

Wie man für FutureOS ein Programm erstellt.

In diesem Artikel wird erklärt, wie man Applikationen für FutureOS programmieren kann. Im Gegensatz zu bekannten Betriebssystemen (BS) ist das relativ einfach.

Grundsätzlich ist es so, dass man für das FutureOS (OS) in Z80 Assembler oder unter C programmieren kann. Assembler ist hierbei die Programmiersprache der Wahl.

Um ein Programm für das OS zu schreiben, kann man prinzipiell jeden Assembler benutzen. Dazu eignen sich sowohl Z80 Assembler für den CPC als auch den PC. Persönlich arbeite ich mit MAXAM von Arnor. MAXAM bietet folgende Vorteile: Er läuft sehr stabil, ist praktisch fehlerfrei und verarbeitet auch sehr große Quell-Code Dateien. Weiterhin besteht er aus Editor und Assembler. Der eingegebene Quell-Code kann also direkt assembliert und ausgeführt werden. Nachteil von Maxam ist, dass er relativ langsam ist. Es bietet sich an, einen CPC Emulator im Turbo Modus zu nutzen. Vor allem wenn der Quell-Code einige Dutzend kB groß ist.

Um eine Applikation mit Maxam zu assemblieren geht man folgendermaßen vor:

- Laden oder Eingeben des Quell-Codes (das Abspeichern bitte nicht vergessen).
 - Assemblieren, z.B. an Adresse &9000. Diese Adresse eignet sich zum direkten testen.
 - Unter Maxam das RSX-Kommando |FDESK aufrufen, und damit ins OS einspringen. Wird das OS mit |FDESK aufgerufen, so werden die ersten 48 KB des Hauptspeichers erhalten.
 - Die assemblierte Applikation im RAM starten: Zum Start der Applikation kann man auf das RUN Icon klicken (oder Hotkey X benutzen). Anschließend drückt man zuerst auf die Taste 3 für „RAM-Applikation“. Dann gibt man die Startadresse ein, in diesem Fall &9000. Zuletzt gibt man die RAM-Konfiguration ein, das ist der Wert &C0 für den Hauptspeicher.
- Alternativ zum RUN Icon kann man die Applikation auch vom Maschinen-Monitor aus starten.
- Die Applikation wird nun ausgeführt. Nach ihrem Ende befindet man sich wieder im Desktop des FutureOS (bzw. im Maschinen Monitor).
 - Abschließend wird das FutureOS wieder verlassen, dazu benutzt man das END Icon.
 - Nun kann man direkt mit Maxam weiter am Source Code arbeiten.

Hinweise: Benutzt man Maxam 1½, so dann lässt sich der Quell-Code schneller laden, da PROTEXT als Editor für den Source-Code benutzt wird.

Worauf sollte man nun achten, wenn man einen Applikation auf diese Weise erstellt?

Während des Erstellens und Erprobens einer Applikation bietet es sich an diese direkt ins RAM zu assemblieren und dort zu starten, so wie oben beschrieben.

Ist die Applikation fertig gestellt, so wird sie selbstverständlich von Datenträger oder aus einem Erweiterungs-ROM heraus gestartet. Wie man Applikationen in ein Erweiterungs-ROM packt ist allerdings Thema für einen anderen Artikel.

Startet man eine Applikation direkt im RAM so ist einiges zu beachten: Beim Aufruf des OS werden einige Adressen verändert. Deshalb kann man seine Applikation nicht DIREKT an JEDE beliebige Adresse im RAM assemblieren.

Folgende RAM-Bereiche werden durch den Systemstart des OS verändert:

- Adresse &0038: dort steht nun &C9 (Code für RET). Die maskierbaren Interrupts sind blockiert.
- Adressen &0066 & &0067: werden mit den Bytes &ED, &45 beschrieben (Code für RETN). Auch die UNmaskierten Interrupts sind blockiert.
- Der Bereich von &A000 bis &FFFF wird mit &00 initialisiert, denn ab &A000 befinden sich im FutureOS die File-Tagging-Bytes (FTBs), der Text-Bildschirm und die System-Variablen. Siehe Handbuch und Datei ‚#OS-VAR.DEU‘

Der restliche Speicher von &0000 bis &9FFF, sowie das freie Erweiterungs-RAM (E-RAM) steht prinzipiell zur freien Verfügung.

