# OpenMSX internals

Wouter Vermaelen

June 14, 2002

# Contents

# Chapter 1

# Analysis

## 1.1 When does a device need to be emulated

A MSX computer has several components: a CPU, audio and video devices and many others. All these devices need to be emulated.

In a real machine all devices work in parallel all the time. Of cource an emulator can only emulate one device at a time. This creates some problems:

First problem, some devices need to emulated *constantly*, e.g.

- screenrefresh by the VDP
- soundbuffers for soundcards

We can approximate this by emulating those devices very regularly.

Second problem, devices can communicate with each other. When two emulated devices wants to communicate we must make sure these devices have advanced equally far in time. Example:

CPU changes by means of I/O-commands the palet settings of the VDP on time T, therefore the VDP must calculate its screen up until time T before the IO command is executed. From that moment on the VDP will use the new palet data when calculating the rest of the screen.

So we need some sort of synchronization mechanism.

## 1.2 Communication in MSX

### 1.2.1 General principles

In a MSX communication happens only between the CPU and another device, never between two non-CPUs. All communication is initiated by the CPU, except for interupts. Communication happens at a specific moment in time but is not instantaneous.

### 1.2.2 (Un)predictable communication

Communication can be either predictable or unpredictable:

- Predictable communication: Most interupts are predictable, example:

  The VDP knows: 'if my settings remain as they are right now, I will generate a VBLANK interrupt at time T'

- Unpredicatble communication, examples:
  - CPU executes an IO instruction
  - CPU executes an instruction who reads from or writes to memory (!)
  - an external interrupt (vb RS232)

Note that *most* unpredictable communication is initiated by the CPU, this is important for some optimizations (see next chapters).

### 1.2.3 Communication with multiple devices at once

Important example: memory mappers (IO 0xfc ... 0xff).

Writing is not a problem but reading is, since the result is not defined in the MSX standard. But some programs expect a certain behaviour...

### 1.2.4 Non-standard behaviour

A device its memory is addressed by slot and address signals. Normally a devices should check both these signals. Some devices (e.g.: ????) ignore slot-select and respond whenever they see their own address.

# Chapter 2

# Design

## 2.1 Communication

### 2.1.1 Initiated by the CPU

The CPU can read or write to a memory location or to an I/O port. The communication happens at a specific moment in time and might take a (short) time to complete. So besides to obvious parameters (adres/port, value) and return values (byte read), we also need to pass and return a timestamp.

### 2.1.2 Initiated by other device (interupts)

#### General

All interupts must be planned beforehand (by synchronization points, see next section). This is easy for predictable interupts, but impossible for unpredictable interupts.

When a device raises an interupt it must pass control to the emulated CPU before it can further advance in time.

#### Unpredictable interupts

There are two possible solutions:

1. Polling. Bij deze oplossing zetten we zelf de SP om de bv 50 HZ T-states. Blijven we bij het voorbeeld van de RS232 dan betekend dat er om de 0.02 seconden gevraagd gaat worden of er een byte klaar staat. Als dit het geval is dan kan de ge-emuleerde RS232 zijn volgende SP plaatsten binnen bv 100 T-States en deze tijd langzaam laten toenemen als blijkt dat er geen byte meer ontvangen is.

   Methode heeft het voordeel dat je zelf een vorm van snelheid begrenzing kunt inbouwen door een apparaat steeds maar een lage hoeveelheid

interrupts te laten genereren. Tevens zou op deze manier het mogelijk eenvoudiger zijn om bv een RS232 connectie te faken aan de hand van een file met een 'opgenomen' connectie. Bij beide methodes moet het device er voor zorgen dat het er rekening mee houd dat een geregistreerd SP wel eens tijdens een DI zou kunnen vallen en dat daarom een gesette interrupt niet noodzakelijker wijze word afgehandeld. Dit wil zeggen, er kunnen bv drie SP geregistreerd worden die elk een setInterrupt() uitvoeren maar dat slechts na de derde setInterrupt de CPU deze pas afhandeld!.

