UNIVERSITY OF LJUBLJANA
FACULTY OF COMPUTER AND INFORMATION SCIENCE

Matej Horvat

Emulation of the Iskra Delta Partner computer

THESIS

FIRST CYCLE
UNIVERSITY STUDY PROGRAM
COMPUTER SCIENCE

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2017/2023

The results of the thesis are the intellectual property of the author and the Faculty of Computer and Information Science in the University of Ljubljana. To publish or otherwise make use of the results of the thesis requires written permission from the author, the Faculty of Computer and Information Science, and the mentor.

The text was written with the author's own text editor and converted to the PDF format with his own converter. Both were written in the NewtonScript programming language and executed with the author's own implementation.

Note: this is an "unofficial" translation of the thesis. The Slovene version from 2017 is the officially submitted and defended version. However, certain errors found in that version have been corrected here and some parts have been updated to reflect later developments.

The Faculty of Computer and Information Science issues the following work:

Topic:

In the 1980s, the Slovenian computer industry was blooming. One of the computers being produced at that time was the Partner from Iskra Delta. Not many were produced; still, one is preserved at the Faculty of Computer and Information Science of the University of Ljubljana. Its continued use and exposure may eventually render it non-operational, so it is necessary to take measures to properly preserve it.

For this thesis, archive the Partner's software, including the corresponding documentation, then, through reverse engineering, study its architecture and software. From the resulting findings, develop an emulator of the computer that will recreate its behavior as faithfully as possible. The final product should be available to use on a regular PC with minimal effort.

Special thanks to:

- the mentor, who made this project possible and provided guidance while I was writing this text,
- Janez Kožuh, who donated a Partner computer to the faculty,
- Tadej Pečar for uncovering the secrets of floppy disks and floppy disk drives,
- doc. dr. Tomaž Dobravec for a replacement floppy disk drive,
- izr. prof. dr. Veselko Guštin for contacting Janez Kožuh,
- members of the Laboratory of algorithmics and the Software engineering laboratory – for lending space and being patient.

Table of contents

Abstract . . . 13
1 Introduction . . . 15
2 Archiving . . . 19
2.1 The central processing unit . . . 19
2.2 The operating system . . . 21
2.3 Working memory (RAM) . . . 21
2.4 Imaging the bootable floppy disk . . . 22
2.5 Imaging the hard disk . . . 23
2.6 Dumping the ROM . . . 28
3 The emulator . . . 33
3.1 Development process . . . 33
3.2 General structure . . . 34
3.3 The central processing unit . . . 34
3.4 Memory . . . 35
3.5 The video board . . . 35
3.5.1 AVDC . . . 36
3.5.2 GDP . . . 41
3.5.3 Displaying the picture on the screen . . . 45
3.6 The serial interface . . . 46
3.7 The keyboard . . . 47
3.8 The mouse . . . 50
3.9 Printing . . . 50
3.10 Mass storage devices . . . 51
3.11 The real-time clock . . . 52
3.12 Other devices . . . 53
4 Evaluation of the final product . . . 55
4.1 Accuracy . . . 55
4.2 What could be improved . . . 56
4.3 The user experience . . . 57
4.4 Portability . . . 57
5 Conclusion . . . 59
References . . . 61

Glossary of acronyms

- ASCII · American standard code for information interchange
- AVDC · advanced video display controller
- BCD · binary-coded decimal
- BDOS · basic disk operating system
- BIOS · basic input/output system
- CCP · console command processor
- CTC · counter/timer channels
- DCGG · display character and graphics generator
- DMA · direct memory access
- DRAM · dynamic random access memory
- EPROM · erasable programmable read-only memory
- GDP · graphic display processor
- ISR · interrupt service routine
- PIO · parallel input/output
- RAM · random access memory
- ROM · read-only memory
- SIO · serial input/output
- TPA · transient program area
- VAC · video attributes controller

Abstract

Title: Emulation of the Iskra Delta Partner computer

Author: Matej Horvat

This thesis presents the development process of a program that emulates a Partner computer, which was produced by the Slovenian/Yugoslav company Iskra Delta in the 1980s. It describes its main components and how they are emulated, as well as its system software and the methods and processes used to archive it.

Keywords: Iskra Delta, Partner, emulator, emulation, reverse engineering, archiving, preservation, virtual machine, CP/M, Z80.

Chapter 1

Introduction

In the 1980s, the Slovene/Yugoslav company Iskra Delta produced desktop computers (microcomputers) as well as larger computers (minicomputers) intended for business use and software for them [14]. Compared to home computers of the time, relatively few people used it; consequently, it is hard to find these computers and information about them today.

The Partner is one of those computers. It is a desktop computer first introduced in 1983 [17] and intended to be used as a small business or development system [14]. Models capable of displaying graphics were introduced later: the 1F/G (with one floppy disk drive), the 2F/G (with two floppy disk drives), and the WF/G (with a hard disk and one floppy disk drive – shown on picture 1.1) [1]. (They are also referred to as the Partner GDP. We are not sure when exactly these models were introduced; the preliminary edition of the user manual is from March 1987 [1], while some applications for them are dated 1985 or 1986.)

As hardware does not have an infinite lifespan, at some point in the future it will no longer be possible to see and use these computers and their software, and it will be increasingly difficult to find people who know anything about them. A part of Slovenian history would disappear.

Preserving hardware is complicated, but preserving software is relatively easy. This is made possible by a program called an emulator. It lets software to be executed on other hardware than it was originally developed for.

To emulate means to implement the interface and functionality of some system on a system with a different interface and functionality [13]. If we apply this definition to an entire computer system, such as the Partner, it means that we implement the Partner's interface (the instruction set of its central processing unit) and functionality (input/output devices) on some other (modern) computer so that software developed for the Partner can be executed on a modern computer.

Picture 1.1: The Partner WF/G used for this diploma thesis.

An emulator is a kind of a virtual machine. Virtual machines allow executing the same code on different architectures and operating systems without having to adapt it specifically. The code is not necessarily the machine code of some physical processor; several programming languages are compiled to virtual machine code (usually called bytecode) that is then interpreted or recompiled [13]. The term "emulator", however, is usually used for virtual machines that recreate physical machines – including input/output devices. Emulators are whole-system virtual machines [13].

An emulator works by executing the machine code of the "guest" program just as it would be executed by the processor in the original computer, and when that virtual processor attempts to access hardware, the emulator responds the same way (or as similarly as possible) as the original hardware would. The program therefore does not "know" it is not being executed in its native environment. If, for example, the user presses a key on the keyboard, the program will react to it the same way as it would on the original hardware, and if it attempts to render a picture or text on the

screen, the user will see the same result as on the original hardware – in this case, an actual Partner computer.

In this diploma thesis, we will take a look at the development of PartEm, a program that emulates a Partner computer, specifically the WF/G model.

In the second chapter, some of the main characteristics of the Partner's hardware and software are presented, and then the procedures used to transfer the Partner's software to a modern computer so that we could even begin developing the emulator.

In the third chapter, the development process of the emulator is presented, as well as its structure, what other components the Partner contains, how they work, and how they are emulated.

In the fourth chapter, we take a look at the final product, compare it to an actual Partner, and present ideas for improvement.

Chapter 2

Archiving

An emulator is useless without any software to execute. In any case, we wanted to archive everything we had at hand:

- the Partner user manual (archived by scanning; it contained a lot of crucial information about how the Partner functions),
- a bootable floppy disk,
- the Partner's hard disk,
- the Partner's ROM.

The contents of the hard disk in particular seemed useful, as we would be able to compare the accuracy of the emulator to the actual Partner in real time. (Later, we also discovered it contained some documentation and source code; both were helpful.)

These procedures we done in parallel with reverse engineering. In particular, making an image of the hard disk contents and dumping the ROM required some knowledge about the Partner's processor, memory, operating system, and input/output devices, so we will describe those in this chapter as well, but how they are emulated will be described in the next chapter.

2.1 The central processing unit

The Partner's central processing unit is a Zilog Z80A microprocessor clocked at 4 MHz [1]. The Z80 is an 8-bit microprocessor introduced by the American company Zilog in 1976 [15]. Its instruction set is a superset of that of Intel's 8080A microprocessor [16]. The Z80A is a somewhat newer version capable of operating at higher clock speeds.

The processor has two sets of eight 16-bit registers or four pairs of 8-bit registers that can be used independently [15, 16]:

- AF or A and F. A is the accumulator; it is used as the implicit input and/or output

operand for most arithmetic operations. F is the flag register; it is modified implicitly by arithmetic operations and used mostly for conditional jumps.
- BC or B and C. These are general-purpose registers, but have a special meaning for some instructions; for example, B is the implicit operand of the DJNZ (decrement and jump if not zero) instruction, which is frequently used to implement loops whose iteration count is known in advance, and BC serves as the counter for the LDIR (load, increment, repeat) instruction and other block instructions.
- DE or D and E.
- HL or H and L. HL is mostly used as a pointer.

