3 Struktur und Datentypen

Hintergründe als Basis der Programmierung

In vorausgegangenen Abschnitt wurde gezeigt, wie man Programme für die Symbian OS Plattform produziert. Um selber Programme zu schreiben, ist allerdings etwas Hintergrundwissen über die Komponenten des Systems und ihre Funktion erforderlich. Ebenso erforderlich ist das Wissen um die Konventionen und Datentypen bei der Programmierung. Dieses Kapitel gibt einen Überblick über den Aufbau des Betriebssystems und die Text Deskriptoren.

3.1 Aufbau von Symbian OS

Symbian OS ist darauf ausgelegt, auf kleinen, tragbaren Computern zu laufen, die vorwiegend als Kommunikationsgeräte benutzt werden. Diese Auslegung hat Konsequenzen auf das Design des Software Systems. Die Betrachtung des Aufbaus von Symbian OS beginnt daher mit einem Überblick über die Hardware. Die anschliessende Betrachtung der Software umfasst die wichtigsten Stichworte im Zusammenhang mit der Multi-Prozess Architektur des Betriebssystems. Es folgen einige Abschnitte über den Mikrokernel, eine Übersicht über die Systemkomponenten, die Funktion der Gerätetreiber, die Speicherverwaltung und die Behandlung von Ausnahmen.

3.1.1 Hardware

Wie bei jedem Computer besteht die Hardware von Symbiangeräten aus dem Prozessor (CPU), Speicher (ROM und RAM), Ein- und Ausgabegeräten, sowie der Stromversorgung. Der Stromversorgung besitzt bei einem batteriegetrieben Gerät eine besondere Bedeutung. Das RAM arbeitet auch als persistentes Speichermedium, eine Festplatte ist nicht vorgesehen. Hier eine Übersicht über die Komponenten der Hardware:

- *CPU:* Symbian OS bzw. EPOC ist für eine 32 Bit CPU entwickelt, die mit deutlich niedrigerer Taktrate als die CPU eines Desktop-PCs läuft. Als Prozessoren kommen z.B. 36, 52, 104 oder 156 MHz ARM (bzw. StrongARM) CPUs in Frage. Das Nokia 7650 hat z.B. eine ARM9 CPU mit 52 MHz Taktrate. Das P800 besitzt eine ARM9 CPU mit 156 MHz.
- *ROM:* Das ROM enthält das Betriebssystem, sowie die Grundausstattung an Middleware und Anwendungen. Bei einem PC liegen im ROM nur ein kleiner Bootstrap Loader und das BIOS. Beim PC werden das Betriebssystem und alle Anwendungen von der Festplatte geladen. Bei Symbian OS ist das System ROM auf das Z: Laufwerk abgebildet. Alle Daten im ROM sind als Datei im Dateisystem auf Z: erreichbar und ausserdem direkt im ROM adressierbar. Programme werden also direkt im ROM ausgeführt, anstatt sie erst (wie bei einem PC) ins RAM zu laden. Manche Symbiangeräte haben 12 oder 16 MB ROM, andere weniger.
- *RAM:* Das RAM erfüllt zwei Funktionen. Es dient einerseits als Arbeitsspeicher (wie das RAM beim PC) für gerade laufende Anwendungen und den Kernel. Andererseits arbeitet es wie eine Festplatte als C: Laufwerk. Das System verwendet den Speicher gerade so, wie es ihn benötigt. Man muss also keinen Speicherbereich für den einen oder den anderen Zweck vorab reservieren. Da das RAM aber meistens sehr beschränkt ist (je nach System 3 bis 16 MB), ist es immer möglich, dass es zu einem Engpass im Arbeitsspeicher kommt ("out-of-memory" Fehler), bzw. dass die Platte volläuft („disk full“ Fehler).
- *Eingabe/Ausgabe Geräte* (I/O Geräte) sind in der Regel die Tastatur, das Display oder der Touchscreen (falls vorhanden). Zusätzlich gibt es Geräte mit einem Erweiterungskarten Einschub (z.B. CompactFlash) für zusätzliche „Festplatten“, die dann als D: Laufwerk angesprochen werden. Auch die serielle Schnittstelle, die Einwählverbindung oder der Infrarot Port bzw. Bluetooth Port sind Eingabe-und Ausgabegeräte.
- Die *Stromversorgung* umfasst den Hauptakku, Zweitakku und die externe Stromversorgung.

Auf die Dauer hilft nur Power

Diese Ausstattung der Hardware hat einige Auswirkungen auf die Anwendungsentwicklung. Die Ressourcen sind sehr beschränkt. Es gibt nur eine vergleichsweise langsame CPU und wenig Speicher. Es gibt keine Festplatte im Sinne eines grossen virtuellen Speichers, wie bei einem PC. Man kann also nicht von einem unbegrenzten Speicherplatz ausgehen, um Anwendungsdaten auszulagern. Gutes Haushalten

mit der Energieversorgung (Power Management) ist äußerst wichtig, da Benutzerdaten meist im RAM vorgehalten werden. Setzt die Stromversorgung aus, so könnten diese Daten verloren gehen. Auch wenn das Gerät ausgeschaltet ist oder die Batterien gewechselt werden, dürfen die Benutzerdaten nicht verloren gehen. Zur Überbrückung dient der Zweitakku, bzw. man verwendet, wie bei neuen Geräten und bei vielen Speicherkarten üblich, Flash RAMs. Ausserdem besteht die Möglichkeit der Datensicherung (Back Up) auf der Festplatte eines PCs.

Zusammenfassend kann man sagen, das Software für ein Symbiangerät sehr kompakt (d.h. speichersparend) entwickelt werden sollte. Auch muss die Software in der Lage sein, durch Engpässe bedingte Fehler (wie out-of-memory) abzufangen, die bei einem aus Prinzip knapp bemessenen System immer vorkommen können. Die Software sollte immer stabil laufen. Symbian unterstützt den Entwickler hierbei durch seine Systemarchitektur. Allerdings funktioniert das nur, wenn der Anwendungsentwickler sich an die vorgegebenen Regeln und Konventionen hält.

3.1.2 Software

Privilegien und Prozesse

Was gibt es bei der Softwareentwicklung zu beachten, um Anwendungen zu realisieren, die sparsam mit den gegebenen Systemresourcen umgehen? Ein Blick auf die Softwarekomponenten, auf denen das Symbian OS basiert, hilft weiter. Hierzu gehören die Abgrenzung durch Privilegien und Prozesse, sowie die Abgrenzung zwischen Anwendungen und DLLs (Dynamic Link Libraries). Zu den beschriebenen Mechanismen gehören die Multi-Threading Architektur, die Kontextübergabe beim Umschalten zwischen Prozessen, sowie die Ausführung von Programmen in Form von ".exe" Modulen und DLLs.

Wie bei jedem modernen Betriebssystem gibt es Grenzen zwischen den einzelnen Softwarekomponenten, so auch in Symbian OS. Diese Grenzen dienen dem Schutz vor unerlaubten Zugriff einer Komponente auf Speicherbereiche einer anderen Komponente oder auf Hardware. Dadurch wird das Betriebssystem stabiler und sicherer. Abbildung 3-1 zeigt eine Übersicht über die Systemkomponenten und ihre Abgrenzung.

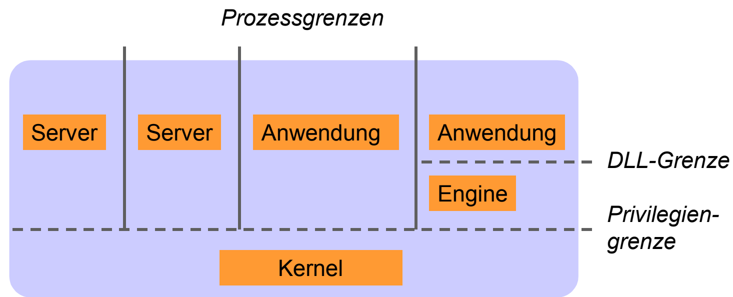


Abb. 3–1
Grenzen zwischen den Komponenten

Die wesentlichen Komponenten sind der Kernel des Betriebssystems, Server und Anwendungen. Hierzu einige Erläuterungen:

- *Kernel:* Der Kernel verwaltet die Hardwareressourcen wie RAM, I/O Geräte usw. Er ermöglicht und kontrolliert den Zugriff anderer Softwarekomponenten auf diese Hardwareressourcen. Der Kernel selbst verwendet durch die Hardware unterstützte Privilegien, um auf diese Hardwareressourcen zuzugreifen. Dies bedeutet konkret, dass die CPU bestimmte privilegierte Instruktionen nur für den Kernel ausführt (im Kernel-Mode). Andere Programme, so genannte User-Mode Programme, werden ohne Privilegienrechte ausgeführt und können somit auf die Hardwareressourcen nur über den Kernel (mit Hilfe einer Kernel API) zugreifen. Die Grenze zwischen dem Kernel und allen anderen Komponenten ist die *Privilegiengrenze*.
- *Anwendung:* Eine Anwendung ist ein Programm mit einer Benutzerschnittstelle. Jede Anwendung läuft in einem eigenen Prozess, der einen eigenem virtuellen Adressraum besitzt. D.h. eine Anwendung kann nicht aus Versehen oder absichtlich Daten einer anderen Anwendung überschreiben, da die Adressräume der Anwendungen voneinander getrennt sind. Dies macht Anwendungen sicherer. Die Grenze zwischen zwei Anwendungen ist daher die *Prozessgrenze*.
- *Server:* Ein Server ist ein Programm ohne Benutzerschnittstelle. Ein Server verwaltet meist eine oder mehrere Ressourcen und besitzt eine API, die Clients den Zugriff auf die Dienste des Server ermöglicht. Ein Client kann z.B. eine Anwendungen oder ein anderer Server sein. Jeder Server läuft normalerweise in seinem eigenen Prozess. D.h. die Grenze zwischen Server und Client ist eine Prozessgrenze. Symbian OS verwendet Server sehr häufig, um Dienste zur Verfügung zu stellen, die bei anderen Betriebssystemen vom Kernel oder Gerätetreibern angeboten werden (beispielsweise den Fileserver).

- *Engine:* Eine Engine ist der Teil einer Anwendung, der für die Programmlogik und die Daten zuständig ist und weniger für die Interaktion mit dem Benutzer der Anwendung. Man kann häufig eine Anwendung in einen Bereich für die Benutzerschnittstelle (GUI) und einen Bereich für die Engine einteilen. Wo genau diese Trennung vollzogen wird, obliegt der Kunst des Software-Ingenieurs. Eine solche Aufteilung lohnt sich bei komplexen Anwendungen. Eine Anwendungs-Engine kann ein getrenntes Quellcode-Modul sein, eine eigene DLL, oder sie kann sogar mehrere DLLs umfassen. Die Grenze zwischen einer Anwendung und dessen Engine nennt man die Modulgrenze bzw. *DLL-Grenze*. Sie dient mehr dem guten Softwaredesign als dem Schutz vor unerwünschten Zugriffen anderer Anwendungen.