2. Converting van Guest OS interrupts naar MSX interrupts Het zou ook mogelijk zijn om als het guest OS een interrupt krijgt dat deze kan ingelast worden in de datastructuur met de SP, aangezien deze al in staat is om SP's tussen te lassen in de bestaande structuur zou dit geen probleem zijn.

### 2.1.3 Communication with multiple devices at once

Most important example: memory mappers (IO 0xfc ... 0xff)

For this kind of behaviour the most usefull principle would be a master-slave approach In het geval van memmorymappers heb ik dit als volgt reeds geprogrammeert:

- De classe MSXMemMapDriver(=master) registreerd zich bij het MSX-Motherboard voor de poorten OxFC .. 0xFF.
- De Driver houd per slot/sublsot bij wat de mappergrootte is. Tijdens de initfase vervangt de MapperDriver alle verwijzingen naar hem in de slotstructuur door verwijzingen naar verschillende MemMapSlaves.
- Bij OUT activiteit zal de driver de gepaste slaves vertellen dat ze nu data moeten lezen/schrijven op andere plaatsen in mapper van hun main-/sub-slot combinatie.
- Bij IN activiteit is het de mapper die de waarden kent en de juiste value aan de CPU terug geeft. Op deze manier is het simpel om de pull-up van de Turbo-R te emuleren, of de conflicten van het lezen van meerdere mappers in de driver te implementeren.

Dit idee ga ik waarschijnlijk ook gebruiken bij de CPU, op die manier heb je een CPU-master en twee CPU-slaves bij de turbo R. De master zal zich dan ook registreren bij het MotherBoard om zo op de CPU-switch poort-van de Turbo-R te kunnen luisteren en de juiste taakdelegatie te kunnen doen.

## 2.2 Synchronization

### 2.2.1 Time-active and time-passive devices

#### Time-active devices

This are devices that can request to be emulated at a future moment in time, such a moment is called a *synchronization point*. The reasons for such a request are:

- Some devices need to be emulated regularly, e.g.:
  - VDP moet regelmatig zorgen voor beeldopbouw
  - soundchips moeten regelmatig nieuwe samples klaarzetten
- Some devices can raise an IRQ, these IRQ must be announced beforehand with a synchronization point. This is easy for predictable IRQ's, but must be estimated for unpreditable IRQ's (see section 2.1.2).

#### Time-passive devices

These devices can be emulated at the moment the CPU tries to communicate with it. Example:

- Keyboard
- RAM/ROM

### 2.2.2 Scheduler algorithm

A few rules:

1. geen enkel device mag in uitvoering 'voorliggen' op de CPU. Dit omdat praktisch alle onvoorspelbare communicate (IO en memory) uitgaat van de CPU. Als een device op een bepaald moment voorligt op de CPU en als nadien de CPU tijdens zijn 'inhaalbeweging' communiceerd met dat device, ... ja dan klopt er iets niet.
2. elk device moet zijn eerstvolgende synchronisatie punt aan de scheduler melden.

Now the algorithm:

1. De scheduler neemt het vroegste synchronisatie punt, en *probeert* de CPU tot op dat punt te brengen.
2. Als dit lukt
   (a) dan wordt ook het device dat dit synchronisatie punt had gezet tot hier gebracht (dit lukt altijd)

(b) als het niet lukt (CPU heeft een IO of een memory instructie uitgevoerd). Dan wordt eerst het device waarmee gecommuniceerd werd bijgebracht (nu niet volledig tot aan synchronisatie punt, maar tot waar de CPU gekomen was) en dan wordt pas de IO of memory opdracht uitgevoerd. Het zou kunnen dat het device een (nieuw, vroeger) synchronisatie punt heeft gezet, dus het algoritme begint weer van vanvoor.

Note a device may not register a new SP that comes in time before the CPU's current time. (Step 2a, 2b)

# Chapter 3

# Implementation

## 3.1 Timetracking

Every device has its own native clock frequency, for the emulation of one particular device it is easy to work with native T-states. But for the global emulator it is easier to work with some least-common-multiple frequency.

The class 'Emutime' offers such an interface: devices work with native T-states, but they are internally converted to global emulator T-states.