The AF register pair is exchanged with its equivalent in the other register set with the EX AF, AF' instructions, and the other three pairs all at once with the EXX instruction. Within the same register set, DE and HL can also be exchanged with the EX DE, HL instruction.

The following 16-bit registers are not duplicated:

- PC, the program counter, which is not directly accessible,
- SP, the stack pointer,
- IX and IY, which are index registers used as the base address in addressing modes with a signed 8-bit displacement.

The processor has a 16-bit memory address space occupied by ROM and RAM and an 8-bit input/output address space occupied by devices.

Interrupts can be handled in three modes. All of them first push PC to the stack, but they differ in what they set it to next [15, 16]:

- In interrupt mode 0, the processor fetches an intruction from the data bus. This is usually an RST instruction, which sets PC to a value between 0 and 38h.
- In interrupt mode 1, PC is always set to 38h. Software must then poll devices to figure out which one caused the interrupt.
- In interrupt mode 2, the low 8 bits of an address are read from the data bus while the high 8 bits are copied from the special register I. The word at that address is then copied to PC. Because each device puts a different value on the bus, execution automatically continues in its ISR. This interrupt mode is used in the Partner; its user manual contains the addresses of pointers to ISRs relative to the I register [1].

2.2 The operating system

The Partner uses CP/M 3 (also called CP/M Plus), developed by the American company Digital Research, as its operating system.

CP/M is a single-tasking disk operating system for microcomputers based on the Intel 8080 microprocessor or compatible (such as the Zilog Z80). It provides functions for working with files and basic input/output functions [11].

It is made up of two parts. The BIOS is the part that differs depending on the computer and is supplied by its manufacturer (in this case Iskra Delta). Its functions are called by the BDOS, which is hardware-independent and implements higher-level functions. It, in turn, is called by applications [11].

In addition to all the functions required by CP/M, the Partner's BIOS also contains a function for drawing graphics primitives [1]. This function exists only on the Partner and only on Partner GDP models. It is used by applications so they do not have to directly control the GDP.

Additionally, CP/M also contains a command interpreter (CCP) for starting programs and working with files, as well as other programs. They (including the CCP) are not permanently loaded, butu are loaded to the same area as other applications [11].

2.3 Working memory (RAM)

The Partner has 112 kilobytes of working memory (RAM). Because the processor has only a 16-bit address bus and can therefore address only 64 kilobytes, the memory address space is divided into two parts. The lower 48 kilobytes are occupied by the currently selected bank, while the upper 16 kilobytes are always accessible and belong to neither bank.

CP/M uses the first bank for itself. The other is available to applications; in CP/M terminology, the application space is called the TPA [11]). The upper 16 kilobytes are also reserved for the operating system; they contain ISRs and entry points to various operating system functions, most of which reside in the first bank.

2.4 Imaging the bootable floppy disk

In addition to the Partner and its user manual, we also had a bootable floppy disk at hand. We wanted to archive its contents as it could be useful for testing the emulator until we had a hard disk image.

The Partner uses high-density 5.25-inch floppy disks. The formatting is unique, so they cannot be read by e.g. a PC AT clone because of the different controller. The formatting is [1]:

- double-sided,
- 73 tracks per side,
- 18 sectors per track,
- 256 bytes per sector,
- sectors are interleaved in a 2:1 ratio (but the BIOS hides that detail),
- the total capacity is 657 kilobytes.

We used a KryoFlux [4] controller for imaging. This controller connects to a modern computer over a USB port and can read the "raw" data encoded on the disk. The decoding is done by software; we used the program HxCFloppyEmulator (meant to be used with the floppy disk drive emulator of the same name, which works like a floppy disk drive but uses a USB mass storage device or SD card as media) [8]. This way, we obtained a sector-by-sector image that can be analyzed and used by the emulator. It is a file that contains one byte for each byte of each sector on the disk (without interleaving). The software or person working with such an image must know its "dimensions" in advance, but in most cases, this is not necessary, because file systems abstract media as an array of equally sized blocks.

We started developing the emulator by analyzing the operating system loader on the disk and tried to "get it as far as possible" – ideally to the point where the operating system is fully loaded. Later, however, through a more detailed analysis, we found out that the disk is meant to boot an older (non-GDP) Partner model (we could not actually try booting our Partner from the disk because its floppy disk drive was not operational, so we did not know that immediately). However, soon later we made an image of the hard disk, so we continued development with that instead.

2.5 Imaging the hard disk

The WF/G model contains a Seagate ST-412 hard disk with a capacity of approximately 10 megabytes [1]. The one in our Partner was fortunately still flawless and contained a lot of software and other useful files (such as a file containing documentation about the hard disk itself and its controller, and the source code for one of the installed applications), so we wanted to make a copy of its contents. (We can only guess how much longer it will remain operational.)

Simply copying all the files (e.g. to floppy disks) would not be sufficient because:

- we could not copy the operating system loader this way; it is not stored in a file, but in a special area occupying the first few sectors of the disk,
- this would not preserve potential deleted files, which are not visible to the operating system anymore, but whose fragments may still remain on sectors that were not reused for some other file,
- we would have to reconstruct the file system for the data to be usable by the emulator; just like the physical computer, the emulator knows nothing about files, but only has a (virtual) disk, which is a collection of sectors whose meaning is determined by the operating system.

The floppy disk drive in our Partner turned out to be non-operational anyway and we did not want to disassemble the computer out of fear of damaging it, or at least not without already having made a copy of the data on the hard disk.

We therefore had to find a way to transfer the entire contents of the hard disk (from the first sector to the last) to a modern computer without using floppy disks, and, if possible, without writing any new data to the disk, as that might overwrite fragments of deleted files. By coincidence, we found a file on the disk that contained a lot of information about it and its controller; for us, the most crucial was the capacity: 1224 tracks, 32 sectors per track, 256 bytes per sector.

All Partners, regardless of the hardware configuration, have at least one serial (RS-232) port, so we tried to find a way to use it for data transfer. At first, we did not even know whether it worked; each time we tried to redirect the output of some program to the serial port (instead of the screen) as specified by the manual, the Partner stopped responding and the computer on the other side received no data regardless of how we configured the serial ports on both sides.

Then we found the program GRMT20 on the hard disk, which was also mentioned in the manual (under the name RMT20 – the actual program identifies itself as "RMT 2.0") as letting the Partner be used as a terminal for some remote computer. We were able to send data with this program; if we typed something on the Partner's keyboard, we saw that on the other computer, but not vice versa. (What also surprised us was that the speed was double of what we specified. The BIOS offers the choice of 1200, 2400, or 4800 bits per second, but in practice, this turned out to be 2400, 4800, or 9600 bits per second. The manual is not consistent regarding the allowed speeds.) But it was already enough for our needs.

The manual listed the input/output port addresses used by the serial port. The keyboard is also connected to a serial port, but uses a different connector [1]. By analyzing the code on the bootable floppy disk that communicates with the keyboard, we found out how to send data through the serial port. We also discovered that if we forcibly terminate GRMT20 (by pressing the SHIFT and BRK keys; on the Partner, this terminates the currently executing program and loads the CCP), the serial port remains usable for our own code to send data through.

With this knowledge, we were able to write a program in the BASIC programming language on the Partner, that – by calling routines written in assembly language – read each sector of the hard disk and sent it through the serial port:

```
10 DEFINT A-Z                'All variables are integers.
20 A = &H7000                'Starting address for machine code.
30 DEF USR0 = A              'Entry point to SETDMA.
31 DEF USR1 = A + 2          'Entry point to SETTRK.
32 DEF USR2 = A + 4          'Entry point to SETSEC.
33 DEF USR3 = A + 6          'Entry point to READ.
34 DEF USR4 = A + 8          'Entry point to SendSec.
39 I = A
40 READ B                    'Read a byte from a DATA statement.
50 IF B = -1 THEN GOTO 90
60 POKE I, B                 'Write it to memory.
70 I = I + 1
80 GOTO 40
90 Z = USR0(0)               'Call SETDMA.
100 FOR T = 0 TO 1223        'Track counter.
110 POKE A + 10, T AND 255
115 POKE A + 11, T \ 256
120 Z = USR1(0)              'Call SETTRK.
```

```
130 FOR S = 1 TO 32             'Sector counter.
140 POKE A + 10, S
145 POKE A + 11, 0
150 Z = USR2(0)                 'Call SETSEC.
160 PRINT "Track"; T; "sector"; S
170 Z = USR3(0)                 'Call READ.
180 IF PEEK(A + 10) = 0 THEN GOTO 210
190 PRINT "Read error!"
200 END
210 Z = USR4(0)                 'Call SendSec.
220 NEXT S
230 NEXT T
```

Lines 30 to 34 define the entry points to machine code routines. Each of them points to a JR (relative jump) instruction because that made fixing bugs easier.

Lines 39 to 80 read data from DATA statements, which contain machine code (more about them later). and write them to addresses starting with 7000h.