Wird eine Applikation allerdings vom Datenträger gestartet, so kann sie im Hauptspeicher den RAM Bereich von &0000 bis &B7FF belegen. Das sind 46 kB im Hauptspeicher.

Testen einer Applikation:

FutureOS nutzt den unteren 16 KB Block (&0000 bis &3FFF) als Sortier-Puffer für Inhaltsverzeichnisse. Doch entsprechende OS Funktionen sichern diesen 16 KB Block im E-RAM. Dies stellt also kein Problem dar.

Allerdings muss sich ein Zeichensatz zwischen &3800 und &3FFF befinden. Es spielt hierbei keine Rolle ob der Zeichensatz sich im unteren ROM oder im RAM befindet. Man kann also auch seinen eigenen Zeichensatz ab &3800 in jede Applikation einbauen.

Zum testen ist es jedoch nicht ratsam seine Applikation an eine Adresse unterhalb von &4000 zu assemblieren.

Weiterhin legen die CPC basierten Assembler auch ihren Quell-Code im unterem Bereich des RAMs ab. Ab &0170 z.B. liegt unter Maxam der Quell-Code. Auch deshalb sollte man zum testen darauf achten die Applikation nicht zu weit unten im RAM zu assemblieren.

Im Bereich von &4000 bis &7FFF findet das E-RAM Banking statt! Das E-RAM-Banking ist nur für einige Applikationen von Bedeutung. Um auf der sicheren Seite zu sein sollte man seine Applikation jedoch nicht unterhalb von &8000 direkt assemblieren, falls Erweiterungs-Speicher genutzt werden soll.

Normalerweise ist es sinnvoll die Applikation ab &8000 oder &9000 zu assemblieren. Der Bereich von &8000 bis &9FFF steht in jedem Fall zur freien Verfügung. Diese 8 KB bieten genügend Platz für eine ausgewachsene Applikation in Z80 Assembler.

Was wenn nachgeladen wird?

Beim Laden (oder Speichern) von Dateien ins E-RAM dient der Bereich von &8000 bis &8FFF manchmal als Sektor-Puffer. Dies ist jedoch nur der Fall wenn größere Datenmengen am Stück ins E-RAM geladen werden. In diesem Fall sollte man ab &9000 assemblieren. In diesem Fall stehen immerhin noch 4 KB zur Verfügung (&9000 bis &9FFF).

Der Bereich von &9000 bis &9FFF wird vom OS nicht angetastet. Hier ist es möglich seine Applikation zu assemblieren, das OS aufzurufen, dann z.B. zusätzliche Daten zu laden und dann die Applikation ab &9000 aufzurufen.

Der Bereich von &9000 bis &9FFF ist zwar "nur" 4 KB groß, aber um 4 KB Code zu erzeugen muss man schon mit großen Quell-Code-Dateien arbeiten!

Zusätzliche Variablen oder Datenbereiche der Applikation können an anderer Stelle im RAM abgelegt werden. In unserem Beispiel unterhalb &9000.

Die Variablen müssen natürlich vom Programm selbst initialisiert werden. Solange man die Interrupts deaktiviert läßt und auf RST Befehle verzichtet kann man seine Variablen oder Puffer von Adresse &0000 an aufwärts anlegen. Auch der Bereich von &A000 bis &B7FF kann verwendet werden.