DLL-Grenzen (bzw. Modulgrenzen) sind im Sinne von Einbussen bei der Systemleistung sehr kostengünstig zu überschreiten. Sie unterstützen die Systemintegrität durch Modularisierung und Kapselung. Die Privilegiengrenze ist etwas teurer zu überschreiten, aber unterstützt die Systemintegrität durch Trennung des Kernel und der Hardware von User-Mode Zugriffen.

Am teuersten ist das Überschreiten der Prozessgrenzen. Diese unterstützen die Systemintegrität und Systemstabilität durch die Trennung des Speicherbereichs der einzelnen Anwendungen voneinander.

Ein Prozess hat einen eigenen Adressraum

Ein Prozess leistet eine Art Schutzfunktion für Programme in Symbian. Jeder Prozess im Symbian Betriebssystem läuft in seinem eigenen Speicherbereich, der von den anderen Prozessen getrennt ist. Die virtuellen Adressen, die beim Ausführen eines Programms dieses Prozesses verwendet werden, werden in physikalische Adressen im ROM und RAM übersetzt. Diese Übersetzung der Adressen wird von der sogenannten Memory Management Unit (MMU) durchgeführt. Beschreibbare Speicherbereiche, in denen ein Prozess läuft, sind dabei nicht von einem anderen Prozess aus erreichbar. Prozesse laufen in voneinander völlig getrennten Adressräumen.

Threads als Kontrollflüsse

Ein Thread ist der rote Faden bei der Ausführung eines Programmes. Ein Prozess kann einen oder mehrere Threads besitzen. Jeder Thread eines Prozesses wird unabhängig von den anderen ausgeführt, aber im selben Adressraum (nämlich dem Adressraum des zugehörigen Prozesses). Ein Thread kann daher Daten eines anderen Threads im selben Prozess ändern, ob absichtlich oder aus Versehen. Daher sind Threads nicht so gut voneinander isoliert wie Prozesse. Abbildung 3-2 zeigt das Prinzip.

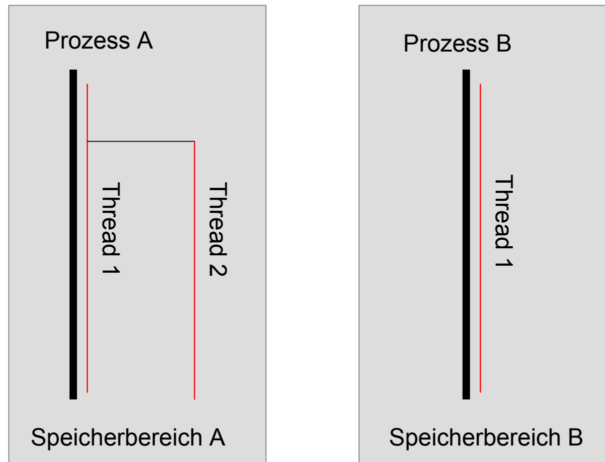


Abb. 3–2
Prozesse und Threads

Multitasking

Threads werden auf präemptive Weise vom Kernel verwaltet. Dazu eine kurze Erläuterung. Der Thread mit der höchsten Priorität, der zu einer bestimmten Zeit an der Reihe ist, wird vom Kernel ausgeführt. Ein Thread, der nicht für die Ausführung geeignet ist, weil er zu dem bestimmten Zeitpunkt gerade z.B. auf Daten eines I/O Gerätes oder auf andere Ereignisse (Events) wartet, wird vom Kernel in einer Warteliste gehalten. Tritt dann das erwartete Ereignis auf, so kann der Thread geweckt werden und wird weiter ausgeführt. In der Zwischenzeit (während der Thread wartet) haben die anderen Threads die Möglichkeit zu arbeiten. Immer, wenn ein Thread geweckt oder schlafen gelegt wird, überprüft der Kernel welches der nächste Thread mit der höchsten Priorität ist und führt ihn aus. Dies kann dazu führen, dass ein Thread, der gerade läuft, mitten in der Ausführung unterbrochen wird, da ein anderer Thread höhere Priorität hat. Diese Unterbrechung eines Threads durch einen Thread mit höherer Priorität nennt man Präemption. Daher auch der Begriff "präemptives Multitasking".

Kontextübergabe

Den Vorgang der Übergabe der Ausführungskontrolle von einem Thread zum nächsten nennt man Kontextübergabe. Hier wurde sehr darauf geachtet, dass der Aufwand bei der Kontextübergabe zwischen Threads sehr gering bleibt um möglichst wenig an Performanz zu verlieren. Dennoch ist die Kontextübergabe mit Kosten bzgl. der Performanz verbunden und in dieser Hinsicht weitaus teurer als z.B. ein Funktionsaufruf. Am teuersten ist allerdings die Kontextübergabe von einem Thread eines Prozesses an einen Thread eines anderen Prozesses. Das hängt damit zusammen, dass die Kontextübergabe in diesem Fall Änderungen an den Einstellungen der Memory Management Unit nach sich zieht. Die MMU muss ihre Tabelle (Mapping Table) für die

Abbildung der virtuellen Adressen eines neuen Prozesses auf die echten physikalischen Adressen im Speicher entsprechend anpassen (laden bzw. auslagern), was natürlich etwas Zeit kostet. Eine Kontextübergabe zwischen zwei Threads im selben Prozess ist also weitaus günstiger. Dies ist auch einer der Gründe, warum bei Betriebssystemen Threads ursprünglich eingeführt wurden. Graphische Oberflächen wären ohne Threads fast nicht denkbar. Abbildung 3-3 zeigt die Abbildung der virtuellen Adressräume zweier Prozesse in den physikalischen Speicher durch die MMU.

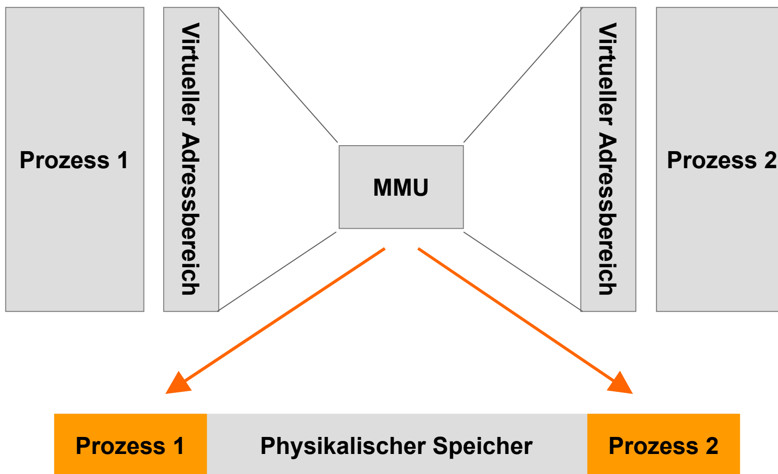


Abb. 3–3 *Kontextübergabe mit Hilfe der Memory Management Unit*

Anwendungen und Prozesse

Normalerweise verwendet bei Symbian jede Anwendung ihren eigenen Prozess und nur einen Thread. Jeder Server verwendet auch seinen Prozess und nur einen Thread. Dies muss aber nicht so sein. In manchen Fallen, in denen z.B. mehrere Server eng miteinander verkoppelt arbeiten, werden sie in einen einzigen Prozess mit mehreren Threads gepackt, um die Kontextübergabe möglichst günstig zu gestalten. Daher laufen z.B. die Server, die mit der Kommunikation zusammenhängen, wie Serial-, Socket- oder Telefonieserver im selben Prozess. Einige Details hierzu finden sich in Kapitel 6.

Bei den vorausgegangenen Erläuterungen wurden einige Begriffe eingeführt, die sicherlich noch etwas erklärungsbedürftig sind. Was ist eine DLL? Was ist ein Quellcode-Modul? Wo werden Programme ausgeführt? Wie funktioniert das Laden? Wie starten Anwendungen und Server? Welche Konventionen sind zu beachten?

Programme verpacken

Was die CPU betrifft, ist ein Programm nur eine Serie von Instruktionen. Um Software aber effektiver verwalten zu können, wird Quellcode in Pakete gruppiert. Diese Pakete sind bei Symbian sind sehr ähn-

lich mit denen von Betriebssystemen wie Windows NT. Zu den Paketen gehören:

- Ein Modul mit der Bezeichnung ".exe" am Ende ist ein Programm mit einem einzelnen Einstiegspunkt: `E32Main()`. Beim Starten eines ".exe" Moduls wird vom System zunächst ein neuer Prozess mit einem Thread erzeugt. Der Einstiegspunkt wird dann in diesem Thread aufgerufen.
- Eine DLL oder Dynamic Link Library ist eine Bibliothek von Programmcode mit eventuell mehreren Einstiegspunkten. Das System lädt eine DLL in den Kontext eines existierenden Threads (und damit in den Kontext eines existierenden Prozesses). Es gibt zwei wichtige Typen von DLLs:

  1. Eine *Shared Library DLL* definiert eine feste API, die von einem oder mehreren Programmen verwendet werden kann. Die meisten Shared Library DLLs haben die Endung ".dll". Ausführbare Programme werden mit den DLLs gekennzeichnet, die sie benötigen und diese werden dann automatisch während der Laufzeit geladen. Das Laden geschieht rekursiv, also auch die Shared Library DLLs, die von anderen DLLs benötigt werden, werden solange geladen, bis alle benötigten Teile zur Verfügung stehen.
  2. Eine *polymorphische DLL* implementiert eine abstrakte API wie z.B. einen Druckertreiber, ein Sockets-Protokoll oder eine EIKON Anwendung. Diese DLLs haben in der Regel eine andere Endung als ".dll", nämlich z.B. ".prn", ".prt" oder ".app". Polymorphische DLLs haben normalerweise nur einen einzelnen Einstiegspunkt. In der Regel werden polymorphische DLLs explizit von dem Programm geladen, das sie benötigt.

Ausführbare Programme müssen vor der Ausführung erst geladen werden. Das bedeutet, dass das Programm und die Datenbereiche für die Ausführung vorbereitet werden müssen. Programme, die sich im ROM befinden, werden dort auch direkt ausgeführt. Hier ist das Laden trivial. Für Programme, die sich auf einer Erweiterungskarte (beispielsweise ein Compact Flash Karte als D: Laufwerk) oder auf der RAM Disk (C: Laufwerk) befinden, bedeutet das Laden mehr Aufwand.

Speicherbedarf von Programmen

Symbian OS optimiert die Größen von DLLs, um sie möglichst kompakt im ROM oder RAM unterbringen zu können. Die Größenoptimierung basiert auf zwei Prinzipien:

- Die meisten Systeme erlauben es, die Einstiegspunkte bei DLLs oder ähnlichen Konzepten entweder per Name oder über eine

Ordinalzahl zu referenzieren. Namen sind aber unter Umständen sehr lang und verbrauchen relativ viel Speicherplatz. Symbian OS unterstützt daher nur die Referenzierung der Einstiegspunkte über Ordinalzahlen.

- Das Laden von Programmen ins RAM kann dazu führen, dass die Adresse im Speicher, an die der Programmcode geladen werden soll, erst zur Ladezeit bestimmt werden kann. Dies bedeutet, dass die ausführbaren Programme Relokations-Informationen enthalten müssen. Beim Laden in das ROM steht die Position eines Programms allerdings schon vorher fest. Um Speicherplatz zu sparen, werden die Relokations-Information in DLLs, die im ROM liegen, beim Symbian OS weggelassen.

