

How to create an application for FutureOS

In this article we will learn how to program an application for FutureOS. In contrast to other operating systems (OS) this is relatively easy.

Generally speaking you can use machine language / Z80 assembler or C to code an application for FutureOS. Assembler is the means of choice in this case. This is because FutureOS is a very fast OS and therefore it's nice to have quick apps too.

You can pretty much use all kinds of assemblers to write any kind of FutureOS application. This can be a Z80 assembler running on the CPC or a cross assembler running on any other system (like a Windows PC). Personally I do use the MAXAM assembler from Arnor, because it runs very stable; is pretty much error-free and can use very huge source code files. In addition it contains an editor aside of the assembler. Therefore the entered source code can directly be assembled and executed. The disadvantage of MAXAM is that it's slow. However you can use a CPC emulator in Turbo-mode – especially as soon as the source code reaches some KB or gets even bigger.

How to assemble an application for FutureOS using the Maxam assembler?

- Load or enter the source code. Please remember to save your source everytime.
- Assemble, for example at target address &9000. This address can be used to test generated code directly in RAM.
- Still working under Maxam enter the RSX command |FDESK. This will start FutureOS. However the command |FDESK will also preserve the first 48 KB of the main memory inside of the expansion RAM (E-RAM).
- Now you can start your generated code directly in RAM.
Either you click on the RUN icon (or use the hot key X instead); subsequently press 3 (for RAM application) and enter the start address (&9000 in our example); finally enter the RAM configuration – that's &C0 for the main RAM of the computer.
Or (instead of using the RUN icon) you can launch your application directly from inside the machine monitor.
- Now the application is being executed. After it finished execution it will return to the Desktop of the FutureOS (or the menu of the machine monitor).
- Eventually you will quit FutureOS; therefore you can use the END icon.
- Now you can continue to use Maxam to work at your Source Code.

Attention: Using Maxam 1½ allows you to load your source code way more quick. This is due to using PROTEXT as text editor.

What's important at application development stage?

As long as you develop and test an application it's probably the best to assemble it directly into RAM instead saving it to disc and executing from there.

As soon as the application reaches final stage you will of course save it to disc and start it from there. In addition you can put your application in an expansion ROM and execute it directly out of the ROM. But that's a topic for another article.

If you assemble an application directly into RAM and execute it there you need to take some things into account: When starting FutureOS some bytes in RAM will be changed. Therefore you can't assemble your program to any address in RAM.

The following parts of the RAM will be altered when starting up the OS:

- Address &0038: it will contain byte &C9 (Code for RET). Therefore the masked interrupts are blocked.
- Address &0066 and &0067: they will contain the bytes &ED, &45 (Code for RETN). Also the nonmaskable interrupts are blocked.
- The block from &A000 to &FFFF will be deleted/initialized using byte &00. The file-tagging-bytes (FTBs), the text screen buffer and the system variables of FutureOS are located beginning at address &A000. Please see Manual and file '#OS-VAR.ENG'

The remaining RAM between &0000 and &9FFF as the expansion RAM (E-RAM) can be used freely in principle.

However, in case you start your application from a mass storage medium then you can use the main RAM between &0000 and &B7FF. That is 46 of the main memory.

Testing your application:

Well, FutureOS does use the lower 16 KB RAM block (&0000 to &3FFF) as buffer for sorting DIRectories. But appropriate OS functions save the lower RAM into a 16 KB block of the E-RAM. So the sorting of DIRs shall not be a problem.

However, there must be a character set being located at address &3800 reaching up to &3FFF. It doesn't matter if the character set is located in the lower ROM or RAM. The ROM contains it anyway, but you can use your own character set too. In the latter case you just put your own character set inside of your application at address &3800.

As long as you just test your application it's probably better not to use a start address smaller than &4000 when assembling your source code.

Furthermore most of the assemblers available for the CPC keep their source code at the beginning of the RAM. Maxam for example stores its source code beginning at address &0170. Therefore you shouldn't pick a too low start address for testing your code.

The 16 KB block between &4000 and &7FFF is used for E-RAM banking! Well, E-RAM banking is important only for few applications I guess. To be on the save side it's better to assemble your application at address &8000 or higher, just in case your application is supposed to deal with E-RAM.

Usually it makes sense to assemble an application to address &8000 or &9000 for testing purposes. The region between &8000 and &9FFF is usually free and the OS will usually not touch it. The only exceptions are file operations dealing with large amounts of data for the E-RAM. These 8 KB provide enough space for developing and testing a fully grown application written in Z80 assembler.

What to consider if your application will load more files?

When loading (or saving) a file (data, code, whatever) into (from) the E-RAM the block between &8000 and &8FFF sometimes serves as sector buffer. But this is only the case when

loading (saving) bigger (>16 KB) chunks of data at once. In this case it's best to assemble your application to target address &9000. You still can use 4 KB (&9000 to &9FFF)

The block between &9000 and &9FFF will never be touched by the OS. Here you can always assemble your program, call the OS, then maybe load additional data and eventually execute it at &9000.

Well, yes the 'save block' from &9000 to &9FFF is 'only' 4 KB in size, but to generate that amount of code you already deal with a big source code file!

Your application can store additional data or variables at other areas in RAM. In this case somewhere below &9000.

All variables must of course be initialized by the application itself. In case you leave the interrupts switched off and you're not going to use the RST commands you can freely use the RAM beginning at address &0000 on upwards. Furthermore you can freely use the block between &A000 and &B7FF to store data or whatever.

When creating applications bigger than 4 KB (which means that the source code is already big) the assembler should put the created code to disc and not to RAM. In this case the application can again use the full 46 KB of the main RAM (&0000 to &B7FF). Subsequently such code can be loaded and executed under FutureOS. If you don't want to overwrite the file-tagging-bytes then just use the main memory area between &0000 and &9FFF. In this case you can use 40 KB including your own interrupt handler and all other RST vectors if you want. If your application uses the RST vectors for itself and has its own interrupt handler it will speed up significantly.

