  NZ,#7BD         ; ROM-routine to find keyvalue
```

When a key is pressed the value 1 to 39 is calculated by the ROM.
When no key is pressed the ROM is skipped and A will have the value of 0.
In all steps A will hold an unique value where the program can work with.

The routine to wait for a key to be pressed can be shortened further.
The loops to wait for a key going up and key going down can be combined.

```
             (XOR  A               ; make sure A <> FF)

waitkeyup    INC   A               ; test from #FF to #00
waitkeydown  LD    BC,(LASTK)      ; get port in C and value in B
             LD    A,C             ; copy port to A
             JR    NZ,waitkeyup    ; if not 0 any key is pressed
             INC   A               ; test from #FF to #00
             JR    Z,waitkeydown   ; if 0 no key pressed
             CALL  #7BD            ; ROM-routine to find keyvalue
```

The double read of LASTK is now deleted. The program uses the value of A in the first test. If you don't know the value of A on entry you can add a XOR A to make it 0 in the first test. When your routine will always start with A holding a value unequal to 255 you can even skip the XOR A.
This routine waiting for a keypress will create a delay that keeps the input in control. A game that only scans the keyboard will need a delay routine to keep the game playable.

Delayroutine
A delayroutine is nothing more then a loop that consumes some time.
You can a delayroutine to control the speed in a game. You can set the game to a fixed speed or use a delayroutine to speed up a game when progress is made. On a ZX Spectrum a HALT will stop the CPU for 1/50 sec. On a ZX81 HALT is used in the screenupdate for the start of the next line to display. HALT is therefore no useable delay on the ZX81. On the ZX81 the systemvariable FRAMES is decreased each 1/50 sec. On a ZX Spectrum a similar sytemvariable is available as a clockcounter. We can use this systemvariable as a simple delayroutine:

```
        LD    HL,FRAMES
        LD    A,(HL)        ; current value lowbyte FRAMES
        SUB   3             ; Any number for the delat needed
wfr     CP    (HL)          ; wait frames
        JR    NZ,wfr        ; until ROM syncs with your value
```

You can also use a routine like this

```
        LD    HL,2000             ; delay value
wait    DEC   HL
        LD    A,H
        OR    L
        JR    NZ,wait
```

The second routine is a byte shorter. You can use this routine to speed a game up by changing the value 2000 in a lower value during gameplay.
Disadvantage: you don't know the exact time 1/50 sec takes and when your game runs on the ZX81-emulator on the ZX Spectrum it will run slower since the emulator will update the timer as it should but running the countercode would go slower. Besides the runtime the emulator has an option to speed up games that use this kind of delay by altering the value after the SUB.

Gamecontrols
After reading a key and tranlating it into a keyvalue you need to evaluate the value for your controls. The most direct way to do this is testing each possible value in a piece of code. Suppose we use QAOP for up, down, left and right. The code could be:

```
        CP    10            ; is it Q
        JR    Z,up
        CP    5             ; is it A
        JR    Z,down
        CP    26            ; is it O
        JR    Z,left
        CP    25            ; is it P
        JR    Z,right
```

At the location up, down, left and right you do what is needed for the move.

Diagonal moves
With diagonal moves you can't use the method of reading LASTK,

When multiple keys are pressed LASTK will see that as no key is pressed.
To add diagonal moves to your game you need to use reading the IN-port directly in your game and evaluate what is read to determine the key pressed. You need to read each key seperately and you need to check the next direction after doing a move in an earlier found direction.

In the same way as above you could code it like this

```
LD      A,#FB           ; port QWERT
IN      A,(254)         ; read port
BIT     0,A             ; test bit 0, matching the Q
CALL    Z,up            ; matching, do up-move
LD      A,#FD           ; port ASDFG
IN      A,(254)         ; read port
BIT     0,A             ; test bit 0, matching the A
CALL    Z,down          ; matching, do down-move
LD      A,#DF           ; port YUIOP
IN      A,(254)         ; read port
BIT     1,(HL)          ; test bit 1, matching the O
CALL    Z,left          ; matching, do left-move
LD      A,#DF           ; port YUIOP
IN      A,(254)         ; read port
BIT     0,(HL)          ; test bit 0, matching the P
CALL    Z,right         ; matching, do right-move
```

A few things spring into sight. To handle a direction the JR Z is changed into a CALL Z. This is needed to continue testing other directions. You must end your direction alteration with a RET. Second change is the size of each direction. Due to the setting and reading of a port you need 7 bytes per direction to fuly test a possible move. Third thing is that reading left and right is done with the same port. That is due to the fact that O and P are besides eachother on the keyboard. This routine can combine the test of left and right together as