All operations involving time should go through this class.

## 3.2 The implementation of the scheduler in openMSX

The MSXScheduler will keep track of the T-states. This 'scheduler'-devices is 'owned' by the MSXMotherBoard. The MSXScheduler is a generic object that can be reused by other devices. If for example someone would like to make the VDP as a collection of internal devices each withs it's own function and needs some internal scheduler ...

### 3.2.1 Scheduler and timetracking

The MSXScheduler keeps a simple list for timescheduling. This is a simple list of the next SP and the device that has set this SP. The list is ordered so that the first element always is the first SP that has to reached. The scheduler will follow this simple procedure:

1. Set the target T-State of the cpu to the first SP in the list.
2. Let the CPU execute until the target T-state.
3. Get the device from the first SP in the list and let it reach its T-state.
4. Remove the first elemnt from the list

5. Goto step 1

Attention. The first element of the list in step 1 could be different from the first element in step 3. This is because of the fact that during step 2 the device can do some I/O to other device who can set a new SP that has a smaller value then the set target T-state. For example, enabling line-interrupts from the VDP could change the first SP to be reached. For this reasson the scheduler, when inserting a list element before the first element, will call the setTargetTStates of the CPU. this will garuantee that the CPU will execute until the first SP element of the list.

Note 1: To get the extra time-info during communication between CPU and devices the calls (readMem,writeMem,readIO and writeIO) pas as their latest parameter the current T-state of the CPU. This makes it possible for devices to "catch up" with the processor if needed. So the graph of Wouter could be implicetly included in the emulation by passing the current time around during device communication. Note 2: Also non-msx devices could be integrated in this list. for example the routine that draw the actual screen on your PC could be called from the scheduler. Also you could make a sort of dummy device that would be called each 1/50 second and would block emulation until there is really 1/50 second past between it's previous execution and the current one.

### 3.2.2 Device Sync points

Since each device has its own natural clockfrequency the scheduler will run at the speed of the lowest common multiple. In case of an MSX2 the CPU is at 3.5 MHz, the VDP runs at 21 MHz. Therefore the scheduler runs at 21 MHz.

However we could make a MSXDevice that uses a clockfrequency of 42 MHz. In this case the scheduler will need run at 42 MHz, however the CPU still needs to run at 3.5 MHz and not at 7 MHz. To overcomme this problem the scheduler has a method called getTimeMultiplier(nativeFreq). You can pass your native frequency and it will tell you by how much you need to multiply the devices native T-states to get to the frequency of the scheduler.

In the first case this would result in the value 6, in the 42MHz case this would return 12. If you ask the scheduler for the current T-state you will need to divide them by this value to get to the native T-state value

The most used SP setting is from "now" until x T-states later. This can be achieved using the method setLaterSP(getTimeMultiplier(nativeFreq) * numberOfNativeTStatesLater , this ).

## 3.3 For interrupts itself

If a device generates an interrupt it should call the MSXMotherboard.raiseIRQ() method. when a device doesn't activate the interrupt line any longer it will call the MSXMotherboard.lowerIRQ(). The emulate CPU will after each instruction check the IRQ-line and respond accordingly (EI/DI, interrupt-mode,...). In the Motherboard the IRQ-line will be implemented using a counter, if irq-counter is zero then no interrupts otherwise the irq-counter should contain the number of devices that are generating an IRQ.

*A device isn't allowed to call the raise(lower)irq twice in a row!!*

The MSXDevice classe contains a standard method setInterrupt() and resetInterrupt() which take care of this restriction and call the MSXMotherBoard if needed.

We could make it mandatory to set the interrupt as an SP. In this way we would only need to test if the IRQ-line is active each time we call the CPU from the scheduler. However for most devices this wouldn't change anything, they still need to emulate up until the SP point and call the raiseIRQ(). Any comments ??

# Chapter 4

# Implementing devices for openMSX

Before you start implementing a new device for openMSX make sure you have read and understand the way openMSX handles eumTime. If your device is able to generate an IRQ you must follow the rules described above.