Line 90 calls the machine code routine SETDMA, which calls the BIOS function of the same name and passes it the address of the buffer that we will use for reading sectors. Then the program enters the loop that iterates over tracks and sectors (the documentation on the hard disk told us how many of them there are).

Inside the loop, we set the track number and then for each sector in the track, we set that sector number, read it (if this fails, we display an error), and send it.

We implmented the code to read and send a sector in assembly language, because BASIC cannot access the disk on the sector level (we could have actually implemented everything in assembly language, but it would have made debugging harder):

```
        org 7000h
        jr SETDMA; 7000h
        jr SETTRK ; 7002h
        jr SETSEC           ; 7004h
        jr READ             ; 7006h
        jr SendSec; 7008h
Param   dw 0                ; 700Ah (placeholder for parameters)
SETDMA  ld a, 12            ; SETDMA (set...
```

```
         ld bc, Buffer        ; ... the sector buffer address)
         jr BIOS
SETTRK   ld a, 10             ; SETTRK (set the track)
         jr BCBIOS
SETSEC   ld a, 11             ; SETSEC (set the sector)
BCBIOS   ld bc, (Param)
BIOS     ld (.par), a         ; (BIOS function number)
         ld (.par+2), bc      ; Pass the parameter.
         ld c, 50             ; (BDOS function "use the BIOS")
         ld de, .par
         jp 5                 ; Call the BDOS and don't return.
.par     db 0, 0, 0, 0        ; (placeholder for parameters)
READ     ld a, 13             ; READ (reads a sector)
         call BIOS
         ld (Param), a        ; Report success or failure.
         ret
SendSec  ld hl, Buffer
         ld b, 0              ; (256 inner loop iterations)
         ld c, b              ; Checksum := 0.
.next    ld a, (hl)           ; Read a byte from the buffer...
         call SendByte        ; ... and send it.
         add a, c             ; Add to checksum.
         ld c, a
         inc hl
         djnz .next
         ld a, c              ; Send checksum.
SendByte push bc
         ld b, 3              ; (iteration count)
.again   push af
.wait    in a, (0DBh)
         and 4                ; (ready flag)
         jr z, .wait          ; Wait until the SIO is ready.
         pop af
         out (0DAh), a        ; Send the byte to the LPT port.
         djnz .again
         pop bc
         ret
Buffer                        ; (buffer starts here)
```

Most routines only call BIOS functions. This is done by setting the A register to the

function number [7] and calling or jumping to (in case no further action is needed – if we change a CALL/RET sequence to JR, we save two bytes and decrease the time needed to type in the program) the BIOS or BCBIOS routine (the latter also reads a parameter from the address 700Ah). In any case, the BIOS function call is done through a BDOS function, which must be passed the BIOS function number and parameters in a specific structure. This is because applications cannot call BIOS functions directly as they reside in the other bank [8]. BDOS functions are called through the address 5; the jump to the BDOS entry point (which may differ depending on the computer and CP/M version, so address 5 is the documented entry point [8]) is there.

Ideally, we would have used an existing protocol to transfer the data, such as XMODEM, which is simple and widely supported among terminal emulation applications, but it requires the receiver to check the checksum of each packet and notify the sender if an error is detected (as well as to request the next packet). Since we could not receive any data on the Partner, we instead used our own slower protocol, which sent each byte of each sector three times, calculated a checksum, and sent that three times as well. On the receiving computer, we wrote a program to decode this format by "voting" (if at least two bytes out of three match, then that value is used, otherwise there was an error). Sending the entire hard disk contents this way took an entire day (in reality even more due to debugging). It would have been possible to do this faster, but we did not want to take any risks.

Sending a sector (each is 256 bytes long) is done by iterating through the buffer and calling a byte-sending routine for each byte, which sends it three times. The value is also added to a checksum and the checksum is finally also sent three times.

Sending a single byte through the serial port is done by writing to its data address (0DAh), but it is first necessary to wait until the SIO is ready for that. This is done by checking the "ready" flag, read from the status address (0DBh), in a loop.

We wrote and assembled the assembly language code on a modern computer and then typed it into the Partner in the form of DATA statements, which are used in the BASIC programming language for pre-made data:

```
240 DATA 24, 10, 24, 15, 24, 17, 24, 40
250 DATA 24, 47, 0, 0, 62, 12, 1, 90
260 DATA 112, 24, 10, 62, 10, 24, 2, 62
270 DATA 11, 237, 75, 10, 112, 50, 44, 112
280 DATA 237, 67, 46, 112, 14, 50, 17, 44
```

```
290 DATA 112, 195, 5, 0, 0, 0, 0, 0
300 DATA 62, 13, 205, 29, 112, 50, 10, 112
310 DATA 201, 33, 90, 112, 6, 0, 72, 126
320 DATA 205, 73, 112, 129, 79, 35, 16, 247
330 DATA 121, 197, 6, 3, 245, 219, 219, 230
340 DATA 4, 40, 250, 241, 211, 218, 16, 244
350 DATA 193, 201, –1
```

Once the hard disk data was transferred and decoded, we were able to use it in the emulator. We also wrote a program to extract all the files from it (for easier use with modern software), including deleted files – this was only possible if the space formerly reserved for those files was not reused for newer files. We found some files by heuristically searching unused file system blocks (those that were not pointed to by any directory entry). Among the deleted files were some applications, source code, and documentation.

2.6 Dumping the ROM

The ROM (actually an EPROM [1]) contains the initial loader, that, when the computer is powered up or reset, checks whether the RAM is working correctly and loads the operating system loader from a floppy disk or hard disk. The latter is stored on the first few sectors of the storage device and loads the operating system.

At reset, the ROM is enabled (made accessible) and occupies the lowest 8 kilobytes of the memory address space regardless of the selected bank. It is disabled by accessing input/output port 80h (the addresses it occupied are then used by RAM), but cannot be re-enabled without resetting the whole computer.

We assumed it is disabled by the operating system or its loader. This assumption was not wrong, because while looking at the loader on the bootable floppy disk, we noticed that it disables the ROM immediately at the start of its execution, but on the hard disk, the ROM is disabled by the BIOS once the latter is loaded. In any case, the result is the same; by the point the operating system is loaded and capable of executing programs, it is too late to dump (make an image of) the ROM.

We therefore concluded that the only way to dump it is to not load the operating system, but instead a program that will read the ROM and send it through the serial

port. As we did not want to lock ourselves out of the operating system, only one option remained: to replace the floppy disk drive and make our own bootable floppy disk containing this program. (We did not have an EPROM programmer/reader and removing the EPROM from its socket would have required an almost complete disassembly of the computer.)

Our first attempt failed because the Partner did not recognize the disk as bootable, so we made a copy of the bootable floppy disk and overwrote part of the operating system loader with our program, which dumped the entire memory address space (because we did not know which exact addresses are occupied by the ROM) through the serial port the same way as the hard disk imaging program. To initialize the serial port, we started the GRMT20 program, then reset the Partner with the switch on the back side.

Most of the resulting dump was "empty" – it contained the pattern "55 AA" (hexadecimal), which is the result of the memory test. Some of it contained data from before the reset. The ROM seemed to be absent (we would have recognized it by any of the messages it displays on the screen), but we did notice the following code fragment:

```
F600    in a, (80h)
F602    ld hl, 0E000h
F605    ld de, 100h
F608    ld bc, 1600h
F60B    ldir
F60D    jp 100h
```

This fragment disables the ROM by reading input/output port 80h, then copies 1600h bytes from address 0E000h to 100h and jumps there. We knew that the operating system loader, once loaded, starts at this location, so we concluded that the ROM loads the loader to address 0E000h and the above code fragment to address 0F600h and executes the latter. It cannot load it to address 100h directly; the processor starts executing code from address 0 on reset, so the ROM must start at that address, and the Partner manual says it is 4 kilobytes in size [1], so address 100h is definitely in the ROM.

We then noticed that this code fragment is present on the bootable floppy disk and hard disk at the same location. We concluded that this is the actual entry point to the loader. We moved our dumping program to that location (overwriting the instruction that disables the ROM) and ran it again. The new dump also contained

the ROM. The following is the final version of the program:

```
          org 0F600h
Start     di                  ; Disable interrupts.
          ld sp, 0FF00h       ; SP := top of our space.
          call Beep           ; Notify the user that we booted.
          call SendBank       ; The first bank is initally active.
          ld d, 40h           ; (16 kilobytes)
          ld hl, 0E000h       ; (start of the shared area)
          call SendArea       ; Send the shared area.
          in a, (90h)         ; Select the second bank...
          call SendBank       ; ... and send it.
          call Beep           ; Notify the user that we are done.
          halt                ; Halt the processor until reset.
SendBank  ld d, 0C0h          ; (48 kilobytes)
          ld h, 0
          ld l, h             ; HL := 0.
SendArea  call SendSec        ; Send a "sector" with a checksum.
          dec d               ; Repeat this D times.
          jr nz, SendArea
          ret
Beep      in a, (0D9h)
          and 4               ; (ready flag)
          jr z, Beep
          out (0D8h), a       ; (A is now 4, which also causes a long beep)
          ret
```

Note: the routines SendSec and SendByte are the same as in the previous program.