```
LD      A,#DF           ; port YUIOP
IN      A,(254)         ; read port
BIT     1,(HL)          ; test bit 1, matching the O
CALL    Z,left          ; matching, do left-move
BIT     0,(HL)          ; test bit 0, matching the P
CALL    Z,right         ; matching, do right-move
```

Scoring
Get rewarded for your gameplay is why we play games.
Define a simple method of scoring. Early pinballmachines gave single point.

Then you got at least 10 points and in the '80s you got at least 1000 points.
Even Pacman gaf 10 points for a dot. Keep it simple. A single point will do fine.
At least 10 points will cost you a useless byte on the unit-positions.
A score must be kept in memory and a score must be shown on screen.
Why not combine both? Keep the score on screen in a displayable notation.

Most of my games will give you 1 point when you capture something, or you make a move. The simple routine to do that is:

```
addpoint    LD    HL,score+5      ; start 1 position behind your score
            DEFB  17              ; a machinecodetrick !
ten         LD    (HL),28         ; set value back to “0”
            DEC   HL              ; go to next digit
            INC   (HL)            ; add 1 point
            LD    A,(HL)          ; get digit
            CP    38              ; test digit > “9”
            JR    Z,ten           ; if so add 1 in next digit
```

When you add a point the progam set HL to the first position behind the last digit. Then a 1 byte savetrick is used. We need to skip the resetting to “0” at label 'ten'. By using opcode 17 we do a LD DE,NN and skip 2 bytes from 'ten'. You can use this trick when DE is not used at this moment. Other opcodes are also possible, like JP C,NN when you know that carryflag is not set on entry. Next HL points to the right digit to increase. After the increase the value is tested for overflow to next digit and corrected. A 5 digit score 99999 with add 1 will make the game crash due to altering the linefeed placed before the score. Adding a check is 5 bytes where an extra byte in the score is just 1 byte.

Highscore
When a game ends what beter result can you have than setting a new hiscore.
We need to check if the played score is more than the current hiscore.
After coding my first checkroutine I have perfeced the routine and I will shaer and explain the perfect hiscorecheck here:

```
            LD    HL,score-1      ; Make HL point to score-1
            LD    DE,hisc-1       ; Make DE point to hiscore-1
            LD    BC,6            ; Set BC to length score+1
same        DEC   C               ; shorten length to copy
            INC   DE              ; goto next digit hiscore
            INC   HL              ; goto next digit score
            LD    A,(DE)          ; get digit hiscore
            CP    (HL)            ; test against digit score
            JR    Z,same          ; if the same, score still the same
```

```
CALL C,#19F9          ; if higher, use ROM to copy
```

An explanation. There are 3 possible situations.

1) Current score is lower than hiscore: 00167 vs 00444
Each digit is tested against the same digit in the other score.
First digit: the same, next digit is called
Second digit: the same, next digit is called
Third digit: hiscore is larger. No carry flag is set, so no call to the ROM.

2) Current score is higher than hiscore: 01234 vs 00123
First digit: the same, next digit is called
Second digit: current digit higher, carry flag is set>
The ROM at #19F9 is called. At #19F9 the command LDIR RET is coded in the original ROM. Your ZX81 must have the original ROM to work here. When #19F9 is called DE and HL points to the remaining part of the score to be copied and BC is lowered to the bytes to copy.

3) Current score is equal to hiscore
All digits of the score will jump back testing the next.
How is the routine then ended?
The line on the screen is cleverly chosen. The hiscore is set at the end of the line and the score at the start. After testing the full score the next test that is done is testing the space (0) against the newline (#76) behind the hiscore. This will be no match so the loop is broken, but also the space is less in value than the linefeed so no call is made to the ROM.
Your routines looks further than needed (outside the box).

Adding hires to your game will need 19+size score bytes in memory.

Simplified calculations
When your game needs a calculation you don't always need to use the formula's that are defined by natural rules. An approximation of a formula will work just as well. You are not coding a science model, you are coding a game. Even the ROM is using models to calculate SIN and COS. Try to find a simple matching formula for your task. I had a game where a bouncing ball needed to rebounce up to 80% of the original heigth. 80% would mean multiply by 8 and divide by 10. With 75% I would have the same number of rebounces before the ball went dropdead. 75% is *3/4. This formula was easier and shorter to code.