Programme ausführen

Wie funktioniert das Laden und Teilen von Programmen? Ausführbare Programme enthalten drei Typen von Binärdaten: den Programmcode, statische Daten nur zum lesen (Read-only) und beschreibbare statische Daten (Read/Write). Symbian macht hier Unterschiede zwischen ".exe" und DLL Programmen.

In ".exe" Modulen können die Daten und der Ausführungskode nicht zwischen verschiedenen Programmen aufgeteilt werden. Ein ".exe" Modul, das ins RAM geladen wird, hat seinen eigenen Bereiche für den Kode, die Read-only Daten und die Read/Write Daten. Wird ein zweites Exemplar des gleichen ".exe" Moduls gestartet, so werden entsprechend neue Bereiche für diese angelegt. Hier gibt es eine kleine Optimierung: ROM basierte ".exe" Module allokieren nur Speicherbereiche für ihre Read/Write Daten im RAM. Der Programmkode und die Read-only Bereiche werden direkt aus dem ROM gelesen.

Bei DLLs ist das anders. Hier können sich verschieden Programme eine DLL teilen. Beim ersten Ladevorgang wird eine DLL an eine bestimmte Adresse geladen. Will ein zweiter Thread dieselbe DLL auch verwenden, muss sie nicht erneut an eine andere Stelle geladen werden. Es wird einfach die schon vorhandene Kopie im Speicher verwendet. Die DLL erscheint also für alle Threads, die sie verwenden, an derselben Adresse im Speicher. Symbian OS besitzt für jede geladene DLL Referenzzähler, mit deren Hilfe festgestellt werden kann, wann eine DLL nicht mehr benötigt wird und der Speicher daher wieder freigegeben werden kann (wenn eine DLL nämlich nicht mehr referenziert ist). ROM basierte DLLs werden überhaupt nicht geladen, sie werden einfach dort im ROM verwendet, wo sie liegen.

Wie werden Anwendungen nun gestartet? Die meisten Server haben ein eigenes ".exe" Modul, um einen eigenen Prozess zu starten. Der Windows Server für die grafische Bedienoberfläche beispielsweise findet sich in `ewsrv.exe`, der Server für das Dateisystem (Fileserver)

in `efsrv.exe`. Manche Server werden auch innerhalb des Prozesses eines anderen Servers als Thread gestartet, um die Kosten bei einer Kontextübergabe möglichst gering zu halten. So startet das Modul `c32exe.exe` beispielsweise den Server für die serielle Kommunikationsschnittstelle. Die anderen Kommunikationsserver verwenden eine DLL und starten ihren eigenen Thread innerhalb dieses Servers.

Einstiegspunkte

Eine Konsolenanwendung wie z.B. `hellotext.exe` wird in einem eigenen Prozess gestartet und öffnet eine eigene Konsole für die Interaktion mit dem Benutzer. Die meisten Anwendungen mit einer graphischen Benutzeroberfläche (GUI) sind allerdings polymorpische DLLs sind, deren Haupteinstiegspunkt `NewApplication()` ein Objekt der Klasse `CEikApplication` zurückliefert, wie das Übungsbeispiel mit der HelloWorld GUI Anwendung in Kapitel 2 gezeigt hat. Der Prozess, in dem eine solche Anwendung läuft, wird von einem kleinen ".exe" Modul gestartet. Dieses Modul ist die `apprun.exe`, an die dann der Name der ".app" als Parameter übergeben wird. Der Vorteil dieser Methode besteht darin, dass, wenn in der Anwendung ein eingebettes Dokument (oder eine andere Komponente) verwendet werden soll, dieses in denselben Prozess geladen werden kann, indem einfach die entsprechende ".app" DLL im selben Thread geladen wird. Das Prinzip ist vergleichbar mit Java Applets, die alle in der selben Java Virtual Machine (JVM) laufen, während Java Anwendungen ihren eigenen `java.exe` Prozess haben.

3.1.3 Mikrokernel

Das Symbian OS besitzt eine sogenannte Mikrokernel-Architektur. Im Gegensatz dazu besitzen Betriebsysteme wie z.B. Linux einen monolythischen Kernel. Abbildung 3-4 zeigt beide Ansätze im Vergleich. Die Mikrokernel-Architektur erlaubt die Realisierung eines speichersparenden Kernels mit einem sogenannten "kleinen Footprint". Der Kernel enthält nur die notwendigsten Systemfunktionen. Hierzu gehören das Memorymanagement (d.h. die Speicherallokierung für Prozesse im Kernel-Mode und User-Mode), die Gerätetreiber und das Power-Management.

Server

Ausserhalb des Kernels (im User-Mode) befindet sich die Middleware mit den sogenannten Server-Komponenten. Die Server stellen die erweiterten Funktionen des Betriebsystems zur Verfügung. Hierzu gehören die Kommunikation, die grafische Benutzerführung (GUI) und der Fileserver. Durch diesen Aufbau ist das Betriebssystem robust, bootet schnell und hat sehr kurze Reaktionszeiten (z.B. auf den Tastendruck durch den Benutzer). Zusätzlich bietet ein Mikrokernel-

Betriebssystem durch seinen modularen Aufbau eine gute Erweiterbarkeit und Sicherheit, da nur ein sehr kleiner Teil des Betriebssystems (der Mikrokernel) im privilegierten Modus läuft. Die Wahrscheinlichkeit für Fehler und Systemabstürze bleibt relativ gering.

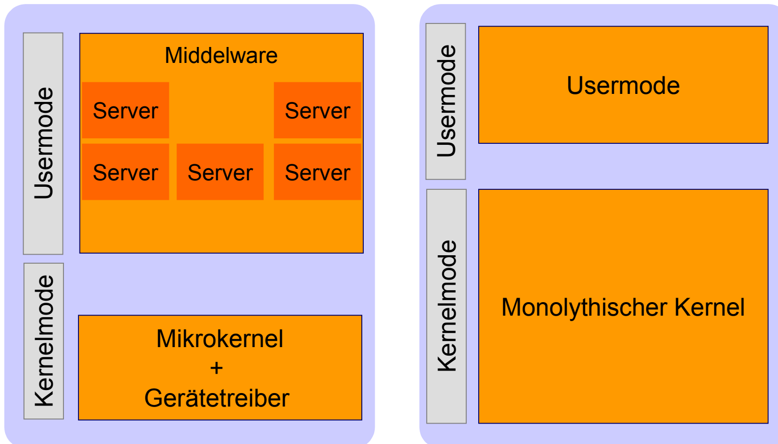


Abb. 3–4
Mikrokernel im Vergleich zum monolythischen Kernel

3.1.4 Systemkomponenten

Symbian OS kann man nach Anwendungsgruppen in verschiedene Bereiche aufteilen:

- *Basis:* sie besteht aus dem Kernel, der Userlibrary und den Basisdiensten wie Fileserver und Sicherheit.
- Darüber befindet sich die *Middleware*, die neben zusätzlichen Sicherheit- und Telephoniekomponenten, APIs für Daten-Management, Text, Zwischenablage, Grafik, Internationalisierung und die Core GUI Komponenten enthält. Multimediakomponenten erlauben die Bearbeitung von Grafiken und Tönen. Die Kommunikationskomponenten ermöglichen den Zugriff auf verschiedenste Protokolle und den Datenaustausch mit der Außenwelt.
- Über der Middleware befinden sich als *Anwendungskomponenten* sogenannte Application Engines und Application Services, wie die Kontakt- und Terminverwaltung oder die Anwendungsinstallation. Daneben bietet Symbian OS schon eine eingebaute Unterstützung für Java Anwendungen.

Application Engines

Abbildung 3-5 zeigt eine Übersicht. Unter den gezeigten Application Engines sind vorgefertigte Komponenten für solche Anwendungen enthalten, die für mobile Endgeräte typisch sind. Dazu gehört der Zugang zu den Teilnehmereinträgen (Contacts) zusammen mit den Kommuni-

kationsmöglichkeiten des Endgerätes (z.B. vCards per SMS austauschen), sowie der Zugang zum Terminkalender (Agenda) mit der Möglichkeit zum Austausch von Kalenderobjekten (vCalender). Unter dem Stichwort "Office" sind Funktionen zusammengefasst, die den Umgang mit Büroanwendungen vorleisten, d.h. Spreadsheets, Textdokumente, Grafiken und kontextspezifischer Hilfe. SynchML ist ein standardisiertes XML-Format für die Synchronisation des Terminkalenders mit einem Desktop-System. Die Ablage von Daten wird durch ein System zum Datenmanagement unterstützt. Ausserdem gibt es für Symbian eine Implementierung des Opera Browsers zur Darstellung von HTML und XML Formaten aus dem Web, bzw. zur Darstellung lokal abgelegter Dateien. Der Zugang zum Web wird durch eine der unter der Gruppe Kommunikation enthaltenen Verbindungen hergestellt. Der Browser übernimmt auch das Laden und die Installation von Midlets. URL-Schemata wie "mailto:", "fax:" oder "sms:" übergibt der Browser an eine passende Anwendung aus der Gruppe Messaging.

Messaging

Zu den unter Messaging zusammengefassten Funktionen gehört die Unterstützung für SMS, MMS, E-Mail und FAX. Die Behandlung der unterschiedlichen Messaging-Formate ist grundsätzlich gleich. Das Symbian OS unterstützt Java in zwei Konfektionsgrössen: das MIDP für kompaktere Mobiltelefone (die oft auch als Smartphones bezeichnet werden), sowie ein umfangreicheres Paket für PDAs mit direktem Anschluss an das Mobilfunknetz. Die grössere Variante basiert auf Personal Java und enthält das Java Phone Paket, das unter anderem den Zugriff auf das Teilnehmerverzeichnis ermöglicht, Telefonanwendungen mit Hilfe der JTAPI unterstützt, sowie Zugriff auf einige mobilfunkspezifische Parameter bietet (wie z.B. Mobilfunkanbieter, Netz, Funkverbindung, Terminal und diesbezügliche Statusmeldungen).

Application Framework

Das Application Framework ist eine Umgebung für Anwendungen, die vom Symbian-Konsortium, Lizenznehmern und von Anwendungsentwicklern aus dem freien Markt bereitgestellt werden. Das Application Framework enthält einen Baukasten für die grafische Benutzerführung (GUI). Hierzu gehören vorgefertigte Komponenten für eine ereignisgesteuerte Benutzerführung mit grafischen Elementen (Widgets), und der Abbildung von Meldungen des Systems. Im weltweiten Einsatz werden die unterschiedlichsten Zeichensätze verwendet. Symbian unterstützt den Unicode Zeichensatz, die Eingabe per Handschrift, Pictogramme, sowie Sortierfunktionen für den Vergleich von Textstrings.