We only found a replacement floppy disk drive a few months after the start of the project; all drives we tried before it either did not work or were not compatible with the Partner. Therefore, the emulator initially loaded the operating system loader by itself into RAM at address 100h and also initialized the hardware. The ROM is therefore not strictly needed, but we wanted to archive it and use it in the emulator for authenticity because it shows a large banner "Delta Partner GDP" at startup. It is shown on picture 2.1.

```
Delta Partner GDP
[ Boot V 1.1 - WF ]
TESTING MEMORY ...
```

Picture 2.1: The text shown by the ROM at startup.

Chapter 3

The emulator

This chapter describes the other components of the Partner and their implementation in the emulator.

3.1 Development process

The development was an iterative process. Once we had an image of the bootable floppy disk and hard disk, our first goal was to get the emulator to the point where it could load the operating system (CP/M).

While doing so, we analyzed its loader, the source code for which is publicly avaliable [10]. At first, we only emulated hardware as much as needed to get the loading and initialization process as far as possible; for example, if some part of the code had a loop waiting for some device to become ready, we simply returned a result that "satisfied" it. We avoided emulating the floppy disk drive controller and hard disk controller completely by intercepting BIOS functions.

After about a month of work, during which we also analyzed the bootable floppy disk image, performed experiments on the actual Partner (to find out how the hardware works), and researching how to image the hard disk, our emulator came to a point where it could load the operating system. From that point on, our goal was to emulate all of the Partner's components as accurately as possible.

We obtained some information about the hardware from the Partner's manual (without it, developing the emulator would have been impossible), and in the case of the video board, from datasheets for the relevant chips. Everything else had to be discovered empirically: to find out how some piece of hardware works, we usually made a hypothesis and then wrote a short program in the BASIC programming language to verify it. We noted the results and implemented the same or similar behavior in the emulator. If the documentation was too vague, it was often helpful to run a program in the emulator and observe how it communicates with the hardware and what data it expects from it. Only for the mouse we were able to use the source code of an application as a reference and so implemented a device that

we did not have and otherwise could not have emulated.

3.2 General structure

Hardware (the digital logic circuits within) works in parallel and synchronizes and controls its behavior with various control signals. The code implementing some program (in our case the emulator), on the other hand, is executed sequentially. As mentioned in the introduction, it is most natural to look at the emulator from the processor's perspective. The processors execues code, and when it has to perform an input/output operation, it does that and then continues code execution. At the highest level is therefore such a loop. Devices used by humans to interact with the computer (keyboard, mouse, monitor) require special treatment. We will first look at the monitor.

On an actual Partner, the picture on the screen is refreshed while the processor is executing code, but in the emulator, for simplicity, the code execution and screen refreshing take turns. In each iteration of the main loop, 80000 processor cycles (the result of dividing the clock speed, 4 MHz, with the screen refresh rate, 50 Hz) are emulated, then the screen is refreshed if its contents have changed since the last refresh. Pending interrupts are also triggered at that time. The emulator then waits 20 ms (1/50 s) or less (depending on how long it took to refresh the screen) and the loop repeats. As a side effect, the emulated processor executes approximately the same number of instructions in a given time interval as the one in the actual Partner (approximately, not exactly, because wait states required by the working memory, which is DRAM, are not taken into account).

In the same loop, the emulator also checks whether a key was pressed on the keyboard, whether a mouse button was pressed or released, and whether the mouse was moved. It passes these events to the appropriate module which processes the event and updates the state of the device. The software running in the emulator will see the new state the next time it will read it. In the case of the keyboard, an interrupt is also fired to notify the software about the keystroke.

3.3 The central processing unit

To emulate the processor, a somewhat modified (for portability) version of the

Zymosis [5] library is used. The library accurately emulates the Z80 processor, including undocumented instructions and other details.

The emulated processor accesses other devices through callback functions. When it needs to access memory or a device, it calls the given function to perform the read or write operation. Memory (ROM and RAM) is taken care of by a pair of functions, while for devices, the read or write request is redirected to the module implementing that device. Each device implements at least the following functions:

- A reset function, which is called at the start of emulation and each time the emulated computer is reset.
- A function for reading data, which takes an address and returns a value.
- A function for writing data, which takes an address and a value. On most devices it also triggers side effects.

Devices can also trigger interrupts. Because of the different nature of each device, there is no universal mechanism for this.

3.4 Memory

In the emulator, memory (ROM as well as RAM) is just an array of bytes accessed by the emulated processor through callbacks. On each access, the address is translated from the emulated processor's address space into the host's address space and then the read or write operation is performed.

3.5 The video board

The picture displayed on the screen is generated by two chips. In general, one generates text and the other generates graphics.

The monitor built into the Partner is monochrome and green. Some configurations of the Partner can also output the picture to a television screen [1]. Ours does not have this option, so the emulator displays the picture as it would be seen on the built-in monitor.

3.5.1 AVDC

The AVDC (a Signetics SCN2674) generates the control signals for the screen and displays text.

Depending on how it is connected to the rest of the system, it can be used in several modes [2]. The Partner uses it in the so-called independent mode, in which the AVDC has its own memory that the main processor cannot access directly, but only by issuing commands.

The memory is divided into two parts, each 4 kilobytes in size. One contains character codes and the other contains their attributes [1]. Read and write operations always affect both.

The following are some of the commands:

- select an initialization register,
- enable or disable the cursor,
- enable or disable interrupts,
- write a character and attribute to the cursor position,
- advance the cursor position,
- fill a block of characters and attributes.

The parameters of the picture generated by the AVDC are set through 15 initialization registers. They are all set through the same input/output port and selected by issuing the appropriate command. Setting a register automatically selects the next one, so software usually sets all at once (and then leaves them unchanged). The following are some of the parameters that can be set:

- Mode of operation. On the Partner, this is always the independent mode.
- Character dimensions. On the Partner, this is always 8×11 [1].
- The duration of various intervals used to control the screen.
- Whether the cursor blinks, and if it does, how often.
- The blinking frequency for characters that have the blink attribute set.
- Number of character columns per row. On the Partner, the user can choose 80 or 132 columns.
- Number of character rows. On the Partner, this is always 26.
- The cursor height.
- Which line of a character is used for underlining if the underline attribute is set.

There are also other registers with their own input/output ports, such as the cursor address and registers that split the picture into multiple parts (the Partner's BIOS does not use the latter).

When generating text, the AVDC can fetch character and attribute codes in two modes:

- In linear mode, it reads them from a contiguous block of memory and displays them in the same order (left to right, wrapping as necessary).
- In row table mode, the software reserves part of the memory for a table that specifies the starting address of each row. The address of the table itself is held in a register. To display each row, the AVDC first reads an address from the table and then fetches characters from the contiguous block of memory starting at the address. This is the mode used by the Partner's BIOS because it has two advantages: it is easy to implement scrolling (only the row addresses have to be rotated) and it allows switching between 80-column and 132-column modes without problems; the BIOS always organizes the rows in memory so that each is 132 characters long. In 80-column mode, the extra columns are simply ignored by both the software and the AVDC. If the user switches to 132-column mode, all the characters stay where they are, only the rows become longer.

The AVDC does not generate the picture all by itself; it also needs the DCGG and VAC [2]. These two chips are not visible to the processor, so they do not have to be emulated explicitly; all that matters is that the final picture looks as it should. The emulator implements their functionality in the same module as the AVDC. The Partner's documentation does not mention these chips, but they are mentioned in the AVDC's datasheet. Generating the picture works roughly like this:

1. The picture is made up of several rows of text and each row is made up of multiple lines of pixels. Because rows are drawn top to bottom and each row left to right, the AVDC reads characters belonging to the particular row in this order, as well as their attributes. (Because each row of characters is made up of several lines of pixels, it has to read each row from memory multiple times. In the emulator, this is not true because the emulator cannot access the host's hardware directly, but draws the full picture in memory and displays that.)
2. The character codes read by the AVDC are passed to the DCGG, which has its own ROM containing glyphs (character shapes) shown on picture 3.1, and (conceptually) produces a picture of 1-bit pixels.
3. The VAC receives the character attributes from the AVDC and determines the colors of the pixels.

Each character has one byte (8 bits) of attributes. The attributes are any combination of those bits. They are:

- the blink bit,
- the underline bit,
- the intensity bit,
- three bits for the background color (on the Partner's monitor, they produce different shades of green; it is not known what they would do on a television screen),
- two bits whose meaning is not fully understood.