```
LD    B,A         ; save original value
RRA               ; divide by 2
```

```
        ADD   A,B           ; 0.5A+A = 1.5A
        RRCA                ; With carry : 1.5A => 0,75A
```

Another option is to precalculated the needed results and store it in a table. If the table is shorter than the formula in code you save bytes.

A simplified calculation will also be faster than a calculation with the ROM-calculator and systemvariable are not used. A random value is easier with a simplified calculation.

Randomness

The ZX81 and its successors use the following formula to calculate the next random number: (75*(your number+1) -1) / 65536

The ROM can calculate that but it is still calculated. It is random because we can't calculate the next value without knowing the current parameters.
For your game you can use ANY routine that “calculates” the next random value. The BASIC-function RND returns a value between 0 and 1, To make it useable for a dice you multiply by 6, add 1 and take the integer.

A simple table for pseudo-randomness is the ROM. My 1K hires game CAR RACE uses the ROM each game in the same order. You can learn where the next car comes just like you can learn the digits of PI.

This routine will give you a dicenumber each time you call the routine:

```
rnd     LD    HL,(seed)     ; somewhere a saved ROMpointer
        INC   HL            ; go to next address in ROM
        LD    A,H
        AND   31            ; keep H in ROM-area
        LD    H,A
        LD    (seed),HL     ; save for next round
        LD    A,(HL)        ; get random value
inrange SUB   6
        JR    NC,inrange    ; set number in range
        ADC   A,6           ; now A is 1 to 6
```

You use the ROM as table to read a random value.
With this method the seed is a pointer to the ROM.
When you set the pointer with a random start it will be random enough for players. You can set a random pointer in the same way as the BASIC does. The command RAND sets the seed with the systemvariable FRAMES. You can do the same in your game. After pressing start you can read FRAMES and store it

in 'seed'.

Compressed screen at loading

The 1K ZX81 can only load 949 bytes savely without crashing.
You can define your screen as above like a chessboard but you can compress it. There are several methods to compress the screen. My tributegame DIGGIT has a very extended compression to make the playscreen fit the 949 bytes. After loading the screen is decompressed to use the full 1024 bytes in 1K of RAM.
For a screen with just a copy of the same line for a number of times (8x on a chessboard) you can set the first line. On the memory location of the second line you set a decompression routine. One screenline and decompressionroutine is less than a full screen. With a compressionroutine you can use extra bytes in your program that can't be loaded. An example and explanation:

```
dfile   db #76                  ; first HALT
line1   db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
line2   LD    HL,gamestart
        PUSH  HL                ; stack startaddress
        LD    HL,line1          ; get line1
        LD    DE,line2          ; destination line2
        LD    BC,7*9            ; 7 lines of length 9 to copy
        JP    #19F9             ; use LDIR in ROM to copy
vars    db    128
```

There a 2 things to keep in mind
1) The model sets the JP (HL) at the end of the screen.
You must calculate the end by hand (=line2+63) and set end there.
2) Decompressing the screen takes time.
If during decompression the displayintrupt is activated the program will execute part of the routine. This will crash your game. The XOR A and EX AF,AF' from the model delays the intruptroutine. This will give enough time to decompress the screen. This screen has 27 bytes, the decompressed screen has 73 bytes. You can code an extra 46 bytes in your game. Well not exactly. The game loads at #4009, 9 bytes further than start or RAM, total size would be 9+949+46=1004. This would leave a stack of 20 bytes. That is really small. I would suggest a stack of 32 bytes. You can try to make it smaller, but 32 is mostly a safe size. This would give you 34 extra bytes to code.

Create a screen with your CLS-routine

A CLS-routine cost some memory, but you can also use it to create your screen. This way you can have a real short screen while loading, but you must create the screen before an intrupt occurs. Like the compressed screen above you need to calculate the end of the screen again. You also need to start your game

directly at the CLS-routine. So you need to clear the score and set lives when compiling to get a good first play. This is a screen creating CLS-routine:

```
cls       LD   HL,vars
          LD   B,8        ; 8 lines
sline     LD   C,8        ; with each 8 fields
cline     LD   (HL),0
          INC  HL
          DEC  C
          JR   NZ,cline
          LD   (HL),118
          INC  HL
          DJNZ sline

; rest of the game

; the display file
dfile     db 118
vars      db 128
last      equ $
```

Code over systemvariables
You can place a routine between dfile and vars which you can call before creating a screen. You can copy a piece of code over the systemvariables and then make the screen. If you do this you reuse the systemvariable-memory. The memory where the routine was when loaded will be reused as screen. The easiest way to reuse the systemvariable-area is by storing your variables there. You need to create room in your game where you can store your routines. That is hard to do when you don't have such space. Compressing the screen is such an option. When you want to resuse the systemvariable-area some systemvariables may not be overwritten. These are:

– Dfile
– Lastk
– Frames and
– Cdflag

All other addresses can be reused for your program.
This is the routine I used in DIGGIT, the screen is compressed so in the screenarea I had room to store the routine while loading and a copy routine to copy the routines,