In case you need more RAM then you can use the E-RAM. Take a look at the XRAM_?? variables to see which block are free for your application. You can reserve / allocate as much E-RAM as you want. Also take a look at the manual and the file '#OS-VAR.ENG'.

You can also use the area between &A000 and &AFFE, as a buffer for data for example. This block usually does contain the file-tagging-bytes, they show if a file is tagged (for operation) or not. Working with discs (especially reading Directories) will alter this block of 4 KB. This area will not be changed by the OS as long as no disc operations take place.

In addition you can temporarily use the area between &B000 and &B7FF as storage buffer for data or similar. This block also gets overwritten when dealing with floppy disc operations. Please have a look at the corresponding FutureOS functions.

The usage of OS functions:

The file '#EQU-API.ENG' contains a collection of entry addresses of the OS functions. They are sorted according to the FutureOS ROM (A, B, C or D) in which they are located.

In your application source code you can either put the EQU's of the used OS functions, which can be found in the above file. Or you can include the whole file '#EQU-API.ENG'. Using the Maxam assembler your code would contain:

READ"#EQU-API.DEU"

Regarding this file you need to take one thing into account. Delete the semicolon before every name of OS versions which you want to use. The semicolons are there to speed up assembly by reducing the number of labels.

The OS functions are divided into four different OS ROMs. Before using an OS function its corresponding ROM must be banked it. The API of FutureOS provides different means to perform this task. In addition you can bank in the ROM directly too.

Example: To call an OS function in ROM A you can use the API entry ROM_A. To use functions of the OS ROMs B, C or D you are going to use entries ROM_B, ROM_C or ROM_D. In all this cases the target address of the OS functions which you want to call is provided in the register HL. Here an example:

;Keyboard management, read a pressed key (see Technical Manual of ROM A)

```
X_XALLET EQU &C115      ;This line is taken from the file #OS-VAR.ENG
                          ;It defines the address of the OS function H_XALLET
LD HL,H_XALLET          ;HL points to the address of the OS function H_XALLET
CALL ROM_A              ;Bank in the FutureOS ROM A and
                          ;execute the OS function H_XALLET
```

;Now the accumulator (register A) of the Z80 CPU contains the value of a pressed key

Attention: Registers BC and HL can't be used to pass parameters or data.

To call an OS function in its corresponding OS ROM and subsequently bank in the previous ROM you can use another set of entries of the API. An example: You can use the API entry ROM_A2C to call an OS function in ROM C and after the execution of the OS function the ROM A will be banked in again. In this case the address of the OS function needs to be passed using the register IX. All other registers of the Z80 CPU can be used freely (load them with parameters etc.). Here an example:

;Wait until any key was pressed (see documentation of ROM C)

```
WATA EQU &C115           ;This line was taken from file '#OS-VAR.DEU'
                          ;It defines the address label of the OS function WATA
LD IX,WATA               ;IX points to the address of OS function WATA
CALL ROM_A2C             ;Bank in FutureOS ROM C, execute the OS function WATA
                          ;in ROM C and then bank in ROM A again
```

;Some key was pressed...

Direct Call of OS Functions:

Advanced Programmers can in addition bank in ROMs directly, just before you call an OS function. This approach can save code lines in some cases.

```
Banking in of OS ROM A: LD BC,(&FF01):OUT (C),C
Banking in of OS ROM B: LD BC,(&FF07):OUT (C),C
Banking in of OS ROM C: LD BC,(&FF0D):OUT (C),C
```

Banking in of OS ROM D: LD BC,(&FF13):OUT (C),C

In case you want to preserve the contents of register BC, please store and restore it using the commands PUSH BC and POP BC. For ROM A for example your source code would look like the next sentence...

Banking in OS ROM A: PUSH BC:LD BC,(&FF01):OUT (C),C:POP BC

For ROMs B, C and D use the similar equivalent commands.

How to End or Quit an Application?

As soon as an application has ended the control must be given back to the OS. This can be achieved by two different ways.

1. The application only altered the lower part of the screen. Or it didn't even output something to the screen. In this case your application can just use the OS function KCLICK to jump back to the Desktop

;Quit this application. All the icons of the upper screen part are preserved

```
KCLICK EQU &FE9A      ;This code line was taken from file '#OS-VAR.DEU'
                        ;Thus the address of OS function KCLICK is defined
LD HL,KCLICK           ;HL points to the address of the OS function KCLICK
CALL ROM_D             ;Banking in the FutureOS ROM D and return to the
                        ;Turbo Desk desktop using the OS function KCLICK
```

;From now on the control of the program is given back to the FutureOS Desktop

2. The application did write or paint something all over the screen. In this case the OS function TUR_D shall be used to give control back the OS. Example:

;Quit this application. The screen was used

```
TUR_D EQU &FEA0        ;This code line was taken from the file '#OS-VAR.DEU'
                        ;Thereby the address of the OS function TUR_D is defined
LD HL, TUR_D           ;HL points to the address of the OS function TUR_D
CALL ROM_D             ;Bank in the FutureOS ROM D and...
                        ;Return to FutureOS using OS function TUR_D
```

;From now on the control of the computer is given back to the FutureOS Turbo Desktop

There is a third way too. It's in between KCLICK and TUR_D. It's the OS function TUR_E. All OS functions of that group are declared in the documentation of ROM D.

Further programming examples:

If you're interested in some more examples the please either refer to the source codes of then provided applications of the OS. Or take a look at the FutureOS homepage:

<http://www.FutureOS.de>