Each row can also be given an attribute causing its characters to be double-width or double-width and double-height. In both cases, each character must be written to the row twice; for example, the word "word" becomes "wwoorrdd" because the AVDC still fetches characters the same way, only their left or right half is drawn depending on whether the column number is even or odd. In the case of double height, the next row must also contain the same characters and it must be explicitly set which row is the upper half and which is the lower half. These attributes are set by the high two bits of each address in the row table.

We of course wanted to use the same font to display the AVDC-generated picture as on the actual Partner, but because the ROM containing the glyphs cannot be accessed by the processor, the only way we could get it was by copying the glyphs by hand. To make the individual pixels easier to see, we wrote a program that showed all 256 characters in double width and height. Picture 3.1 shows the entire character set. The Partner's BIOS, however, ignores the highest bit when displaying a character, and which character the other 7 bits represent depends on the currently selected character set (besides ASCII and "YUSCII" – the Yugoslav character set – there are 11 more, of which 3 are undocumented). The BIOS also interprets escape sequences; some of them select other character sets while others set the attributes of subsequent characters, move the cursor, etc.

The manual also mentions the possibility of using custom characters, but we found no documentation on how to do so (nor did we reverse engineer it) while developing the initial version of the emulator.

The following code shows how to display a blinking letter "A" at the current cursor position, which is then advanced:

```
.wait1      in a, (39h)
```

```
        and 20h             ; The AVDC ready flag...
        jr z, .wait1        ; ... must be 1.
.wait2  in a, (36h)
        and 10h             ; The AVDC memory access flag...
        jr nz, .wait2       ; ... must be 0.
        ld a, 'A'           ; Put the letter "A"...
        out (34h), a        ; ... into the character register.
        ld a, 1             ; Put the blink attribute...
        out (35h), a        ; ... into the attribute register.
        ld a, 0ABh          ; "Write character at cursor and advance".
        out (39h), a        ; Issue the command.
```

In addition to the aforementioned registers, the Partner also has a register containing flags that affect the whole picture generated by the AVDC. These flags are not fully documented, so we will only mention the ones implemented by the emulator:

- A flag that stretches "lit" pixels by an additional pixel to the right. This makes characters bolder, but the screen resolution remains the same. The BIOS sets this flag in 80-column mode and clears it in 132-column mode, but both states (0 and 1) work in both modes. This is shown by picture 3.1.
- A flag that inverts (bitwise negates) the picture; bright pixels become dark and vice versa.
- A flag that the emulator uses to distinguish between 80-column and 132-column modes, but is otherwise not understood. In the latter mode, characters are narrower.

Picture 3.1: The character set in 80- and 132-column modes with and without stretching.

The emulator stores the AVDC's memory and registers in arrays. The picture is drawn as on the actual AVDC (except that the processor is halted while drawing); for each row, characters are read starting with the address specified in the row table and for each column, a foreground color (the color of the actual character) and background color (which fills the rest of the rectangular character cell) is chosen. The character (or a half or a quarter of it, if the row is double-width or double-height) is then drawn (scaled and stretched as needed) in the cell.

The AVDC can also trigger interrupts. The Partner's BIOS enables only the vertical blanking interval interrupt. The emulator implements this interrupt, but not others, such as the interrupt that can be triggered when the AVDC starts drawing the first line of pixels of each character row, or the interrupt triggered when a command is processed.

3.5.2 GDP

The GDP (a Thomson EF9367) is capable of displaying graphics as well as text.

To do so, it is provided with 128 kilobytes of DRAM [1] containing two so-called pages of 1024×512 pixels. Each pixel is a single bit that can be "lit" or "dim". Software can write to either page while the GDP displays the other (or the same) page.

For better performance, a resolution of 1024×256 with non-square pixels (width to height ratio of 1:2) can be chosen (however, there are still only two pages, not four) and the so-called fast write mode can be used, in which screen refreshing is suspended as it otherwise generates constant read operations, during which writing is not possible.

The central processing unit does not have direct access to the framebuffer. It triggers write operations by issuing commands, of which there are three kinds:

- Line-drawing commands. The GDP implements Bresenham's algorithm in hardware [3]. To draw a line, the software simply sets the starting and ending point of the line and issues a command.
- Text-drawing commands. The GDP contains a ROM containing glyphs for all ASCII characters. Text-drawing commands copy these glyphs to the framebuffer and become a part of the picture. Text can be drawn horizontally (left to right) or vertically (bottom to top; the characters are rotated counter-clockwise by 90°). Regardless of the orientation, the text can be upright or italic and can also be scaled horizontally and vertically with independent integer factors from 1 to 16 [3]. (We could have copied the font in a similar way as the one used by the AVDC/DCGG, but there was no need to because it is shown in the GDP's datasheet.)
- Control commands for e.g. clearing the screen, switching between drawing mode and erasing mode, etc.

Commands are issued by writing them to a dedicated input/output port. Reading that port returns status flags. The most important flags are the ready flag and the flag telling whether a screen refresh is in progress.

Erasing previously drawn lines and text is done by choosing an "eraser" instead of the "pen" and then issuing the same sequence of drawing commands as used to

draw them. (This method is not perfect if the elements being erased intersect with other elements.)

The Partner implements another way: writing to the framebuffer can be done either in "normal" mode (where bits are set if using the pen and cleared if using the eraser) or in "XOR" mode, which applies a bitwise XOR operation to the bits (if using the pen, the bits are negated, otherwise they are left unchanged).

By using a command that temporarily allows direct access to the framebuffer, reading is also possible. The Partner allows reading a single pixel at a time. The software sets the coordinates, issues the command, then reads the bit from a special register implemented by the Partner's video board (not the GDP).

All other functionality is controlled through other registers, each of which has its own address. The registers are also used to pass parameters to commands:

- The first control register specifies whether the pen or eraser is selected, whether drawing is enabled (if not, drawing commands have no effect except moving the pen), whether fast write mode is enabled, whether cyclic screen mode is enabled (if it is, the upper bits of coordinates are ignored so that the pen is never out of bounds), and which interrupts are enabled.
- The second control register controls the text orientation and the line type (full, dotted, dashed, or dot-dash).
- The character size register specifies the scaling factors used by character drawing commands.
- The 8-bit ΔX and ΔY registers specfy the difference in coordinates between the starting and ending points of a line. They are unsigned; their signs are provided as part of the line drawing command.
- The 16-bit X and Y registers specify the starting coordinates of the line or character. The origin is in the lower left corner. These registers are automatically updated after each drawing operation; when drawing a line, ΔX and ΔY are accumulated to them, and when drawing text, X or Y (depending on the text orientation) is updated to contain the next character position.
- There are also registers returning the current position of the light pen, but the Partner does not support one.

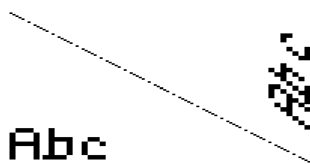
The Partner's video board has two additional registers affecting the operation of the GDP. The first specifies by how many lines the picture is scrolled down. The other contains flags for:

- which page is being displayed,
- which page is being drawn to,
- the drawing mode (normal or XOR),
- the resolution (1024×256 or 1024×512),
- the scroll mode; the lines scrolled out of the picture can either be displayed at the top instead or those lines can be left empty.

The following code shows how to draw a line and some text – the result is shown on picture 3.2:

```
            ld a, 18h           ; Select high resolution.
            out (30h), a
            ld a, 6             ; Clear the screen, X := 0, Y := 0.
            call GDPCmd
            xor a               ; A := 0; select horizontal upright characters.
            out (22h), a
            ld a, 43h           ; Scaling factors: 4× in the X, 3× in the Y dimension.
            out (23h), a
            call Abc            ; Print "Abc".
            call GDPWait
            ld a, 0Fh           ; Vertical italic characters, line type "dot-dash".
            out (22h), a
            xor a
            out (29h), a        ; Low 8 bits of X := 0.
            ld a, 100
            out (2Bh), a        ; Low 8 bits of Y := 100.
            out (27h), a        ; ΔY := 100.
            ld a, 200
            out (25h), a        ; ΔX := 200.
            ld a, 15h           ; Draw a line; ΔX positive, ΔY negative.
            call GDPCmd
            call GDPWait
            ld a, 20
            out (2Bh), a        ; Low 8 bits of Y := 20.
            ld a, 34h           ; Scaling factors: 3× in the X, 4× in the Y dimension.
            out (23h), a
Abc         ld a, 'A'
            call GDPCmd
            ld a, 'b'
            call GDPCmd
```

```
            ld a, 'c'
GDPCmd      call GDPWait
            out (20h), a          ; Issue a command.
            ret
GDPWait     push af
.wait       in a, (2Fh)           ; (GDP status register)
            and 4                 ; (ready flag)
            jr z, .wait
            pop af
            ret
```



Picture 3.2: An example of a line and text on the GDP.