## 4.1 The life cycle of an MSXDevice

### 4.1.1 In the beginning

In the beginning there was nothing.... And then a lot of things happened and after a few billion yeas it all resulted in openMSX. And internally in openMSX also a lot of things need to happen before you get a running emulator. This is what happens before a device becomes aware of its existance.

- The main loop creates an MSXConfig object and let it parse a configuration file.
- A MSXMotherBoard is create, this object can be viewed as the bus of the MSX and keep tracks of all the devices and slot layouts.
- The devicefactory creates the objects that are indicate by the MSXConfig object. It loops trough the objects that are summed up in the configuration, each time creating an object of the indicated class and adding to the deviceslist of the motherboard. Each MSXDevice also gets a pointer set towards it personal configuration object. For people who need a more vivid description of what has happended up until now: We just create all the needed chips and PCB's and tossed them all on a big pile.

The life cycle of an MSXdevice follows the following basic steps

- Object is instantiated

- The device reads its configuration and prepares itself to run
- The device waits for calls made by the scheduler/CPU.
- When destroyed the object nicely releases it sources.

### 4.1.2 Instantiation

The life cycle starts with the birth of the object. During this constructor fase the MSXDevice can set some of its internal parameters to default values. The general rule being that this initialisation must be fail-proof. Meaning that during this initialization we can not try to malloc some memory or read a file or something else that could go hay-wire. By following this simple rule we can safely asume that the device factory can contruct all of the desired devices with out bailing out. If we need to bail out this will result in the termination of the entire emulator. If for instance the total system memory isn't enough to even create an object.

### 4.1.3 Initialization

Alsmost imediately after the device is instantiated, the first method that will be called is the setConfigDevice(MSXConfig::Device*). This will allow the MSXDevice to read its own configuration during the following initialization. Remember that the instantiation should be simple and fail-safe. Now it is time to begin the more dangerous internal set up. At this point all needed devices are instantiated, only there configuration isn't complete yet. Now the motherboard start calling the init() method of all of it's known devices. Know you can try to read files, allocated memory and (since all devices are existing) trying some initial communication with other MSXDevices. Be aware howver that this device itself may not have yet been initialized.

During the initialisation the device should register its place in the slot structure with the MSXMotherBoard. Also registering I/O ports is done at this point.

One of the reasons to have this method seperate from the instatiation is the fact that later on we could decide that each device should be able to be re-initialized during run time. It would be advicable to write your code so that this is possible. For pointers this would mean that the contructor makes them NULL pointers and that the init should check if the pointers is empty before trying the malloc, and constructs such a like.

### 4.1.4 Start

One of the first things the device should do in his start method is asking the motherboard for the actual emutime. Remember that it should be possible to insert a device during the emulation. So it isn't smart to assume that

the start method will be called when emutime is still zero. This is ofcourse only needed if the device uses time ˆ_Ôuring this start the device sould also set it's synchronisation points if needed. ex. a VDP should register it's first following interrupt.

### 4.1.5 Stop

### 4.1.6 Reset

Remark 1 : creating of MSXDevices ———————————— the sequence in which MSXDevices are used is as follows –if ¡slotted¿ are embeded in the ¡device¿-block this device will be visible in the slotstructure of the emulated MSX.Multiple ¡slotted¿blocks are possible f.e. a simple 64KB RAM expansion is visible in one slot,one subslot and the 4 pages of it. –the¡ parameter¿ block is parsed and every key,value pair is passed to the setParameter methode. This method is used to set internal parameters of the MSXDevice, f.e. the amount of vram could be passed to a VDPdevice, or the number of mapper pages for a MemMapper. -once athe configuration file is parsed and all devices created, all devices in the deviceslist have their init() method called. In the init() method the devices can allocate desired memmory, register I/O ports using the register_IO_Ini/IO_Out method of the MSXMotherBoard. -at last the virtual MSX is started, the motherboard sends a reeset signal to all devices and starts emulating, if the user requests a reset, the reset method is called for all devices and the MSX comenses al over. The reset method is used for warm-boot, so for example memory should NOT be cleared from most memory devices.