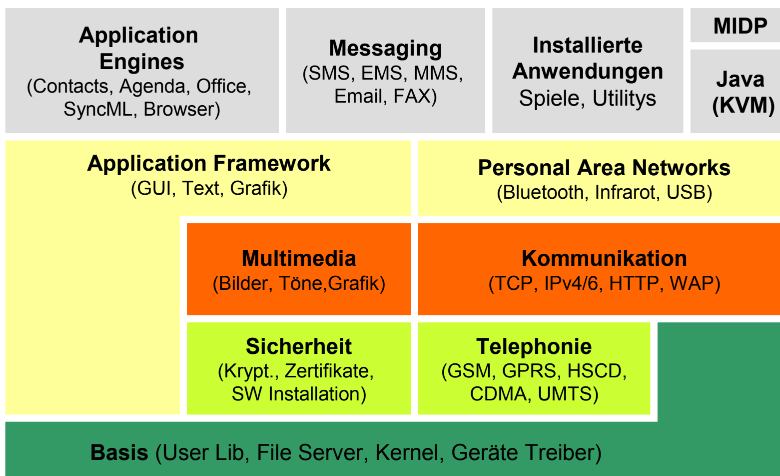


Abb. 3–5
Kernel und Systemkomponenten

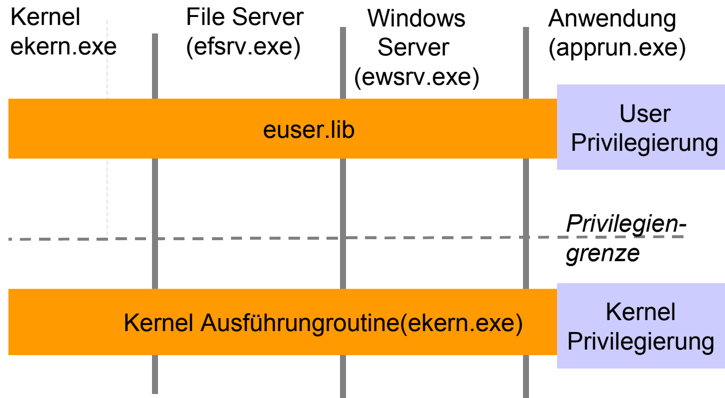
Personal Area Networks

Zum Personal Area Network gehören der Bluetooth, sowie die Möglichkeit zur Vernetzung via Infrarot (IrDA) und USB. Die Kommunikationskomponenten zusammen mit Bluetooth, Messaging und der Telefonie werden in Kapitel 6 näher beschrieben. Mobile Geräte stellen ein Sicherheitsrisiko für den Benutzer und für den Eigentümer vertraulicher Daten dar. Für die Vertraulichkeit der Verbindung sind daher sind kryptografische Verfahren erforderlich. Die Mobilfunknetze stellen verschlüsselte Verbindungen bereits auf Transportebene bereit. Die in Abbildung 3-5 unter dem Begriff Sicherheit zusammengefassten Komponenten stellen Funktionen bereit für die Verschlüsselung von Daten und für den Umgang mit Zertifikaten (die sogenannte Public Key Infrastruktur). Zertifikate sind ein wichtiger Bestandteil für einen sicheren mobilen Einsatz, speziell dann, wenn fremde Anwendungen bzw. fremde Dokumente auf das Endgerät geladen werden. Durch Zertifikate wird sichergestellt, dass solche Anwendungen (bzw. Dokumente) aus einer vertrauenswürdigen Quelle stammen, sowie dass der Code der Anwendungen (bzw. der Inhalt der Dokumente) unversehrt ist und nicht manipuliert wurde.

Basis

Symbians grundlegenste Komponente führt die Bezeichnung E32. E32 besteht aus dem Kernel und aus der Benutzerbibliothek (User Library). Der Kernel besitzt höhere Privilegien als jede andere Komponente. Die User Library, `euser.dll`, ist der am höchsten privilegierte User-Mode Code. Sie bietet Funktionen zu anderem User-Mode Code und zum kontrollierten Zugriff auf den Kernel. Abbildung 3-6 zeigt eine Übersicht.

Abb. 3–6
Kernel und User Library



Der Kernel selbst hat zwei Hauptkomponenten: In der *Kernel Ausführungsroutine* `ekern.exe` läuft privilegierter Code in einem Thread, der normalerweise im User-Mode ausgeführt wird. Diese kann daher von anderen höher-priorisierten Threads oder vom Kernel Server benachteiligt werden. Der *Kernel Server* ist der Hauptthread seines eigenen Prozesses und läuft immer privilegiert. Er ist der am höchsten priorisierte Thread im System. Er allokiert und de-allokiert kernelseitig Ressourcen, die entweder vom System selbst, oder von User-Mode Programmen benötigt werden. Der Kernel ist ein einziger Thread: er verarbeitet Anfragen nacheinander, also nicht-präemptiv.

3.1.5 Gerätetreiber

Systemgeräte wie der Bildschirm, das Keyboard, Digitizer (für den Stift), Sound Codec, Status LEDs, Spannungssensoren, Serielle Schnittstelle, CF Card, etc. werden alle von hardwarenahen (low-level) Gerätetreibern gesteuert. Es ist möglich, neue Geräte anzuschliessen und Treiber dafür zu schreiben. Normalerweise machen das die Hersteller der Geräte (Symbian OEMs). Symbiangeräte sind typischerweise nicht so leicht vom Endbenutzer zu erweitern wie PCs.

Kernel Ausführungsroutine und Anwendungsprogramm

Ein Gerätetreiber ist in mehreren Teilen implementiert. Die Kernel Ausführungsroutine unterstützt Gerätetreiber, so dass ein User Programm eine Anfrage an einen Treibercode senden kann, der auf Kernel Seite (entweder in der Kernel Ausführungsroutine oder im Kernel Server) läuft. Solche Anfragen initialisieren typischerweise eine Geräteoperation, oder teilen dem Treiber mit, dass das Programm darauf wartet, dass auf dem Gerät etwas passiert. Treiber behandeln auch Interrupts von Geräten und teilen dem User-Programm oder Kernel-

Programm mit, dass ein Ereignis eingetroffen ist, auf das das User-Programm oder Kernel-Programm wartet.

Interrupts

Die Interrupt Verarbeitung arbeitet auf zwei Ebenen: Den ersten Verarbeitungsschritt macht die *Interrupt Service Routine (ISR)*. Interrupt Service Routinen müssen sehr kurz sein und können nicht viele Aufgaben erledigen, da sie jederzeit aufgerufen werden könnten, sogar mitten in einer Kernel Server Operation. Normalerweise geben sie nur eine Bestätigung an das Gerät, das den Interrupt ausgelöst hat und setzen ein Flag, um den Kernel aufzufordern, einen sogenannten *Delayed Function Call (DFC)* für die weitere Verarbeitung zu starten. Abbildung 3-7 zeigt die beteiligten Komponenten.

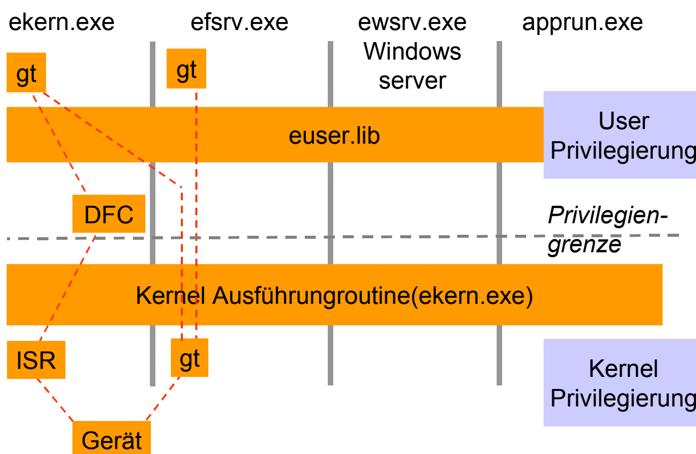


Abb. 3–7
Kernel und Gerätetreiber

Der Kernel führt den Delayed Function Call dann zu gegebener Zeit aus. Die Ausführung erfolgt sofort, falls gerade User-Mode Code ausgeführt wurde. Andernfalls erfolgt die Ausführung, sobald der Kernel die Privilegierungsgrenze zurück zum User Code überschritten hat. Delayed Function Calls können fast alle Kernel APIs verwenden.

3.1.6 Speicherverwaltung

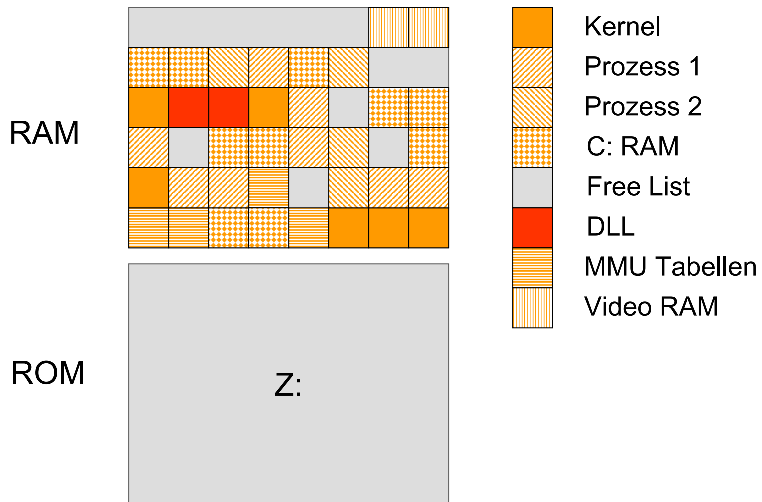
Wie bereits vorher angesprochen wird der Systemspeicher von der Memory Management Unit, der MMU verwaltet. Die Verwaltung des ROM ist einfach. Das ROM besteht vollständig aus Dateien, die in einem Verzeichnisbaum auf dem Laufwerk Z: liegen. Alles wird dabei auf feste Adressen abgebildet, so das die Daten jeder Datei ganz einfach gelesen werden können. Programme können direkt im ROM ausgeführt werden.

Die Verwaltung des RAM ist interessanter. Das physikalische RAM wird bei Symbian von der MMU in 4k Blöcke aufgeteilt. Jedem Block können die folgenden Inhalte zugeordnet werden:

- Der virtuelle Adressraum eines User-Prozesses. Hiervon kann es natürlich mehrere geben.
- Der virtuelle Adressraum des Kernel Server Prozesses.
- Die RAM Disk als C: Laufwerk. Auf diesen Bereich im RAM kann nur über den Fileserver Prozess zugegriffen werden.
- DLLs, die nicht von ROM geladen werden. Das RAM für diese DLLs wird nach dem Laden als Read-only markiert. Jede DLL erscheint für jeden Thread, der sie verwendet an exakt der gleichen Stelle.
- Mapping Tabellen der MMU. Die MMU ist darauf optimiert, diese Tabellen möglichst klein zu halten, es gibt aber praktisch keine Grenze für die Zahl von Prozessen und Threads in Symbian OS.
- Die Free-List enthält alle Blöcke, die noch nicht allokiert wurden.

Abbildung 3-8 zeigt eine Übersicht. Es gibt in Symbian OS keinen virtuellen Speicher mit einer großen Auslagerungsdatei, wie in PC Betriebsystemen. Jeder Block, der benötigt wird, wird also aus der Free-List entnommen. Wenn es keine leeren Blöcke mehr gibt, wird bei der nächsten Anfrage nach Speicher zwangsläufig ein „out-of-memory“ oder ein „disk-full“ Fehler auftreten.