An example of reading a pixel (X and Y registers must already be set, the GDPWait routine is the same as above):

```
            call GDPWait          ; If the GDP is executing a command, wait.
            ld a, 15              ; (direct framebuffer access command)
            out (20h), a          ; Issue a GDP command.
            call GDPWait          ; Wait until it is done.
            in a, (36h)           ; Read the register containing the pixel...
            and 80h               ; ... in the highest bit.
            ret                   ; A is 0 if lit and 80h if dim.
```

Emulating the GDP is quite simple. When the command port is written to, the command is interpreted and executed, and the other ports either read or set a register. Drawing is done into an array where each pixel occupies one bit. When refreshing the screen, the contents of the displayed page are drawn – for lit pixels this means that the intensity of that pixel is increased by some constant value. This overlays the GDP's picture on the AVDC's picture.

Rendering of the GDP's picture is skipped if fast write mode is enabled (which is not any faster than otherwise in the emulator, but disables screen refreshing as on actual hardware) or the page to display is empty. The latter is an optimization that

speeds up screen refreshing in non-graphical (text-only) applications. Whether the page is empty is kept track of by a per-page flag that is set when a "clear screen" command is issued and cleared when any drawing command is issued.

3.5.3 Displaying the picture on the screen

The AVDC and GDP each generate their own picture. These pictures are independent of each other. The emulator has to render the resulting frame combining both pictures into an array of equally sized square pixels. The question of what is the Partner's screen resolution is not easy to answer because:

- The AVDC generates (in theory) an arbitrarily-sized picture of 8×11-pixel characters. The number of rows and columns is adjustable; the Partner's BIOS uses only 80×26 and 132×26. This means 640×286 or 1056×286 pixels, where in the first case, pixels are also horizontally stretched. (The AVDC allows up to 128 rows and 256 columns; the emulator only supports the dimensions used by the BIOS.)
- The GDP generates a picture of 1024×256 or 1024×512 pixels; the pixels are only square in the latter case (the density is approximately 128 dots per inch in either dimension). This picture is always overlaid on the AVDC's picture so that intensities of overlapping pixels are summed.

If we take the maximum of each dimension, this gives us 1056×512, but in reality, the picture generated by the AVDC is somewhat larger (approximately by one row's height in each direction), meaning it must be scaled. This must be done without affecting aesthetics and without too much computational overhead, as the screen may have to be refreshed up to 50 times per second (for graphics, this is not a problem, because the pixels are either square or doubled in height).

We settled on a resolution of 1056×572 or 1024×572 pixels, if the host's screen is narrower than 1056 pixels. The scaling is done as follows:

- The height of the AVDC's picture is always doubled (hence the 572-pixel height).
- If the AVDC is displaying 80 columns, every other pixel is double-width; the AVDC's picture is then 960 pixels wide, which is still narrower than the GDP's picture, but most applications still look good enough. The picture is also centered horizontally.
- If the AVDC is displaying 132 columns and the screen is narrower than 1056

pixels, every fourth character is only 7 pixels wide instead of 8; the AVDC's picture is then 1023 pixels wide.
- The GDP's picture is doubled in height if the 1024×256 resolution is selected, otherwise it is not scaled. Either way, it is centered vertically because it is shorter than the AVDC's picture.
- If the screen is wider than 1024 pixels, the GDP's picture is also centered horizontally.

3.6 The serial interface

The Partner can have up to three serial ports conforming to the RS-232-C standard in the form of a DB-25 connector [1]. (Ours only has one port, which is standard in all configurations.) The keyboard is also attached to a serial port, but uses the DIN 5 connector.

The serial ports are controlled by two SIO controllers, each of which has two channels [16]. Each channel occupies two input/output ports [1, 16]; they are listed in table 3.1.

Name	Data	Status	Note
CRT	0D8h	0D9h	keyboard
LPT	0DAh	0DBh	present on all Partners
VAX	0E0h	0E1h	optional (along with MOD)
MOD	0E2h	0E3h	optional (along with VAX)

Table 3.1: Serial ports and their addresses.

In table 3.1, the first column contains the name of the port as referred to in CP/M. The second column is the data port; writing to it sends a character and reading from it returns the pending character if one is available [16].

The third column is the port used for reading status information and changing settings (e.g. the transfer speed). Multiple registers of the serial communications controller are accessible over this port [16]. The BIOS always leaves the status register selected after (re-)initializing the ports. Most applications only read from it, so the emulator implements only two bits of the status register that are necessary for applications to send and receive data; bit 2 tells whether the controller is ready

to accept a character to be sent, and bit 0 tells whether a character has been received (and not read yet from the data port). Other than that, the emulator does not implement any serial port functionality, but it does emulate certain devices that may be connected to it.

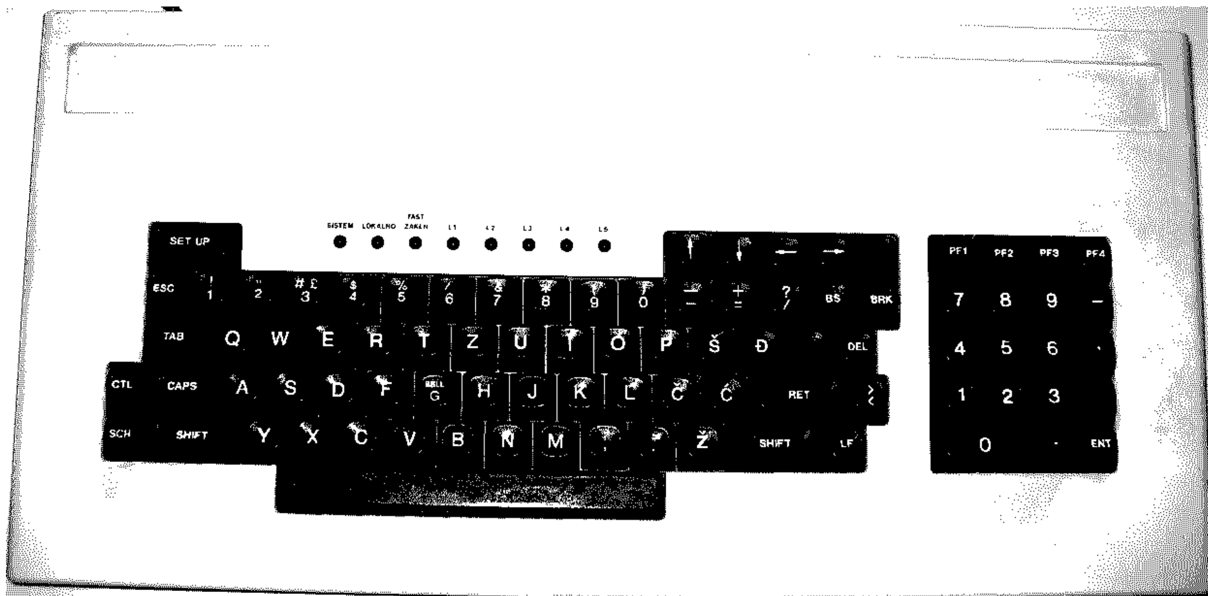
The BIOS checks the presence of the VAX and MOD ports (or rather the SIO controlling them) by setting one of their registers and reading it back. If it receives the expected result, it assumes that both ports are present.

The rutine SendByte in the hard disk imaging program shows how to send data over a serial port. Synchronous reading is done in the same way, except bit 0 of the status register has to be polled first.

3.7 The keyboard

The Partner's keyboard is connected to a serial port with the DIN 5 connector. It communicates with the computer at a speed of 300 bits per second [1]. It was produced by the company Gorenje.

In addition to alphanumerical keys and the numeric keypad, it has a few special keys that are not present on today's standard (PC) keyboards; they are shown on picture 3.3. The emulator maps those keys to PC keys in the closest positions.



Picture 3.3: The Partner's keyboard.

It also differs from a PC keyboard in that it is "independent". The scancodes produced by some keys vary depending on the state of SHIFT, CTL, and CAPS keys, and the latter do not produce any scancoedes themselves. The computer therefore cannot detect these keys being pressed or released, nor does it distinguish between e.g. the combination SHIFT+A and the A key being struck while CAPS is pressed. Each key press (but not release) triggers an interrupt to which the BIOS responds by reading the serial port's data port. If a key is pressed for a longer time, the same scancode is sent multiple times; the repeating is therefore implemented by the keyboard, not the computer. The scancodes are in most cases identical to the ASCII characters they represent, so software does not have to translate all of them, only those produced by function keys.

The keyboard also has eight LEDs (labeled, in Slovene, "SYSTEM", "LOCAL", "KEYB. LOCK.", and "L1" to "L5"; the meaning is not explained in the manual, but normally only the "SYSTEM" light is lit) and a buzzer that can produce a short or a long beep. Both the LEDs and the buzzer are controlled by software by sending 8-bit values:

- If the lowest two bits are "10", a short beep sounds, and if the lowest three bits are "100", a long beep sounds.
- Independently of this, the state of the LEDs also changes. Even though there are

eight of them, they are not simply mapped to the 8 bits of the value: if the lowest two bits are "00", the value is first decremented by 2, and if they are "10" or "11", the value is decremented by 1 – otherwise it is not modified. Only then are the LEDs set according to the values of the individual bits, where the lowest bit controls the leftmost LED ("SYSTEM") and the highest bit controls the rightmost LED ("L5").