```
        LD HL,fieldc            ; field and right over sysvar
        LD DE,#4000
        LD bc,37
        LDir

        LD HL,rnd1              ; random routine
        LD e,nxtlin mod 256     ; to be set over sysvar
        LD c,17                         ; copy random routine
        LDir

        jp norm                         ; decompress the screen


; field and right are copied over the sysvar, saves 35 bytes!
fieldc  LD      A,B
        PUSH    BC
        RRA
        LD      L,A
        XOR     A
        LD      B,A
        LD      H,A
        LD      A,C
        RRA
        LD      C,A
        JR      pf2
d_file  dw      dfile           ; screenpointer, not codeable in lowres
pf2     LD      A,L
        ADD     A,A                     ; x 2
        ADD     A,A                     ; x 4
        ADD     A,L                     ; x 5
        ADD     A,A                     ; x10
        LD      L,A
        ADD     HL,HL           ; x20
        ADD     HL,BC           ; add x
        LD      BC,scr
        ADD     HL,BC
        POP     BC
        LD      A,(HL)
        RET

rightc  LD      A,35            ; right is also copied over sysvar
        CP      C
        RET     Z
```

```
        INC   C
        DB    24,jrp              ; A JR jrp won't work due to relocation

; randomroutine placed over sysvar
rnd1    LD    HL,rseed                  ; seed pointer
        LD    A,(HL)                    ; get seed RRCA ; a=a/2
        RRCA                    ; a=a/2
        RRCA
        RRCA
        XOR   31                ; swap low bits
        ADD   A,(HL)            ; add seed
        DB    17                ; hide frames in DE
frames  DW    65535             ; frames used by zx81
        ADD   A,E               ; add framecounter
        LD    (HL),A            ; save new seed
        RET
```

I stated that DFILE and FRAMES were not allowed to be overwritten.
Yet in this copy I do it both. However... Both systemvariables keep their function. DFILE is still the start of screen and frames is hidden in the opcode LD DE,NN. Best of all the random routine now stored at NXTLIN is a short routine that also use the timer to create more randomness. The command RAND nr and funtion RND are combined in 1 routine.

To copy routines in lowres over the systemvariables is harder since you need another compression to save bytes. In 1K hires it is easier since you need an area to define your screen. In this area you can store your routines that will copy over the systemvariables.

SP-area

The further we come in this book the harder the bytesaving-routine become to code. This trick is only useable if you can include the StackPointer area in your gamecode. As said a game can hold 949 bytes. At the end of the memory the stackpointer is set. This would making coding into the SP-area impossible.
You need about 32 bytes for th stackpointer to make your program run. During the screenintrupt 5 registers and a call are added extra to the stack. Your game might need a few stacked registers too. After loading and before the first screendisplay the stack only needs 2 addresses. The rest is not yet used.
You can use the unused memory for a onetime routine. DIGGIT does it.
A normal game has this layout

SYSTEMVARIABLES
PROGRAM

SCREENAREA
STACKPOINTER

DIGGiT swaps the stackpointer and the screenarea. The unpacked screen will fill up to the end of RAM. On the stackpointerare the unpacking of the screen is coded and fully run before the intrupt occurs.

Routines on screen
The problem in 1K is a shortage of memory
To save memory you can set one time routines or data on the screen.
You need to prevent that the code is executed while on the screen.
This can be solved with a JP (HL) on the first position on the screen.
This is an example of commands on the screen.

Suppose we have an 8x8 screen.

```
dfile   db #76                  ; first HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db 0,0,0,0,0,0,0,0,#76  ; 8 fields and HALT
        db #E9                  ; the JP (HL) to fill the rest of the screen.
```

And you want to add redefined controls where this is default:

```
key1        CP      10          ; is it Q
            JR      Z,up
            CP      5           ; is it A
            JR      Z,down
            CP      26          ; is it O
            JR      Z,left
            CP      25          ; is it P
            JR      Z,right
```

This could be your screen at the start.

```
dfile       DB      118
line1       DB      0,#E9       ; the “only” line!
loadstart   LD      HL,key1+1   ; first key in program
```