Abb. 3–8 Speicherverwaltung in RAM und ROM



Wird ein ".exe" Modul gestartet, so wird ein neuer Prozess erzeugt, der einen einzelnen Haupt-Thread besitzt. Es können auch während der Laufzeit eines Prozesses noch andere Threads in diesem Prozess gestartet werden. Der Adressraum eines Prozesses enthält Bereiche für:

- systemübergreifenden Speicher: hierzu gehören das ROM und die in das RAM geladenen shared DLLs
- prozessübergreifenden Speicher: wie das Image des ".exe" Moduls und seine beschreibbaren statischen Daten
- Speicher für jeden Thread: einen sehr kleinen Stack und einen default Heap. Die maximale Größe des Heap kann vom Symbian OEM festgelegt werden und beträgt in der Regel 2 MBytes.

Don' t panic

Der Stack eines Thread kann nach dem Starten des Threads nicht mehr über die vorgegebene Grösse wachsen. Bei einem Stack-Überlauf löst ein Thread eine Systemreaktion aus (der Thread „panicked“) und wird sofort beendet. Die normale initiale Grösse des Stacks beträgt 12 kBytes. Der Stack wird für C++ Variablen in jeder Funktion verwendet. Man sollte also große Variablen möglichst auf dem Heap unterbringen.

Allokation von Speicher im Heap

Der Heap wird verwendet, wenn Speicher mit dem C++ Operator `new` oder über Funktionen der Benutzerbibliothek wie z.B. `User:Alloc()` allokiert wird. Falls möglich, wird der Speicher aus existiernden Blöcken allokiert, die bereits zuvor dem Heap zugeteilten waren. Falls das nicht geht, fordert der Heap-Manager zusätzliche Blöcke aus der Free-List des Systems an. Falls auch dies nicht klappt, gibt es einen "out-of-memory" Fehler.

Jeder Thread besitzt seinen eigenen Heap (default heap), welcher für die Allokation mit C++ `new` und die De-allokation mit `delete` verwendet wird. Das allokieren und de-allokieren ist recht effizient, wenn es auf dem lokalen und nicht mit anderen geteilten Heap geschieht. Wenn bei einer Allokation der Heap nicht wachsen muss, so werden nur sehr wenige Instruktionen benötigt. Es werden keine Privilegiengrenzen überschritten und es ist keine Synchronisation mit anderen Threads notwendig.

Stack

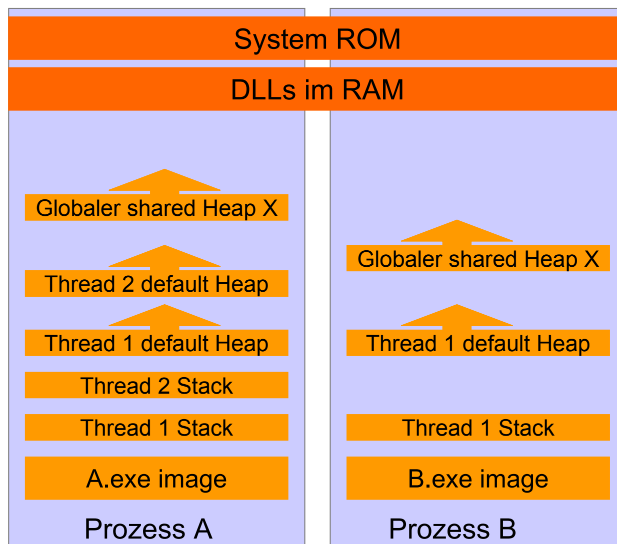
Auf den Stack sollte man wie gesagt nur kleine Objekte legen, wie z.B. Integer Werte oder Strukturen wie z.B. Rechtecke (`TInt x`, `TRect region`). Die meisten Objekte, vor allem die größeren, gehören auf den Heap: `CEikDialog* dialog = new CBeispielDialog`. Objekte, deren Klassennamen mit C beginnt können nur auf den Heap.

Objekte, deren Klassenname mit T beginnt, können entweder Membervariablen anderer Klassen oder Variablen auf dem Stack sein. Diese sollten auf gar keinen Fall auf den Stack, wenn man nicht sicher ist, dass sie recht klein sind. Besondere Vorsicht ist bei Dateinamen geboten (`TFileName filename`). Ein Dateiname ist 256 Buchstaben groß, das sind 512 Bytes im Unicode-Build von Symbian OS. Auf

einem 12 kBytes grossen Stack gibt es also nicht sehr viel Platz für viele Dateinamen!

Die Größe des Stack eines ".exe" Moduls kann man beeinflussen. Dies geht in Konsolenanwendungen, Servern oder in Programmen ohne GUI, aber nicht in EIKON Programmen (".app"), da diese mit der `apprun.exe` gestartet werden. Man kann auch die Stackgröße beim expliziten Starten eines Threads aus einem Programm heraus kontrollieren. Wenn man eine Anwendung hat, die einen Algorithmus implementiert, der einen sehr großen Stack benötigt, so muss man diesen Algorithmus entweder in einem eigenen ".exe" Modul oder in einem separaten Thread unterbringen.

Abb. 3–9
Speicherzuordnung für Prozesse und Threads



Threads besitzen voneinander unabhängige default Heaps und jeder Thread allokiert Speicher auf seinem eigenen Default Heap. Da aber alle diese Heaps im selben Adressraum des Prozesses liegen kann jeder Thread in einem Prozess auf Objekte eines anderen Threads auf dessen Heap zugreifen (passende Synchronisationsmethoden vorausgesetzt).

Achtung: Symbian OS unterstützt keine beschreibbaren statischen Variablen in DLLs!

DLLs unterstützen nur Read-only Variablen und Programmcode. Beschreibbare statische Variablen werden nur von ".exe" Modulen unterstützt. Dies erfordert ein wenig Disziplin beim Design von Symbian OS bzw. EPOC Anwendungen, was aber dank Objektorientierung einfach sein sollte. Die hauptsächliche Schwierigkeit besteht bei der Portierung von Code. Code für andere Betriebssysteme wie z.B. für PCs gehen oft davon aus, dass statische Variablen in DLLs erlaubt sind. Die einfachste Lösung für dieses Problem ist die Verwendung eines ".exe" Moduls für den portierten Code.

Warum werden aber beschreibbare statische Variablen in DLLs nicht unterstützt?

Beschreibbare, statische DLL's verbrauchen viel Speicher

Der Grund liegt in der minimalen Blockgrösse für die Allokation von Speicher im RAM. Jede DLL, die beschreibbare statische Variablen unterstützt, würde ein eigenes Stück RAM benötigen. Und das in jedem Prozess, der diese DLL verwendet. Angenommen, es gibt ungefähr 100 DLLs in Symbian OS, wobei die typische Anwendung ca. 60 davon verwendet. Angenommen, dass ca. 20 Anwendungen gleichzeitig laufen und zusätzlich noch 10 System Server (Fileserver etc.) für diese Anwendungen laufen. Die minimale Größe für die Speicherallokierung in der MMU beträgt 4 kBytes. Wenn nun jede DLL nur einen Block (4k) als beschreibbare statische Variable belegt, wäre der Speicherbedarf bei 4k x (20 Anwendungen + 10 Server Prozesse) x 60 DLLs = 7200 kByte RAM nur für die beschreibbaren statischen Variablen.

Auf einem PC ist das kein Problem, aber für ein Handheld Gerät mit teilweise nur 3,5 Mbyte RAM wie das Nokia 7650 ist das etwas zu viel. Das Problem ist vor allem aber auch, das viele Entwickler sich oft gar nicht bewusst sind, dass sie beschreibbare statische Variablen verwenden und damit den Speicherbedarf hochschrauben. Deshalb wurde beim Design von EPOC entschieden, dass beschreibbare statische Variablen in DLLs nicht zugelassen werden.

3.1.7 Fehlerbehandlung

Diese Anwendung wird wegen eines unerhörten Vorfalls geschlossen

Unter der Fehlerbehandlung (Exception Handling) versteht man die Reaktion des Systems auf Ausnahmesituationen (Exceptions). Ausnahmen müssen nicht unerwartet sein und als unerlaubte Vorgänge behandelt werden (etwa nach der Methode "diese Anwendung wurde wegen eines ungültigen Vorgangs geschlossen"). Einige solche Situationen lassen sich einkalkulieren und können vom Anwendungsprogrammierer abgefangen werden. Welche Fehlersituationen gibt es denn nun? Bei einem mit knappen Ressourcen ausgestatteten Gerät sind das beispielsweise Engpässe im Arbeitsspeicher (sogenannte "out-of-memory" Fehler). Im Grunde genommen kann der Speicher in jeder Programmzeile ausgehen, die ein neues Objekt im Arbeitsspeicher (Heap) konstruiert.

Eine andere Kategorie von Fehlern wären Probleme beim Öffnen von Dateien oder von Kommunikationsverbindungen. Die Datei könnte beispielsweise nicht vorhanden sein, oder bei einem Schreibzugriff von einer anderen Anwendung blockiert sein. Ein Verbindungswunsch für einen Kommunikationskanal kann fehlschlagen. Ähnlich

verhalten sich Fehler, die durch falsche Benutzereingaben verursacht werden. Allen diesen Fehlerkategorien ist gemeinsam, dass sie in aller Regel der Anwendungsprogrammierer abfangen oder bearbeiten möchte.

Planmässig aussteigen

Was kann der Anwendungsprogrammierer tun? Bei Engpässen im Speicher bleibt wahrscheinlich nichts anderes übrig, als die Anwendung planmässig zu beenden, und zwar mit einem Hinweis auf mangelnde Speicherkapazität an den Benutzer. Es sei denn, die Anwendung kann ohne die angefragten Resourcen sonst etwas sinnvolles tun. Bei gescheiterten Dateizugriffen oder Kommunikationswünschen könnte die Anwendung beispielsweise einen weiteren Versuch zu einem späteren Zeitpunkt starten. In jedem Fall ist aber eine planmässige Beendigung ohne Verlust von Daten (z.B. Benutzereingaben) möglich, sowie ein definierter Ausstieg mit einem klaren Hinweis an den Benutzer. Bei fehlerhafte Benutzereingaben ist ein Abfangen der Fehler und ein Benutzerdialog auf jeden Fall die Methode der Wahl.

Absturz

Was ist die Alternative zur Fehlerbehandlung durch den Anwendungs-programmierer? Wenn keine spezielle Reaktion vorgesehen ist, lösen Fehler eine Systemreaktion aus (eine sogenannte Panik des fehlerhaften Threads). Der Thread und der zugehörige Prozess werden vom Kernel gestoppt. Die Beendigung des Anwendungsprozesses ist zwangsläufig mit der Freigabe aller Resourcen des Prozesses verbunden. Man kann in diesem Fall auch vom "Absturz" der Anwendung sprechen. PC-Nutzer, denen so etwas bei der Erstellung eines Textdokumentes einmal passiert ist, wissen eine vernünftige Fehlerbehandlung zu schätzen. Was bei einem solchen Absturz ärgerlich ist, ist der Datenverlust.