Some values affect the behavior of the keyboard. The user can do so with the Set up program built into the BIOS, which can be brought up at any time with the SET UP key. (This is also where the user can switch the AVDC to 80-column or 132-column mode and invert its picture.) However, this program has no effect on our keyboard; there are apparently at least two models, each with its own set of commands. On the keyboard supported by Set up, the values are interpreted as follows:

- Bit 3, if set, disables the keystroke click.
- Bit 5, if set, disables automatic keystroke repetition.
- Bit 7, if set, selects the Yugoslav "QWERTZ" layout, otherwise the American "QWERTY" is used.

All of these commands require bits 0, 1, 2, 4, and 6 to be set, otherwise the keyboard's behavior is not modified (but in any case, the state of the LEDs is).

To provide a better user experience, the emulator understands the commands for the keyboard model assumed by Set up, even though we do not have such a keyboard. Most applications only read the keyboard (via BDOS functions) and never send anything to it, so the difference between models is not a major issue.

The Beep routine in the ROM-dumping program shows how to send data to the keyboard.

The buzzer also produces a very short beep (which sounds more like a click) on each keystroke, unless this is disabled by software.

When the emulator detects a key being pressed, it translates it into the appropriate keystroke and remembers it. It then triggers an interrupt (the manual told us which one) and plays a sound if necessary. The BIOS then responds by reading the serial port's data port, where the emulator passes it the scancode. If the user keeps the key pressed for some more time and automatic keystroke repetition is enabled, the emulator keeps triggering interrupts periodically until the key is released.

3.8 The mouse

Some applications developed specifically for the Partner support a mouse connected to the serial port. We do not have one, but the source code for the VIGRED program showed us what protocol it uses.

The program detects and initializes the mouse by sending it a byte with the value 99 and then tries to read 60 bytes. If the ASCII string "LOGI" is found in those 60 bytes, it assumes that the mouse is connected.

The mouse does not send data constantly, but must be polled by sending a byte with the value 80. The mouse responds with five bytes:

- The first byte tells the state of the buttons; bit 4 is the left button, bit 3 is the middle button, and bit 3 is the right button.
- The second byte contains the low 6 bits of ΔX in its low 6 bits.
- The third byte contains the high 6 bits of ΔX in its low 6 bits (two's complement).
- The fourth and fifth bytes are the same, but for ΔY. The Y coordinate's origin is at the bottom.

In the emulator, the mouse is implemented as a finite state machine. In the initial state, it waits for a command. When it receives the value 99, it enters to a state that returns the ASCII character "L" on the next read, then "O", "G", and "I", followed by zeros, because we did not know what data should be in the other 56 bytes expected by the software. The mouse then again enters the initial state. In a similar way, the value 80 causes the five bytes described earlier to be sent.

3.9 Printing

The emulator can redirect data sent to the serial port into a file. In this way, it is possible to "print" text files from e.g. the WordStar program.

This is not a perfect solution because most printers that were used with the Partner also support changing the attributes of the printed text (e.g. the font and its size) or even graphics. The emulator does not support any of that, but for most text files, this is not an issue.

3.10 Mass storage devices

All Partner models have at least one floppy disk drive. The WF/G model also has a hard disk connected to a Xebec S1410 controller. The ROM supports booting from both.

Because we wanted the emulator to reach a usable state as soon as possible (meaning that it could load CP/M and run at least a few applications), and also because we did not have any information about the floppy disk drive controller, we chose an alternative approach: instead of emulating these devices, we intercept calls to disk-related BIOS functions. This is an example of virtualization at the driver level [13].

CP/M always calls the BIOS through documented entry points, so the emulator checks the PC value before executing each instruction. If it matches any of the entry points into BIOS disk-related functions, the emulator performs the requested operation and returns the RET instruction, which prevents the actual BIOS code from being executed. The disk-related functions are the following [7]:

- HOME. Sets track 0 as the track to be used for the next disk I/O operation.
- SELDSK. Selects the device to be used for subsequent disk I/O operations.
- SETTRK. Sets the track to be used for the next disk I/O operation.
- SETSEC. Sets the sector to be used for the next disk I/O operation.
- SETDMA. Sets the 16-bit address of the buffer that will be filled with data from the next sector read or that will be written to the next sector written.
- READ. Reads the sector specified by earlier SETTRK and SETSEC calls from the device specified by an earlier SELDSK call to the buffer specified by earlier SETDMA and SETBNK calls.
- WRITE. Writes data from the buffer into the sector; the parameters are the same as for READ.
- SETBNK. Sets the bank containing the address given to SETDMA.

This method mostly works because CP/M is the only operating system for the Partner found so far and most applications do not access these devices directly, but only through CP/M. The most notable exceptions are the WF and DISKETTE programs, whose purpose is exactly to test the hard disk and its controller, and the currently inserted floppy disk, respectively, and the FORMAT program, which formats floppy disks.

The ROM also accesses these devices directly (as it otherwise cannot begin loading the operating system), so the emulator also intercepts some of its routines.

Other devices could also be faked with this method, but in the interest of accurate emulation, we did not use it unless necessary.

3.11 The real-time clock

To provide the operating system with the current date and time, the Partner contains a real-time clock powered by a battery.

The clock is accessible as a set of registers containing temporal components (milliseconds, seconds, minutes, and the hour, day, month, and last two digits of the year) in "packed BCD" format. Each register has its own input/output port through which it can be read and written at will.

The emulator obtains the date and time from the host operating system. Because the Partner's BIOS and CP/M do not support dates beyond the year 1999 (due to bugs, those dates are not displayed properly; this could be patched), the emulator subtracts a multiple of 28 from the year to make it lower than 2000 (e.g. 2017 – 28 = 1989). The Gregorian calendar repeats every 28 years; for example, July 1, 2017 was a Saturday and July 31, 2017 was a Monday; it was the same in 1989. (This solution will work until, and including, February 28, 2100, unless the host operating system has some other limitation.)

The real-time clock also has a small memory used to store user preferences set by the Set up program. The emulator reads the contents of this memory from a file on startup (or uses default values) and saves it on exit if it was modified.

If the battery runs out and the memory loses power, its contents become invalid; all bits are read as ones. Four of those bits are used by the BIOS to remember the default character set. This value is normally between 0 and 8; when we first turned the computer on, it was 15 (binary: 1111), which caused the wrong characters to be written into AVDC memory; the BIOS does not check whether this value falls into the allowed interval. Picture 3.4 shows the effect of this bug. The Set up program thankfully looks correct regardless of the selected character set, so we were able to fix this problem on day 1 by selecting the Yugoslav character set.

```
A>  vic

PhysiculrD vic s:r
IwInpde,OwOdepde,§w§ aiul,XwXrn-Xrff
CRTrrrr9600rrIO§rrrrLPTrrrr§800rrIO§Xrrrrrrrrrr9600rrrrrrrr
rrrrrrr300rrrrrrrrrrrrrrrrrNONErrrrrrrrrGDPrrrrNONErrOrrrrrr


Cdaa nerAssignm nes:r
CONIN:rrwrCRTrrrr
CONOUT:rwrGDPrrrr
AUXIN:rrwrLPTrrrr
AUXOUT:rwrLPTrrrr
L§T:rrrrwrLPTrrrr

Ene arn !russignm nerrarhierRETURNr

A>device

Physical Devices:
I=Input,O=Output,S=Serial,X=Xon-Xoff
CRT    9600  IOS    LPT    4800  IOSX          9600
       300                 NONE         GDP    NONE  O


Current Assignments:
CONIN:  = CRT
CONOUT: = GDP
AUXIN:  = LPT
AUXOUT: = LPT
LST:    = LPT

Enter new assignment or hit RETURN
```

Picture 3.4: Text on the screen after the first boot and after selecting a character set.

3.12 Other devices

There are a few more chips in the Partner that we do not emulate:

- The DMA controller. It is used for transfers between the working memory and the

floppy disk drive controller or the hard disk controller. Since we do not emulate the latter, we also do not emulate the former. It is not known whether it has any other use.
- The CTC. It is used in the same situations as the DMA controller. It could theoretically also be used by applications for their own needs, but we did not observe this.
- The parallel port controller (PIO). Our hard disk contains some source code that sends and receives data through it, but we do not know what devices it was used for in practice.

Chapter 4

Evaluation of the final product

In this chapter, we will assess the accuracy and correctness of the emulator and list possible improvements.