```
        LD    B,4              ; 4 keys control
        DB    254,118          ; CP 118, skip Newline
        LD    DE,udlr          ; UDRL text
loop    LD    A,(DE)           ; get next direction
        LD    (line1),A        ; write on screen
        DB    254,118          ; CP 118, skip Newline
        INC   DE               ; next pointer
        EXX                    ; swap registers
wup     INC   A                ; Test for keyup
wdown   LD    BC,(lastk)       ; get last key
        DB    #26,118          ; skip newline LD H,N
        LD    A,C
        JR    NZ,wup
        INC   A
        JR    Z,wdown
        DB    33,0,118         ; skip newline LD HL,nn
        CALL  #7bd
        EXX
        LD    (HL),A           ; store key pressed
        INC   HL
        INC   HL
        DB    254,118          ; CP 118, skip Newline
        INC   HL
        INC   HL
        DJNZ  loop
        LD    A,(line1+1)
        DB    254,118          ; CP 118, skip Newline
        LD    (line1),A
        XOR   A
        LD    (line1+1),A
        DB    254,118          ; CP 118, skip Newline
        JP    start
        DB    "U"-27,"D"-27,"L"-27,"R"-27
        DB    0,118
        DB    #E9
```

After loading you start the program at LOADSTART and will alter
the keycheckroutine with the keys pressed on start.
The program needs extra commands to pass the linefeed.
The text "only" is between brackets since the JP (HL) trick will fill the entire screen with the key to press.
The CLS-routine will then clear the screen and make it useable for the game.
This is an example how you can set code on the screen.

Hidden data on the screen
Minesweeper is a game where you must find out where mines are hidden.
The data of the game would show on the screen when you select a field.
You need to define a screen for the size of the game you play.
The info about the game must be stored in memory. Due to the unique way the ZX81 handles the screen you can define the size of the screen and store the info of the game on the screen without showing that info. You can hide data on the screen invisibly. We already know of 2 executable codes on the screen: HALT and JP (HL).

If you have run the redefining routine from above you must have noticed the space between each letter. This space is caused in the time it takes to execute the JP (HL). The display of a character takes 4 tstates and the execution of JP (HL) also. The timing causes a space to be displayed when executing a command. We can use this trick to display spaces where we in fact store information about the game. A few of my 1K games that use this trick:

MINESWEEPER:
Information about the field near n mines or a mine is stored invisibly.

SNAKE:
The game plays on a black screen, the snake is shown as spaces, but is in fact information which way the tail is moving.

DIGgIT
When jumping you can only go up 1 pixel. A test on the character above is done. You see a space but in fact a hidden data is coded to block the upgoing move from a jump.

FINGERTWISTER:
A hidden character is placed before the first key on a row to find the start of the row.

PATHMAZE:
Like MINESWEEPER the whole maze is hidden on the screen

HIDDEN PCTURE:
The full picture is hidden on the screen.

but also 16K games use this trick.
LODE RUNNER uses this trick to hide the escapestairs and to find the digged holes.

The commands that you can use as hidden data are:

Hex opcode	Command	
40	LD	B,B
49	LD	C,C
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
57	LD	D,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
5F	LD	E,A
64	LD	H,H
6D	LD	L,L
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
7D	LD	A,L
7F	LD	A,A

BC and HL are used by the display. Only nonchanging setting are possible
There are many more commands with just 4 tstates but these are the only
that have bit 6 set which is needed to hide the data.

Since so many games I have coded use this trick I decided to recode
my ZX81-emulator for the ZX Spectrum to allow these game in the emulator.
The ZX Spectrum Next still needs an update from the emulator to allow these
games to run properly with the right display.

Simple routines
The model set command #E9 as an endmarker for the compressed screen.
You can use this as a test to find the end of the screen. Here is a
clear screen routine:

```
cls     LD      HL,dfile+1              ; start at first screenposition
```

```
clrfld  LD    (HL),0          ; clear field
skiplf  INC   HL              ; go to next field
        LD    A,(HL)          ; get value
        CP    #76             ; test against Newline
        JR    Z,skiplf        ; skip the newline
        CP    #e9             ; test against end of screen
        JR    NZ,clrfld       ; if not continue
```

The ZX81 doesn't have a function like SCREEN$ on the ZX Spectrum.
You can make your own function in a piece of code that you can both use as a PRINT AT and a SCREEN$. I mostly use BC as X and Y coordinate in a game.
You can use them also to calculate the position on the screen.
Suppose your coordinates on the board run from 1,1 to 8,8
Your field would the be calculated like this

```
field   LD    HL,dfile-9
        LD    A,B             ; Y coordinate
        ADD   A,A             ; Y*2
        ADD   A,A             ; Y*4
        ADD   A,A             ; Y*8
        ADD   A,B             ; Y*9
        ADD   A,C             ; Y*9+X
        ADD   A,L             ; Add position of dfile
        LD    L,A             ; set position
        RET
```

Note:
This routine needs the entire screen within the same highbyte.
A larger screen needs a different routine.
HL holds the screenposition. In your game you can read what is on the screen with LD A,(HL) or write something to the screen with LD (HL),n

Codingtips
Coding a 1K game is tense. A few tips to be succesful.
1) Draw on paper what you want to do, think how you would organize the memory.

2) Use the model
The model is optimized in bytes and it starts your game. It also sets a minimized screen.

3) Make slow steps, test every step and save progress in backups.
When you screw a version you have a recent version to fall back to.

4) First make your game work. Even it is over 1K.
When your game works you know how much bytes you need to reduce to make it fit When your game doesn't fit 1K yet start with the simple optimizations, look for 16 bits loading of registers and check if the high byte is the same. If so only set the low byte. Make sure you know the flags in the game.
When you need a SBC HL,DE without the carry and carry is never set before then you don't need a AND A to rest te carry, an example