Diese letzte Art der Fehlerbehandlung durch Absturz der Anwendung bleibt einer anderen Kategorie von Fehlern vorbehalten, nämlich den Programmierfehlern. Hier besteht die eigentliche Fehlerbehandlung darin, das Programm solange zu verbessern, bis Abstürze möglichst selten passieren. Um eine saubere Programmierung zu unterstützen, wird vom Symbian Emulator ein Absturz der Anwendung auch dann ausgelöst, wenn eine Anwendung "Müll" im Speicher (Heap) hinterlässt, d.h. Objekte, die nicht mehr vom Stack aus referenziert sind. Der Auslösezeitpunkt ist dabei das Ende des Programmes. Speicherhygiene ist immer dann wichtig, wenn Anwendungen lange laufen sollen. Das Schliessen der Anwendung beseitigt natürlich anwendungsbedingten Müll im Speicher, ebenso wie das Neustarten des Systems systembedingten Müll beseitigt. Für ein System, das als PDA fast nie ausgeschaltet wird, bzw. das als Mobiltelefon höchstens den Modus zwischen "smartem" Mobiltelefon und reinem PDA wechselt (den

sogenannten "Flight Mode", der die Mobilfunkschnittstelle abschaltet), sind die Anforderungen natürlich höher.

Speicherhygiene

Java-Programmierern mögen diese Fragestellungen nicht ganz fremd sein. Allerdings bietet Java für das Haushalten im Heap (die Speicherhygiene) und für die Fehlerbehandlung wesentlich mehr Komfort. Mit dem Löschen nicht mehr benötigter Objekte braucht sich der Java-Programmierer überhaupt nicht zu befassen. Der Garbage Collector entsorgt den Müll aus dem Arbeitsspeicher. Auch die Fehlerbehandlung ist durch Sprache bereits unterstützt (`try`, `catch`, `finally`) und wird somit auch vom Compiler überprüft. Da C++ als Programmiersprache von Symbian weniger komfortabel ausgestattet ist, gibt es für Symbian-Programmierer einige Konventionen zu beachten, auf die nun näher eingegangen wird.

Weder Symbian noch C++ verfügen über einen Garbage Collector. Zum Zeitpunkt der Schöpfung von Symbian war auch eine geeignete Fehlerbehandlung von C++ bzw. fehlertolerante Konstruktoren für konventionelle Entwicklungsumgebungen nicht verfügbar. Daher sind *Speicherhygiene* und *Fehlerbehandlung* hier etwas miteinander verknüpft. Bei der Konstruktion eines Objektes kann es zu einem Engpass im Speicher kommen, der in Fehlerbehandlung berücksichtigt werden muss. Ebenso kann das durch den Anwendungsprogrammierer erforderliche Löschen von Objekten im Zusammenhang mit Fehlersituationen zu Problemen führen.

Speicherfresser und Löcher

Wünschenswert ist, dass ein Objekt stets zusammen mit seiner Objektreferenz existiert (siehe z.B. das halloHeap Objekt in Abbildung 3-10), oder stets zusammen mit seiner Objektreferenz nicht existiert. Die beiden andern Zustände sind unerwünscht. Wird beispielsweise das Objekt nicht gelöscht, aber die Objektreferenz verschwindet, passiert weiter nichts. Es wurde allerdings unnötig Arbeitsspeicher allokiert. Wiederholt sich der Vorgang, so schwindet im Laufe der Zeit der verfügbare Arbeitsspeicher. Man hat einen "Speicherfresser" oder ein "memory leak". Gemeiner ist der Zustand, bei dem das Objekt gelöscht wurde, aber die Objektreferenz noch existiert und irgendwo hinzeigt, also möglicherweise in die Irre führt. Bei einem weiteren Löschen des Objektes gehen dem Prozess im Arbeitsspeicher irgendwelche Daten verloren, was zu sehr schwer nachvollziehbaren Folgefehlern führen kann.

Symbian verfolgt für C++ Programmierer zur Speicherhygiene und Fehlerbehandlung folgende Konzepte:

- *Fehlertolerante Konstruktoren* und *Konventionen* beim Erzeugen und Löschen von Objekten, wie z.B. den Cleanup Stack,

- eine *Fehlerbehandlung* für Funktionen, die planmässig aussteigen und in die Fehlerbehandlung laufen.

New (ELeave) und Cleanup Stack

Zu den *fehlertoleranten Konstruktoren* und *Konventionen* gehört der Konstruktor `new(ELeave)`. Dieser Konstruktor erzeugt zunächst ein temporäres Objekt, das erst im Erfolgsfall der Aktion an das gewünschte Objekt übergeben wird. Anderenfalls läuft er in eine vom System vorgesehene Fehlerbehandlung. Eine weitere Konvention ist die Verwendung eines sogenannten *Cleanup Stacks*. Referenzen von Objekte, die nicht Member anderer Klassen sind, kann man auf diesem separaten Stack ablegen. Der Cleanup Stack ermöglicht dann so etwas wie eine individuelle Garbage Collection.

Betrachtet man beispielsweise folgende Sequenz: ein Objekt `CX` mit der Objektreferenz `x` wird erzeugt, anschliessend wird dieses Objekt in einer Funktion verwendet, die in eine Fehlersituation läuft. In diesem Fall würde die Objektreferenz vom Stack verschwinden, ohne dass das Objekt gelöscht wurde. Mit Hilfe des Cleanup Stacks kann man die Sequenz geeignet klammern: Im Anschluss an die Erzeugung des Objektes wird erst eine weitere Objektreferenz auf dem Cleanup Stack erzeugt (`CleanupStack::PushL(x)`). Dann wird die Funktion aufgerufen. Schliesslich wird die Referenz auf dem Cleanup Stack wieder beseitigt (`CleanupStack::PopAndDestroy(x)`). Im Normalfall passiert also ausser der Erzeugung und Löschung einer überflüssigen Referenz nichts. Im Fehlerfall der Funktion steht der Fehlerbehandlung aber eine Referenz auf den Cleanup Stack zur Verfügung, die zum Löschen des erzeugten Objektes benutzt werden kann.

Eine weitere Massnahme sind Konventionen für den Anwendungsprogrammierer beim Löschen und bei der Erzeugung von Objekten. Zu den guten Sitten gehört es beispielsweise, vor der Erzeugung eines Objektes zu prüfen, ob das Objekt nicht bereits existiert. Beim Löschen von Objekten ist es in einigen Fällen sinnvoll, auch die Objektreferenz auf Null zu setzen, um Folgefehler zu vermeiden. Nähere Einzelheiten und Muster hierzu finden sich in den Anwendungsbeispielen des SDK, sowie in [Tasker].

Kennzeichen „L"

Funktionen, die planmässig aussteigen, laufen in eine *Fehlerbehandlung*. Solche Objekte werden namentlich gekennzeichnet mit einem grossen "L" am Ende. Die Bezeichnung "L" soll auf "Leave" hinweisen, also auf den Sachverhalt, dass dieses Objekt im Fehlerfall in die vorgesehene Fehlerbehandlung aussteigen wird. Das Kennzeichen "L" ist vergleichbar mit dem bei Java üblichen Hinweis "`throws exception`". Jeder Funktion mit Kennzeichen "L" wird ein mögliches Scheitern implizit unterstellt. Als funktionales Äquivalent von

"try-catch" in Java für die Fehlerbehandlung kann man `Trap()` bzw. `TrapD()` betrachten. Ein Anwendungsbeispiel wäre:

```
TrapD(error, FunktionL());
if (error) User::Leave(error);
```

User:: Leave für den planmässigen Ausstieg

Im Fehlerfall wird die Fehlerbehandlung `User::Leave()` aufgerufen und ihr der 32-Bit Fehlercode `error` vom Typ `TInt` übergeben. Diese spezielle Funktion `User::Leave()` ist allerdings bereits vom System vorgeleistet. Die Klasse `User` enthält einige statische Methoden mit Bezug zum Anwendungsthread und findet sich im API Guide des SDK unter System Static Functions. Mit der Methode `User::Exit()` beispielsweise würde der Anwendungsthread beendet. Die Methode `User::Panic()` verursacht einen Absturz auf Initiative der Anwendung mit Angabe einer Fehlerkategorie. `User::Leave()` ist die reguläre Methode für die Fehlerbehandlung bei Funktionen mit dem Kennzeichen "L". Es ist nicht nötig, dass Methoden zur Fehlerbehandlung unmittelbar aus einer Trap-Anweisung aufgerufen werden. Funktionen mit Kennzeichen "L" rufen in der Regel andere Funktionen mit Kennzeichen "L" auf. Es genügt, wenn weiter oben in der Sequenz eine Trap-Anweisung vorhanden ist. Im obigen Beispiel ist in der Trap-Anweisung also redundant, ein einfacher Aufruf der Funktion `FunktionL()` hätte genügt.

Vom Anwendungsprogrammierer wird in der Regel auch nicht erwartet, dass er Fehlerbehandlungen mit Trap-Anweisungen selber schreibt. Zur Speicherhygiene genügt der Cleanup Stack, für die weitere Fehlerbehandlung die Verwendung von Funktionen vom Typ "L" zusammen mit der Methode `User::Leave()`. Im API Guide des Symbian SDK finden sich unter Base/Memory Management/Using Cleanup Support/Exception Handling einige Erläuterungen und Beispiele, darunter dieses:

```
void doExampleL()
{
CExample* myExample = new CExample;
if (!myExample) User::Leave(KErrNoMemory);
// im Fehlerfall hier mit Indikation No Memory aussteigen

// weitere Verarbeitung

// aufräumen und beenden
delete myExample;
}
```

Falls die Konstruktion des Exemplars `myExample` nicht funktioniert hat, steigt diese Routine mit Aufruf der Funktion `User::Leave()`

aus und signalisiert als Ursache einen Speicherengpass mit dem Symbian Fehlercode `KErrNoMemory`. Bei normaler Rückkehr der Funktion bleibt der Fehlercode bei der Vorgabe `KErrNone` (kein Fehler). Auch den Nutzen des Konstruktors `new(ELeave)` gegenüber dem im Beispiel oben verwendeten Standardkonstruktors `new()` lässt sich an diesem Beispiel rasch zeigen. Bei Verwendung der alternative Programmzeile `CExample* myExample = new (ELeave) CExample;` für die Konstruktion entfällt die folgende Programmzeile im Beispiel mit der Abfrage der Fehlerbedingung und dem eventuellen Ausstieg komplett. Im Fehlerfall wird implizit die Fehlerbehandlung aufgerufen. Die Fehlercodes sind in einer Header-Datei e32sdt.h abgelegt. Eine Übersicht findet sich in der API Referenz des Symbian SDK unter System Wide Error Codes.

3.2 Text Deskriptoren

Verarbeitung von Strings

Deskriptoren sind für die Verarbeitung von Strings in Symbian OS zuständig. Sie erlauben ein sicheres und konsistentes Arbeiten mit Strings und Binärdaten gleichermaßen. Deskriptoren sind komfortabler als die Stringverarbeitung in C, aber erlauben dennoch die volle Kontrolle über den Speicherverbrauch eines Strings, im Gegensatz zu Java. Im Unterschied zu Java gibt es bei der Programmierung auf Symbian sehr wenige Vorgaben. Bei den knappen Resourcen eines kleinen Gerätes erscheint ein "manueller" Einfluss auf die Speicherverwaltung daher wünschenswert. Bei der Verwendung von Deskriptoren sollte man sich stets über die Speicheranforderungen im Klaren sein.