Bei größeren Programmen (d. h. der assemblierte Code ist größer als 4 KB) ist der Quell-Code bereits riesig, und Maxam sollte von und auf Disk assemblieren. Falls der assemblierte Z80 Code auf Diskette geschrieben wurde kann man wieder die vollen 46 KB (&0000 bis &B7FF) nutzen. Dieser Code kann anschließend unter FutureOS geladen und gestartet werden. Um die Datei-Tagging Bytes nicht zu überschreiben sollte man den Speicher von &0000 bis &9FFF nutzen. Dabei können diese 40 kB inklusive der Interrupt & RST Vektoren genutzt werden. Eine Applikation darf also ihre eigenen RST Vektoren haben. Natürlich darf sie auch ihre eigene Interrupt Verwaltung besitzen. Dies macht Applikationen deutlich schneller.

Das E-RAM kann man nutzen, wenn es in den XRAM_?? Variablen freigegeben wurde (siehe Handbuch und Datei ‚#OS-VAR.DEU‘.

Es kann auch der Bereich von &A000 bis &AFFF verwendet werden. Beispielsweise für Puffer oder Daten. Dieser Bereich enthält übrigens die Datei-Tagging-Bytes, die anzeigen ob eine Datei aktuell markiert ist oder nicht. Vorausgesetzt es findet während dieser Zeit kein Disketten-Betrieb statt. Die Arbeit mit Dateien verändert diese 4 KB.

Weiterhin kann man dem Bereich von &B000 bis &B7FF temporär als Datenspeicher nutzen, dieser Bereich wird allerdings auch von einigen Floppy-Disc-Funktionen überschrieben. Siehe Dokumentation der FutureOS Funktionen.

Die Verwendung von OS Funktionen:

Die Datei ‚#EQU-API.DEU‘ enthält eine Sammlung der Einsprung-Adressen aller OS Funktionen. Diese sind nach ihren ROMs (A, B, C oder D) sortiert.

Im Quellcode bietet es sich an entweder die EQUs der verwendeten OS Funktionen einzufügen oder aber die gesamte Datei ‚#EQU-API.DEU‘ einzubinden. In Maxam schreibt man hierzu:

READ“#EQU-API.DEU“

Dabei ist folgendes zu beachten: Das Semikolon vor jeder genutzten OS Funktion muss entfernt werden. Diese Semikolons sind eingefügt, um das Assemblieren zu beschleunigen und die Anzahl der Labels gering zu halten.

Da die OS Funktionen auf vier verschiedene ROMs verteilt sind muß das entsprechende ROM vor dem Aufruf der Funktion eingeblendet sein. Dazu bietet das API des FutureOS mehrere Möglichkeiten, man kann die ROMs allerdings auch per Hand einblenden.

Um z.B. eine OS Funktion in ROM A aufzurufen wird der API Einsprung ROM_A benutzt. Für die ROMs B, C oder D würde man die Einsprünge ROM_B, ROM_C oder ROM_D benutzen. Hierbei wird die Adresse der OS Funktion in Register HL übergeben. Hier ein Beispiel:

;Abfrage der Tastatur, lese gedrückte Taste (siehe Dokumentation von ROM A)

```
X_XALLET EQU &C115      ;Diese Zeile ist aus Datei #OS-VAR.DEU übernommen
                        ;Dadurch wird die Adresse der OS Funktion H_XALLET definiert
LD HL,H_XALLET         ;HL zeigt auf die Adresse der OS Funktion H_XALLET
CALL ROM_A             ;Einblenden von FutureOS ROM A und
                        ;Ausführen der OS Funktion H_XALLET
```

;Der Akku enthält nur den Wert einer gedrückten Taste

Beachte: Die Register BC und HL können nicht zur Übergabe von Parametern genutzt werden.