```
        CP      5
        JR      Z,nosub
        AND     A               ; undo C when A<5
        SBC     HL,DE
nosub   LD      A,(HL)
```

can be

```
        XOR     5               ; test A=5 and reset C
        JR      Z,nosub
        SBC     HL,DE
nosub   LD      A,(HL)
```

Joystickcontrols
When your game fits you can add extra options like joystickcontrols.
The two most known joysticks are Kempston and more recently the ZXPAND-joystick port. The Kempstonjoystick is a shorter routine to code than the ZXPAND-joystick. The controls are different as well

The Kempston-joystick reads controls as 000FUDLR, the 5 bits of the joystick at the end with a bit set when a direction is used.

The ZXPAND-joystick reads controls as UDLRF000, the 5 bits of the joystick at the start with a bit reset when a direction is used.

No joystick connected will return the value 255 in both cases

Adding joystickcontrol would add a whole new check of directions if you want to have both keyboard and joystick. Not everyone has a joystick interface so keyboard must be part of your game. Joystick is bitwise and keyboard is keywise. A full kempstonjoystick routine adds 22 bytes to the code.

```
Kempston    XOR     A
            IN      A,(31)          ; read port 31
            INC     A               ; test no connection
```

```
                JR      Z,testkeyb      ; test keyboard
                DEC     A               ; undo change

                RRA                     ; test bit 0
                JR      NC,right        ; right pressed
                RRA                     ; test bit 1
                JR      NC,left         ; left pressed
                RRA                     ; test bit 2
                JR      NC,down         ; down pressed
                RRA                     ; test bit 3
                JR      NC,up           ; up pressed
                RRA                     ; test bit 4
                JR      NC,fire         ; fire pressed

testkeyb        LD      BC,(lastk)
; and the rest of keyboardcontrol
```

To add ZXPAND-joystick you need this routine

```
                LD      BC,#E007        ; preload portaddress
                LD      A,#A0           ; value to write
                OUT     (C),A           ; activate joystick
                INC     HL              ; wait about 10 tstates
                DEC     HL              ; after activation
                IN      A,(C)           ; read joystick

                INC     A               ; test portaddress
                JR      Z,testkeyb      ; no zxpand interface

                ADD     A,A             ; test bit 7
                JR      C,up
                ADD     A,A             ; test bit 6
                JR      C,down
                ADD     A,A             ; test bit 5
                JR      C,left
                ADD     A,A             ; test bit 4
                JR      C,right
                ADD     A,A             ; test bit 3
                JR      C,fire
```

With a trick you can shorten the code in both routines with 5 bytes, but that is not easy to explain. I leave it like this.

Intruptroutine

On almost all Z80-machines you can use IM 2 to handle intruptroutines.
Not on a ZX81. There is another way to trigger intrupts The only downfall
to intrupts on other machines is that only a short time can be used for your own
routines. The ZX81 needs enough time to also update the screen.

The update of the screen is done with a jump to the address pointed in IX.
You can use this to make your intruptroutine by setting IX to your routine.
In your game you set the following command

```
        LD      IX,intr             ; make IX point to intr-routine
```

Your routine must have the following structure:

```
intr    LD      HL,dfile+#8000      ; load HL with highmem screen
        LD      A,#F5               ; A holds #F5 for right display
        LD      BC,#1901            ; 24 lines screendispla like ROM
        CALL    #2B5                ; display the screen

; your code comes here
```

To end the intrupt correctly you need to add the following lines