4.1 Accuracy

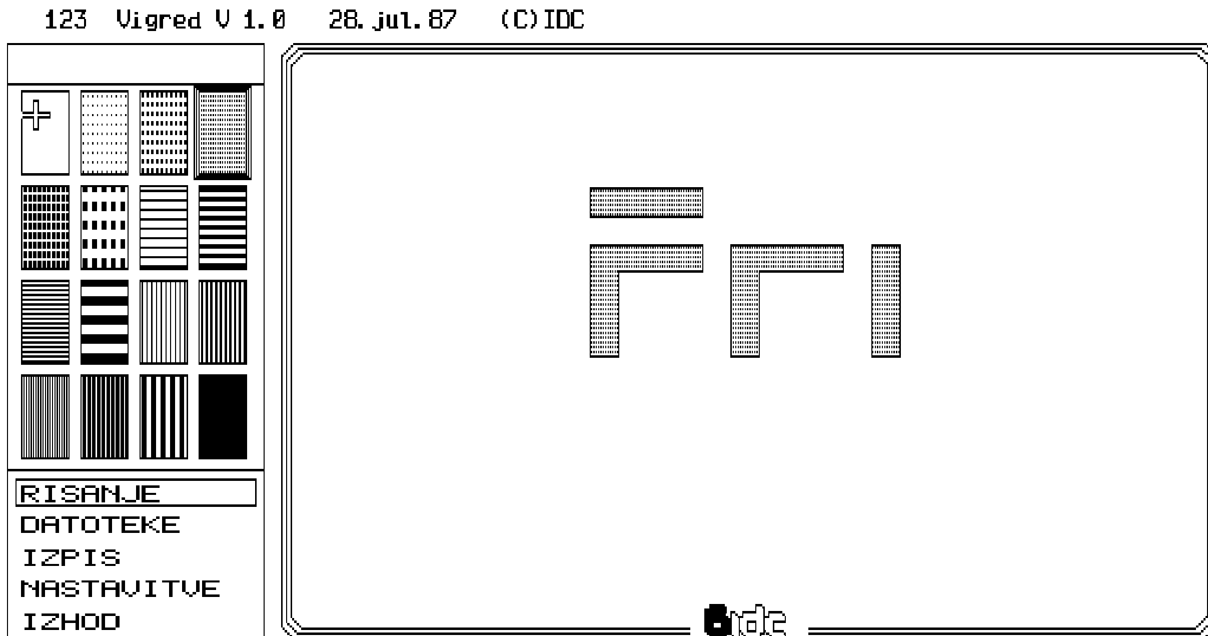
In general, we tried to make the emulator behave as close to an actual Partner as possible, unless the amount of effort required made that infeasible. From a user's perspective, applications should behave the same as on an actual Partner even though the emulator differs in certain details. Picture 4.1 shows the VIGRED application, which required some more work than others before it ran entirely correctly.

These details are mostly time-related and are a consequence of the discrepancy between actual hardware, which runs in parallel, and the emulator, where everything is done sequentially for simplicity. This is most evident in the emulation of the AVDC and GDP, where changes in memory and registers only take effect periodically instead of immediately, because the emulator does not simulate a cathode ray tube monitor.

The other time-related difference are the various delays, which are also a consequence of the parallel operation of hardware. The AVDC, GDP, and other devices do not execute most operations immediately (from the processor's perspective, "immediately" means faster than the processor can execute the instruction following the one that caused the operation on the device), but only after a certain time, and the processor is notified of this either by an interrupt or through a status register that must be checked periodically. DRAM-type memory also incurs delays on each read and write and must be refreshed periodically, which temporarily prevents the processor from accessing it.

In the emulator, all these operations take effect immediately: when the processor accesses a device, the corresponding function is called to perform the operation, and only then is code execution resumed. This could be fixed, but is not crucial;

after all, (properly written) software should not depend on such details, since the documentation for most devices does not even mention the duration of such delays; the only way to find the duration is empirically by measuring it.



Picture 4.1: The VIGRED application in the emulator.

4.2 What could be improved

To achieve "perfect" emulation, we would also have to emulate devices that we currently do not – they were listed in the previous chapter.

The following functionality is also desired:

- Emulation of a graphics tablet. We have source code that communicates with it; it might be helpful in the same way as the mouse-related code was.
- Emulation of at least one of the printers supported by Iskra Delta's applications.
- Emulation of at least one of the plotters or graphics-capable printers supported by Iskra Delta's applications.

Some non-crucial features would make using the emulator and exchanging data

easier, such as:

(Note: these features were later implemented in PartEm 1.5, released in 2021.)

- integration with the host's file system,
- importing files into disk images; currently this has to be done manually with a hex editor,
- easier exporting of files from disk images,
- copying and pasting text between the emulator and the host.

4.3 The user experience

Because the Partner was not used by many people and their number will not increase in the future, it makes sense for the emulator to try to mimic an actual Partner in terms of aesthetics. It does this by using sound recordings of the hard disk, keyboard, power switch, and reset switch. The picture on the screen also starts out dim and slowly reaches full brightness, while the sound of the hard disk spinning up is played.

4.4 Portability

An emulator that runs on only one or a handful of operating systems is not very useful, as it is then itself in danger of not being able to be easily run in the future.

Portability is achieved by using the C programming language (which can be compiled practically anywhere, and the language also lends itself well to emulator development) and the SDL (Simple DirectMedia Layer) library [6]. The latter provides a consistent interface for working with input and output devices (for example: reading the keyboard and mouse, playing sound, drawing to the screen) on several operating systems. The emulator can also be compiled with multiple compilers for multiple operating systems and architectures.

It can also be compiled with the Emscripten compiler, which translates C code to JavaScript [12]. Emscripten also implements a large part of the SDL interface. This way, the emulator can run inside a Web browser on the domain matejhorvat.si, but this requires much more computing power than if it is compiled to machine code.

Chapter 5

Conclusion

At the start of the project, we were not sure that we would be able to produce a working emulator due to hardware problems (floppy disk drive, serial port) and a lack of documentation. With some luck and reverse engineering, we got quite far; from a user's perspective, the emulator behaves almost identically as an actual Partner.

Of course, there is room for improvement; as mentioned in the previous chapter, some devices are not emulated at all and applications only work because they do not access them directly or not at all, and some devices only have the minimum amount of functionality implemented for most applications to run without issues.

The project took about four months. The emulator itself was written in approximately one month, but the development took turns with research and reverse engineering. It is made up of just over 3500 lines of code. This does not include the Zymosis and SDL libraries.

As part of the project, an additional 1000 lines of code were written in various languages. This includes e.g. the hard disk imaging program and ROM-dumping program, decoding these dumps, and various experimental code used to reverse engineer the hardware.

The results are available on the Web site matejhorvat.si.

With a similar approach, it should also be possible to emulate other Slovenian computers, such as the Dialog and Triglav.

(Note: since this thesis was first published, work on them has already started.)

References

[1] Iskra Delta. Partner WFG, 2FG, 1FG – priročnik za uporabnike. 1987.

[2] Signetics. SCN2674/SCN2674T Advanced Video Display Controller (AVDC). Available at: http://www.datasheet4u.com/datasheet-pdf/Signetics/SCN2674/pdf.php?id=524408, 1987. [Accessed on May 1, 2017]

[3] SGS-Thomson Microelectronics. EF9367 MOS graphic display processor (GDP). Available at: http://www.datasheet4u.com/datasheet-pdf/SGS-Thomson/EF9367/pdf.php?id=604475, 1988. [Accessed on May 1, 2017]

[4] KryoFlux. Available at: https://www.kryoflux.com/. [Accessed on December 12, 2016]

[5] Zymosis. Available at: http://repo.or.cz/w/zymosis. [Accessed on April 8, 2017]

[6] Simple DirectMedia Layer 1.2.15. Available at: https://www.libsdl.org/download-1.2.php. [Accessed on April 2, 2017]

[7] CP/M information archive: BIOS. Available at: http://seasip.info/Cpm/bios.html. [Accessed on April 20, 2017]

[8] CP/M information archive: BDOS system calls. Available at: http://seasip.info/Cpm/bdos.html. [Accessed on April 20, 2017]

[9] HxC Floppy Emulator. Available at: http://hxc2001.com/download/floppy_drive_emulator/. [Accessed on March 28, 2017]

[10] Digital Research Source Code. Available at: http://www.cpm.z80.de/source.html. [Accessed on April 2, 2017]

[11] Digital Research. An introduction to CP/M features and facilities. 1978.

[12] Emscripten. Available at: https://kripken.github.io/emscripten-site/. [Accessed on July 1, 2017]

[13] J. E. Smith, R. Nair. Virtual Machines – Versatile Platforms for Systems and

Processes. Morgan Kaufmann Publishers, 2005.

[14] A. Vilfan, J. Vilfan. Iskra Delta med zahodom in vzhodom. Bit, 1984, issue 1.

[15] J. C. Nichols, E. A. Nichols, P. R. Rony. Z-80 Microprocessor Programming & Interfacing: Book 2. Howard W. Sams & Co., 1980.

[16] L. A. Leventhal. Z80 assembly language programming. Osborne/McGraw-Hill, 1979.

[17] A. P. Železnikar. Evropska mikroračunalniška industrija. Informatica, 1984, issue 1.