Um eine OS Funktion in einem bestimmten OS ROM aufzurufen und anschließend wieder ein bestimmtes OS ROM einzublenden nutzt man weitere Einsprünge des API. Beispielsweise kann man mittels API Einsprung ROM_A2C eine OS Funktion in ROM C aufrufen und nach deren Beendigung wird wieder ROM A eingeblendet. Die Adresse der OS Funktion wird hierbei im Register IX übergeben. Alle anderen Register der Z80 können mit Parametern geladen werden. Hier ein Beispiel:

;Warten bis eine beliebige Taste gedrückt wurde (siehe Dokumentation von ROM C)

```
WATA EQU &C115          ;Diese Zeile ist aus Datei #OS-VAR.DEU übernommen
                        ;Dadurch wird die Adresse der OS Funktion WATA definiert
LD IX,H_XALLET         ;IX zeigt auf die Adresse der OS Funktion WATA
CALL ROM_A             ;Einblenden von FutureOS ROM C, Ausführen der
                        ;OS Funktion WATA in ROM und rückblenden vom ROM A
```

;Eine Taste würde gedrückt

Direkter Aufruf von OS Funktionen:

Für fortgeschrittene Programmierer empfiehlt es sich die OS ROMs direkt einzublenden, bevor man eine OS Funktion aufruft. Dies spart in einigen Fällen Programm-Zeilen.

```
Einblenden vom OS ROM A: LD BC,(&FF01):OUT (C),C
Einblenden vom OS ROM B: LD BC,(&FF07):OUT (C),C
Einblenden vom OS ROM C: LD BC,(&FF0D):OUT (C),C
Einblenden vom OS ROM D: LD BC,(&FF13):OUT (C),C
```

Soll das Register BC erhalten bleiben, so ist es mittels PUSH und POP zu sichern.

Für ROM A würde man das so schreiben:

```
Einblenden vom OS ROM A: PUSH BC:LD BC,(&FF01):OUT (C),C:POP BC
```

Für die ROMs B, C und D analog dazu.

Das Verlassen einer Applikation:

Ist eine Applikation beendet, so muß sie die Kontrolle wieder an das OS zurück geben. Dies kann auf zwei Arten geschehen.

1. Die Applikation hat nur die untere Hälfte des Bildschirms verändert, oder gar nichts auf den Monitor geschrieben. In diesem Fall genügt es mittels OS Funktion KLICK ins Desktop zurück zu springen:

```
;Applikation verlassen. Die Icons der oberen Bildschirm-Hälfte sind erhalten
KLICK EQU &FE9A          ;Diese Zeile ist aus Datei #OS-VAR.DEU übernommen
                        ;Dadurch wird die Adresse der OS Funktion KLICK definiert
LD HL, KLICK            ;HL zeigt auf die Adresse der OS Funktion KLICK
CALL ROM_D              ;Einblenden von FutureOS ROM D und
                        ;Rücksprung ins Desktop mittels OS Funktion KLICK
;Ab hier ist die Kontrolle wieder an das FutureOS Desktop zurück gegeben
```

2. Die Applikation hat den Bildschirm beschrieben. In diesem Fall wird die Kontrolle mittels OS Funktion TUR_D an das OS zurück gegeben:

```
;Applikation verlassen. Auf den Bildschirm wurde zugegriffen
TUR_D EQU &FEA0          ;Diese Zeile ist aus Datei #OS-VAR.DEU übernommen
                        ;Dadurch wird die Adresse der OS Funktion TUR_D definiert
LD HL, TUR_D            ;HL zeigt auf die Adresse der OS Funktion TUR_D
CALL ROM_D              ;Einblenden von FutureOS ROM D und
                        ;Rücksprung ins FutureOS mittels OS Funktion TUR_D
;Ab hier ist die Kontrolle wieder an FutureOS zurück gegeben
```

Es gibt auch noch ein Zwischending zwischen KLICK und TUR_D. Es ist die OS Funktion TUR_E. Alle diese OS Funktionen sind in der Beschreibung zu ROM D erläutert.

Weitere Programmier-Beispiele:

Wenn sich jemand für Programmierbeispiele interessiert, dann sei an dieser Stelle auf die Quell-Codes der zum OS mitgelieferten Programme verwiesen. Oder auf die FutureOS Heimseite: <http://www.FutureOS.de>