```
        CALL    #292                ; handle end of screen
        CALL    #220                ; framecounter and keyboard
        LD      IX,intr             ; Set IX back
        JP      #2A4                ; Exit correctly
```

When you use these intrupts and use the screen as mentioned on the first line
you don't need the systemvariable DFILE anymore. This will give you a block
of 37 bytes to code freely from #4000 to #4024.

This intrupt is also the step into hiresroutines.

1K hires
True hires was already known in 1984 and in 1996 further improved,
The improvement from 1996 led to a demo of hires in 1K.
The demo draws circles in a small 6x8 character sized screen.
I found this demo when I was searching for true hires on a ZX81 to convert a
game from the ZX Spectrum to the ZX81. This 1K hires demo was nice, but I
was thinking how to use it to code a game in just 1K with hires graphics.

My first idea was a game that would fit in that small screen. I had a 4x4 shiftpuzzle like GUUS FLATER in my mind, but then I got a really out of the box idea. I wanted a game that gave the illusion of a full screen with just a small screen in reality. When you hold a lantern in a cave you only see the enlighted surroundings. When you move you see a different view with that same lantern. That was the idea I would plan in my game. A person trapped in a dark maze in search of 5 keys to escape. The screen was a 5x5 area but the area moved over the entire screen, giving the illussion of a full screen.

This is the hiresdisplayroutine of my first 1K hires game:

```
begin       LD   IX,hr          ; set hiresstart

;hires display WIWO DIDO the case of Mazeddy's castle
hr          LD   B,3            ; delayroutine
hr0         DJNZ hr0

            LD   HL,screen
            LD   E,bytcol
topline     LD   BC,#8000+lines
            INC  B              ; always 1 topline
            LD   A,192-lines
            SUB  B
            LD   (notend+1),A

xmove       CALL delay

hr1         CALL lbuf2
            DJNZ hr1
notend      LD   B,0

hr2         LD   A,H
            LD   I,A
            LD   A,L
            CALL lbuf+#8000
            ADD  HL,DE
            DEC  C
            JP   NZ,hr2

hr3         CALL lbuf2
            DJNZ hr3

            CALL #292
```

```
            CALL #220

            LD   IX,hr
            JP   #2A4

lbuf        LD   R,A
            DEFB 00,00,00,00,00,64,64,64
            DEFB 64,64,64,64,64,64,64,64
            DEFB 64,64,64,64,64,64,64,64
            DEFB 64,64,64,64,64,64,64,64
delay       RET  NC


lbuf2       LD   D,3
lb2         EX   (SP),HL
            EX   (SP),HL
            DEC  D
            JP   NZ,lb2
            NOP
            RET
```

Although far from optimized this routine is cleverly designed to work with selfmodfiying code to display the screen on various locations

This routine only displays a hiresscreen, no lowres text. The game switched to a lowresscreen when the exit was found. The next game got lowrestext and hires display. The hires display in each game is different. After many games I made this model for the display and initial setup:

```
; Model 1K hires

            ORG  #4009

; program starts here, both BASIC and machinecode
; the initialization also repairs any possible 48K bug.

Basic        EX AF,AF'       ; delay int,opcode no bit6
             LD H,B          ; preset for 48K bug to #40
             JR init0        ; continue where room

             DB 236,212,28   ; The BASIC
             DB 126          ; fully placed over sysvar
```

```
             DB 143,0,18     ; start BASIC=#4009 also MC

             DW last         ; needed by loading
chadd        DW last-1
xptr         DW 0
stkbot       DW last
stkend       DW last
berg         DB 0
mem          DW 0
             DB 128

             DB 0,0,0

; all above reusable AFTER loading
lastk        DB 255,255,255; used by ZX81
margin       DB 55           ; used by ZX81
nxtlin       DW basic        ; reusable after load
init0        LD IX,hr        ; hr lowbyte bit 5 reset
                             ; lowbyte over flagx which
                             ; resets bit 5 on load
                             ; HR must be set on right
                             ; address or game crashes
             LD E,L          ; DE now #xx.L
taddr        DW 0            ; used by ZX81 on LOAD only
                             ; unharmed code
             LD B,4          ; copy >1K code
frames       DB #16+1        ; LD D,n , after LOAD -1
             DB #C0          ; highbyte must have bit 7 set
coords       LDIR            ; DE now #C0.L = Hl +#8000
                             ; 48K bugfix beforedisplay
prcc         JP start        ; continue to mainprog

cdflag       DB 64           ; used by ZX81

; Place ANY code to fill up to above #4040

; some code or data here

; HR must start AFTER #403F, but before #4070
hr           LD HL,lowres+#8000 ; the lowres display
             LD BC,#309
             LD A,#1E        ; I-reg value for char
             LD I,A
             LD A,#FB
             CALL #2B5       ; show lowres screen

;your hr part
```

```
; fixed end of HR-routine
exit          CALL #292          ; back from intrupt
              CALL #220
              LD   IX,hr
              JP   #2A4

x             EQU 101
lowres        DB 118,0,0,0
score         DB 28,28,28,28,0
lives         DB 28,0

              DEFB "N"+x,"A"+x,"M"+x,"E"+x,128
hiscore       DEFB 28,28,35,37,118
              DEFB 118,#e9

vars          DEFB 128

last EQU $
```

Like the earlier model this needs explanation. After loading the basic-line is again executed and a machinecodestart to BASIC is made.