Einige Stichworte zur Verwendung von Deskriptoren: Deskriptoren sind für die Verarbeitung von Strings und Binärdaten vorgesehen. Sie stellen eine einheitliche Programmier-schnittstelle (API) zur Datenverarbeitung dar. Die Verwendung von Deskriptoren hilft, Fehler durch Speicher-Überläufe (Buffer-Overflows) zu vermeiden. Die Dekriptoren enthalten die Adresse und Länge der von ihnen repräsentierten Daten. Bei der Verarbeitung von Zeichen vereinfachen sie die Verwendung von Unicode-Zeichen. Für Binärdaten gibt es gesonderte Deskriptoren im 8 Bit Format.

In diesem Abschnitt wird auf die Verwendung von Deskriptoren im Zusammenhang mit ihren unterschiedlichen Anforderungen an die Speicherung eingegangen. Um den praktischen Einstieg zu erleichtern, wird dieser Abschnitt von einer kleinen Übung in Kapitel 3.3 begleitet.

3.2.1 Deskriptoren im Speicher

Deskriptoren (bzw. Strings) können in unterschiedlichen Speicherorten zu finden sein. Deskriptoren im Programmtext finden sich im Programmspeicher (d.h. im nicht beschreibbaren Speicher, also sinngemäss im ROM). Die anderen beiden Orte der Speicherung wären der Stack oder der Heap. In diesem Abschnitt werden die Deskriptoren im Zusammenhang mit ihrer Speicherung betrachtet. Abbildung 3-10 zeigt eine Übersicht.

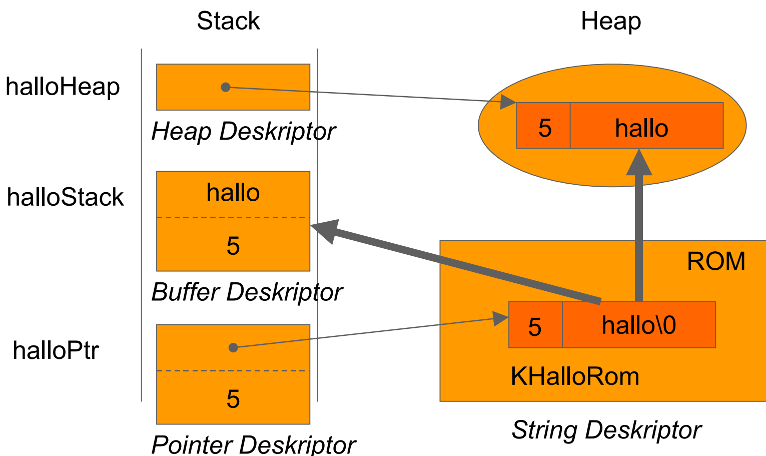


Abb. 3–10 *Deskriptoren in Stack, Heap und Programmspeicher*

String Deskriptor

Das Macro "_LIT" ermöglicht das Anlegen eines Strings im Programmcode. "_LIT" steht hierbei für „Literal". Die Anweisung `_LIT(KHalloRom, „hallo")` legt einen *String Deskriptor* an, auf den man über den Namen `KHalloRom` zugreifen kann. Der Inhalt des Deskriptors ist „hallo". Da der Deskriptor zu einer Programmzeile gehört, findet er sich im Programmspeicher.

Pointer Deskriptor

Einen *Pointer Deskriptor* auf diesen String bekommt man mit der Anweisung `TPtrC halloPtr = KHalloRom`. Ein `TPtrC` ist ein Objekt, welches einen Pointer und eine Länge enthält. Die Anweisung kopiert diese beiden Informationen in `halloPtr`. Das C im Namen von `TPtrC` bedeutet, dass der Inhalt, auf den `halloPtr` zeigt, nicht verändert werden darf, da er konstant ist. *Der Pointer Deskriptor* befindet sich im Stack.

Buffer Descriptor

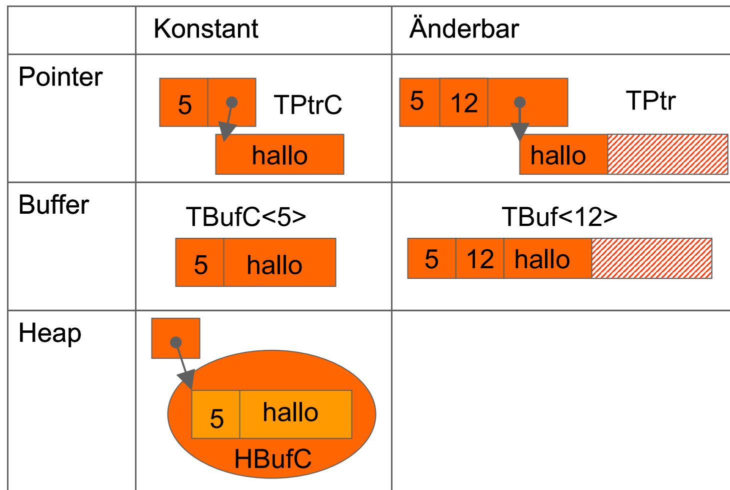
Man bekommt die Daten des Strings in den Stack, wenn man einen Puffer dafür erzeugt: `TBufC<5> halloStack(KHelloRom)`. `TBufC<5>` ist ein 5 Zeichen langer *Buffer Deskriptor*. Dieses Objekt enthält auch wieder Information über die Länge des Strings und den Inhalt. Die Daten sind wieder konstant, also nicht änderbar.

Um die Daten auf den Heap zu bekommen, muss man ein Heap basiertes Pufferobjekt erzeugen und die Daten dort hineinkopieren: `HBufC* halloHeap = KHalloRom().AllocLC()`. Ergebnis ist der String "hello" im Heap, auf den `halloHeap` verweist, siehe Abbildung 3-10. Einige Details hierzu finden sich in der Übung in Kapitel 3.3.

3.2.2 Deskriptor Klassen

`TPtrC`, `TBufC` und `HBufC` sind alle von der Klasse `TDesC` abgeleitet. Alle besitzen einen Pointer und eine Angabe über die aktuelle Länge des Inhalts. Alle erben sie die `const` Funktionen von `TDesC`. Die wichtigsten Unterschiede dieser Klassen liegen darin, wie sie Stringdaten enthalten oder darauf referenzieren. Außerdem unterscheiden sie sich in der Weise, wie sie initialisiert werden. `TPtr` und `TBuf` erben beide von `TDes`. `TDes` wiederum ist von `TDesC` abgeleitet. `TDes` besitzt zusätzlich noch eine maximale Länge und einige Funktionen, die über die `const` Funktionen von `TDesC` hinausgehen. Abbildung 3-11 zeigt das Beziehungsgeflecht.

Abb. 3–11 *Übersicht über die Klassen von Deskriptoren*



Das erste Maschinen-Wort (bei 32 bit Systemen die ersten 4 Bytes) enthält immer die Länge des Strings. Bei änderbaren Deskriptoren enthält das zweite Maschinen-Wort die maximale Länge des Strings. `TDes::MaxLength()` ist bei allen änderbaren Deskriptoren identisch implementiert. Die Position der Daten variiert allerdings je nach Typ des Deskriptors: `_LIT` erzeugt einen `TLitC`, welcher nicht von `TDesC` abgeleitet ist, während `_L` einen `TPtrC` erzeugt und ohne

Namen verwendet werden kann. Da `_L` allerdings zu den künftig aussortierten Elementen gehört (deprecated), sollte man lieber `_LIT` verwenden.

3.2.3 Deskriptor Funktionen

Die im letzten Abschnitt beschriebenen Deskriptor Klassen bringen viele Methoden zur Verarbeitung von Zeichenketten (Strings) mit. Diese Methoden erben die Deskriptoren von den Klassen `TDes` bzw. `TDesC`. Nach Kategorien sortiert ergibt sich folgende Übersicht:

- Direkter Zugang zu den Daten: `Ptr()` liefert einen Pointer auf die Daten zurück, die der Deskriptor repräsentiert. Die Daten können durch den Pointer nicht verändert werden.
- Eigenschaften der Daten: `Length()` hat als Rückgabewert die Länge der Daten des Deskriptors. `Size()` gibt die Datenmenge gemessen in Bytes zurück (bei 16 Bit Integers also der doppelte Wert wie bei `Length()`).
- Vergleich der Daten von Deskriptoren: `Compare()` vergleicht die Daten des Deskriptors mit dem Übergabeparameter der Methode. Wenn Länge und Inhalt übereinstimmen, sind die Zeichenketten identisch. Bei Übereinstimmung der Inhalte (bis zur Länge der kleineren Kette) können die Zeichenketten immer noch unterschiedlich lang sein.
- Zeichen lokalisieren: `Locate()` sucht nach einem vorgegebenen Zeichen in Vorwärtsrichtung und meldet die erste gefundene Position zurück.
- Daten finden: `Find()` sucht nacht einer vorgegebenen Zeichensequenz in Vorwärtsrichtung.
- Daten entnehmen: `Left()` extrahiert den linken Teil einer Zeichenkette zu einer als Parameter übergebenen maximalen Länge N, indem die Methode einen Pointer Deskriptor mit der vorgegebenen Länge zurückgibt. `Right()` arbeitet sinngemäss mit der Rückgabe eines Pointer Deskriptors für die letzten N Zeichen der Kette. `Mid()` extrahiert N Zeichen aus der Mitte der Kette.
- Die Eigenschaften der Daten verändern: `SetLength()` ändert die in Zeichen gemessene Länge des Deskriptors. `SetMax()` setzt die Länge veränderbarer Deskriptoren auf die maximal mögliche Länge (siehe TBuf<12> in Abbildung 3-11). `Zero()` setzt die Länge auf Null.
- Daten kopieren und ersetzen: `Copy()` kopiert Daten in den Deskriptor, die die alten Daten ersetzen. Die Länge des Deskriptors

wird an die neuen Daten angepasst, wobei die ursprüngliche Länge nicht überschritten werden darf. Weitere Methoden, die den Datenbestand verändern sind `Append()`, `Insert()`, `Delete()`, und `Replace()`. Bei `Replace()` sind im Vergleich zu `Copy()` unterschiedliche Längen möglich.

- Konversion von Text: `Capitalize()` verändert Zeichen in Grossbuchstaben (Capitels). `LowerCase()` konvertiert in kleine Buchstaben.
- Löschen von Daten: `Delete()` löscht Daten aus dem Deskriptor ab einer vorgegebenen Position und passt die Länge an. `Trim()` löscht Leerzeichen am Anfang und Ende der Zeichenkette.

Detaillierte Erläuterungen finden sich in den API Beschreibungen des SDK. Für die Verwendung von Deskriptoren für Binärdaten sollte man immer die 8-bit Versionen verwenden.

3.3 Übung: Deskriptoren in Symbian OS

Deskriptoren sind für das Arbeiten mit Strings unter Symbian OS unabdingbar. Sie sind im Vergleich zu C relativ komfortabel zu nutzen, erlauben aber im Gegensatz zu Java dem Entwickler die volle Kontrolle über den Speicherverbrauch eines Strings zu behalten. In dieser Übung beschäftigen wir uns mit dem Anlegen von Strings in C im Vergleich mit Symbian Literalen, sowie einigen Funktionen zur Bearbeitung von Strings.