```
Basic         EX AF,AF'          ; delay int,opcode no bit6
              LD H,B             ; preset for 48K bug to #40
              JR init0           ; continue where room
```

The systemvariables are used to the fullest to do all necessary initialisation.
EX AF,AF' will load A' with the value 9, After the USR-function BC holds #4009 and A holds the same value as C. With the EX AF,AF' the first intrupt is delayed in display. This is needed to do all initialisation in time.
LD H,B The H-register is preloaded with #40.
And the next thing is a jump over the BASIC-line.

```
init0         LD IX,hr           ; hr lowbyte bit 5 reset
                                 ; lowbyte over flagx which
                                 ; resets bit 5 on load
                                 ; HR must be set on right
                                 ; address or game crashes
              LD E,L             ; DE now #xx.L
taddr         DW 0               ; used by ZX81 on LOAD only
                                 ; unharmed code
              LD B,4             ; copy >1K code
frames        DB #16+1           ; LD D,n , after LOAD -1
              DB #C0             ; highbyte must have bit 7 set
```

```
coords      LDIR            ; DE now #C0.L = Hl +#8000
                            ; 48K bugfix beforedisplay
prcc        JP start        ; continue to mainprog
```

For the display we need to set the IX-register. This is done now. This is coded over the systemvariable FLAGS. FLAGS needs bit 5 reset on loading. IX must point to an address with the low byte of IX having bit 5 reset, This is mentioned further in the model. Next is LD E,L. L always has a value within systemvariablerange. There will not be any relevant code on this address. TADDR will have unrelevant code after loading and can be skipped.
Now B is set to 4 so BC holds #409, a bit more than 1K.
FRAMES is decreased once after loading so this now holds LD D,#C0.
As you can see bit 7 of highbyte in FRAMES is kept set. At this moment DE holds #C020, HL holds #4020 and BC #409. The LDIR now copies over 1K to 32K higher. When less memory in RAM it is copied over the start. This will never crash. The copy is only needed for ZX81 computers with 32K, available RAM above 48K. The display of a 1K hires game is different per game. Some games use a JP command executed while doing a display with a “fake” call to upper memory. 32K RAM is prepared to jump back to the RAM-command on the real address but only for the first byte, A 3 byte opcode would now pickup the address on the real address when 32K RAM is available. The first time I was ponted to this bug I thought I could not solve this and 1K hires games with this trick would not work on a 32K RAM ZX81 without a simple fix.Each game could have a not working command on any address. A full copy of the game in upper memory would fix the problem. I added the copy in a game and the game worked. Later I added the full copy in the systemvariables. Your game is now ready to run on every ZX81.

The next part in the model is the hiresdisplayroutine.
This routine will start with the display of a lowresscreen.
I started to code all games with score, lives, name and hiscore and after that I added the hiresscreen.

```
; some code or data here
```

The routine or data here starts ar #403C. HR must be at #4040 or more.
A small routine or some data can be set here. Even the few lines of the screen. Anything to fill the 4 bytes.

```
; HR must start AFTER #403F, but before #4070
hr          LD HL,lowres+#8000 ; the lowres display
            LD BC,#309
            LD A,#1E        ; I-reg value for char
            LD I,A
```

```
            LD A,#FB
            CALL #2B5      ; show lowres screen
```

This part does the display of the lowresscreen. You need to display 192 lines in total, including your hires display. If your hiresscreen has 64 lines then your lowresscreen must display 128 lines. 1 lowres text is 8 lines. You can “fool” the ZX81 with empty lines by setting the C-register higher. This model will show 2 lines of lowrestext after 8 empty lines, the C-register and the B-always hold 1 more than it displays. The lowresscreen will use 16+8=24 lines. Your hires screen can use 192-24=168 lines. You can use more, but it better to stay in a 192 line display. As you might see the model is now almost equal to the routine to have your own intruptroutine. The only difference is the number of lines displayed in the lowresscreenroutine.

```
;your hr part
```

You need to add your customized hiresdisplayroutine. You need to think out a way to store your screen and a way to display the screen. And at this point it is up to you to figure that out. I don't know what you want in the game. All I can offer are 81 games with all some kind of display. Reuse the display for your own games, learn from them and make your own game. The sources are available,

```
; fixed end of HR-routine
exit        CALL #292          ; back from intrupt
            CALL #220
            LD   IX,hr
            JP   #2A4
```

This part ends the display and is the same as the userintruptroutine.

Epilogue

So this ends the description I made from coding games in just 1K.
I hope it can give any use for your coding.

Johan “Dr Beep” Koelman