3.3.1 Anlegen von Strings in C

Ein Beispiel in C

Um die Unterschiede zu C zu verdeutlichen, schauen wir uns kurz die Möglichkeiten zur Erzeugung eines Strings in C an. Ein einfacher String in einem C Programm ist ein Character Array, der bei Symbian OS dann im Programmspeicher vorliegt (sinngemäss im ROM, also in einem schreibgeschützter Bereich, der physikalisch in einem RAM oder ROM liegen kann):

```
static char hallorom[] = "hallo";
```

Um auf dem Stack einen Pointer auf diesen String zu bekommen, macht man folgendes:

```
const char* halloptr = hallorom;
```

Um den String selbst auf den Stack zu legen, deklariert man ein Character Array mit der entsprechenden Größe und kopiert danach den Inhalt des statischen Strings in das Character Array:

```
char hallostack[sizeof(hallorom)];
strcpy(hallostack, hallorom);
```

Den String kann man auf dem Heap ablegen in dem man entsprechend viel Speicher auf dem Heap allokiert und dann den Inhalt des Strings dort hinein kopiert:

```
char* halloheap = (char*)malloc(sizeof(hallorom));
strcpy(halloheap, hallorom);
```

3.3.2 Anlegen von Strings in Symbian

Symbian Konventionen

Das Macro _LIT für "Literal" ermöglicht das Anlegen eines Strings im Programmcode nach den Konventionen bei Symbian:

```
_LIT(KHalloRom, „hallo“);
```

Diese Programmzeile legt einen *String Deskriptor* an, auf den man über den Namen `KHalloRom` zugreifen kann. Der Inhalt des Deskriptors ist „hallo“.

Man bekommt die Daten des Strings in den Stack, wenn man zuerst einen Puffer (Buffer) dafür erzeugt:

```
TBufC<5> halloStack(KHelloRom);
```

`TBufC<5>` ist ein 5 Zeichen langer *Buffer Deskriptor.* Dieses Objekt enthält auch wieder die Information, über die Länge des Strings und den Inhalt. Die Daten sind wieder konstant, also nicht änderbar. Einen *Pointer Deskriptor* auf den String bekommt man folgendermaßen:

```
TPtrC halloPtr = halloStack;
```

Ein `TPtrC` ist ein 8kBytes großes Objekt, welches einen Pointer und eine Länge enthält. Der Befehl oben kopiert diese beiden Informationen in halloPtr. Das C im Namen von `TPtrC` bedeutet, das der Inhalt, auf den `halloPtr` zeigt, nicht verändert werden darf, da er konstant ist.

Um die Daten auf den Heap zu bekommen muss man nun einen Heap basiertes Bufferobjekt erzeugen und die Daten dort hineinkopieren:

```
HBufC* halloHeap = KHalloRom().AllocLC();
```

Diese Zeile bewirkt folgendes:

- `HBufC*` ist ein Pointer auf einen heap-basierten Buffer Deskriptor.
- Durch die Funktionsklammern nach `KHalloRom()` wird ein Operator aufgerufen, der `KHalloRom` in die Basisklasse aller Deskriptoren wandelt. Diese ist `TDesC`.

- AllocLC() allokiert einen HBufC der benötigten Größe auf dem Default Heap und kopiert den alten Deskriptorinhalt in den neuen. HBufC.AllocLC() legt HBufC* gleichzeitig auf den Cleanup Stack, damit das Objekt später gelöscht werden kann. Dies geschieht dann mit der Methode CleanupStack::PopAndDestroy().

3.3.3 Weitere Funktionen von Deskriptoren

Es gibt einige Funktionen, die das Arbeiten mit Strings erleichtern. Append() ermöglich z.B. die Erweiterung eines Deskriptors durch Anhängen eines zweiten Strings. Um zwei Strings auf dem Stack aneinander zu hängen, geht man folgendermaßen vor:

```
_LIT(KWeltRom, " welt!");
TBuf<12> halloWeltStack(KHalloRom);
halloWeltStack.Append(KWeltRom);
```

Append() verküpft Strings

Dies erzeugt zunächst einen zweiten Deskriptor mit dem Inhalt “welt!”. Danach wird auf dem Stack ein für beide Strings entsprechend großer Buffer Deskriptor angelegt und der Inhalt von KHalloRom dort eingefügt. Durch Aufruf der Methode Append(), werden die beiden Strings im Buffer Descriptor an einander gehängt. Will man die gleiche Übung nun auf dem Heap durchführen, so geht man folgendermaßen vor:

```
HBufC* halloWeltHeap=HBufC::NewMaxLC(KHalloRom().Length()+
KWeltRom().Length());
TPtr halloWeltAnhaengen=halloWeltHeap->Des();
halloWeltAnhaengen=KHalloRom;
halloWeltAnhaengen.Append(KWeltRom);
```

Zuerst wird hier ein neuer *Buffer Deskriptor* auf dem Heap angelegt. Dies geschieht mit der statischen Methode NewMaxLC() von HBufC. Dabei wird die Größe des neuen Deskriptors als Parameter mit übergeben. Der Deskriptor wird auch gleich auf den Cleanup-Stack gelegt und als Rückgabewert bekommt man einen Pointer auf den neuen Buffer Deskriptor. Da der Buffer Deskriptor nicht verändert werden darf, erzeugen wir einen neuen Pointer Deskriptor. Einen Pointer auf die Daten im Buffer Deskriptor auf dem Heap bekommen wir durch Aufruf der Methode Des() (steht für Deskriptor). Die Länge des änderbaren Pointer Deskriptors wird dabei auf die Länge des Buffer Deskriptors gesetzt. Nun können wir den Inhalt von KHalloRom dort hineinkopieren (der Operator = ist dafür entsprechend überladen). Als letztes verwenden wir die Append() Methode um den Inhalt von KWeltRom anzuhängen.

3.3.4 Aufrufen einer Funktionen zum Ändern von Deskriptoren

Wrapper Funktionen

Möchte man in einem Programm öfters zwei Strings aneinander hängen, so ist es sinnvoll, eine Funktion zu definieren, die dies erledigt. In unserem Beispiel definieren wir dafür die Methode `greetEntity()`:

```
void greetEntity(TDes& aGreeting, const TDesC& aEntity)
   {
      aGreeting.Append(aEntity);
   }
```

Die Funktion besitzt zwei Parameter: „`TDes& aGreeting`" und „`const TDesC& aEntity`". `TDes` ist die Superklasse aller änderbaren Deskriptoren. `TDesC` ist die Superklasse aller konstanten (dafür steht das C) Deskriptoren. `TDesC` besitzt die Funktion `Length()` um die aktuelle Länge des Strings und die Funktion `Ptr()` um die Adresse der Daten zu bekommen. Da wir nicht genau wissen können welche konkreten Klassen hier verwendet werden, sollte man in Funktionen für Descriptoren, die man nicht ändern möchte immer die Superklasse `TDesC` verwenden. Für zu ändernde Descriptoren verwendet man `TDes`. Nun können wir zwei Strings mit Hilfe dieser Funktion an einander hängen:

```
TBuf<12> halloWelt(_L("hallo"));
greetEntity(halloWelt,_L(" welt!"));
```

Hier wird der String „`hallo`" mit dem Makro `_L()` initialisiert. Dieses Makro erzeugt einen `TPtrC`. `GreetEntity` wird also mit einem `TBuf<12>` und einem `TPtrC` als Parameter aufgerufen. Wir haben nun folgende Deskriptoren kennen gelernt:

- `TPtrC`, `TBufC` und `HBufC` . die alle von der Klasse `TDesC` abgeleitet sind. Alle besitzen einen Pointer und eine Angabe über die aktuelle Länge des Inhalts. Und alle erben sie die `const` Funtionen von `TDesC`. Die wichtigsten Unterschiede dieser Klassen liegen darin, wie sie Stringdaten enthalten oder darauf referenzieren. Außerdem unterscheiden sie sich in der Weise, wie sie initialisiert werden.
- `TPtr` und `TBuf` erben beide von `TDes`. `TDes` wiederum ist von `TDesC` abgeleitet. `TDes` besitzt zusätzlich noch eine maximale Länge und einige Funktionen, die über die konstante Deskriptorklasse hinausgehen.

3.3.5 Vollständiger Programmcode der Übung

Programmtext

Hier nun der vollständige Programmcode zu Übersicht. Wie alle anderen Übungen lässt sich dieses Programm auch von der Web-Seite des Buches laden.

```
#include <e32base.h>
#include <e32cons.h>

LOCAL_D CConsoleBase* console;

void greetEntity(TDes& aGreeting, const TDesC& aEntity)
   {
      aGreeting.Append(aEntity);
   }

void mainL()
    {
   console->Printf(_L("Dieses Programm schaut man sich am besten im
Debugger an\n"));
   // string im program text
   _LIT(KHalloRom, "hallo");
   // string auf dem stack
   TBufC<5> halloStack(KHalloRom);
   //Pointer auf den stack string
   TPtrC halloPtrStack=halloStack;
   TInt l = halloPtrStack.Length();
   console->Printf(_L("Länge des Strings: %d"),l);
   // string auf dem heap
   HBufC* halloHeap=KHalloRom().AllocLC();
   //Pointer auf den string
   TPtrC halloPtrHeap=halloHeap->Des();
   // aneinanderhängen von strings auf dem stack
   _LIT(KWeltRom, " welt!");
   TBuf<12> halloWeltStack(KHalloRom);
   halloWeltStack.Append(KWeltRom);
   // aneinanderhängen von strings auf dem heap
   HBufC* halloWeltHeap=HBufC::NewMaxLC(KHalloRom().Length()+
KWeltRom().Length());
   TPtr halloWeltAnhaengen=halloWeltHeap->Des();
   halloWeltAnhaengen=KHalloRom;
   halloWeltAnhaengen.Append(KWeltRom);
   // verwenden einer funktion zum ändern von strings
   TBuf<12> halloWelt(_L("hallo"));
   greetEntity(halloWelt,_L(" welt!"));
   // _LIT literal
   const TDesC& halloRef=KHalloRom;
   // löschen der strings
   CleanupStack::PopAndDestroy(2);
   }
```

```
void consoleMainL()
    {
   // eine Konsole holen
   console=Console::NewL(_L("Hallo Text"),
                    TSize(KConsFullScreen,KConsFullScreen));
   CleanupStack::PushL(console);
   // funktion aufrufen
   mainL();
   // auf tastendruck warten
   console->Printf(_L("[ taste druecken ]"));
   console->Getch();
   // konsole beenden
   CleanupStack::PopAndDestroy();
    }

GLDEF_C TInt E32Main()
    {
   __UHEAP_MARK;
   CTrapCleanup* cleanupStack=CTrapCleanup::New();
   TRAPD(error,consoleMainL());
   __ASSERT_ALWAYS(!error,User::Panic(_L("PEP"),error));
   delete cleanupStack;
   __UHEAP_MARKEND;
   return 0;
     }
```