C: Programm Elemente

Schlüsselwörter

```
C89: auto      break     case      char      const     continue  default
     do        double    else      enum      extern    float     for
     goto      if        int       long      register  return    short
     signed    sizeof    static    struct    switch    typedef   union
     unsigned  void      while     volatile
C99: _Bool     _Complex  _Imaginary                    inline    restrict
C11: _Alignas  _Alignof  _Atomic   _Generic  _Noreturn _Static_assert
     _Thread_local
```

Basis Datentypen

Wertebereiche sind hardwareabhängig. Folgende Werte für 32-Bit Architektur.

```
char    1 Byte      -128 bis 127 oder 0 bis 255
int     4 Bytes     -2^31 bis 2^31-1
float   4 Bytes     -3.4·10^38 bis 3.4·10^38
double  8 Bytes     -1.79·10^308 bis 1.79·10^308

signed char                 1 Byte    -128 bis 127
unsigned char               1 Byte    0 bis 255
[signed] short [int]        2 Bytes   -32768 bis 32767
unsigned short [int]        2 Bytes   0 bis 65535
[signed] int                4 Bytes   -2^31 bis 2^31-1
unsigned [int]              4 Bytes   0 bis 2^32-1
[signed] long [int]         8 Bytes   -2^63 bis 2^63-1
[signed] long long [int]    8 Bytes   -2^63 bis 2^63-1
unsigned long [int]         8 Bytes   0 bis 2^64-1
unsigned long long [int]    8 Bytes   0 bis 2^64-1
long double         10 Bytes  -1.2·10^4932 bis 1.2·10^4932
```

Ein Pointer ist 8 Bytes.
Grösse eines Datentyps erhalten via **float.h** und **limits.h**

Standard Integer Typen (Typ-Aliase) via stdint.h
int8_t , uint8_t, int16_t , uint16_t, int32_t , uint32_t, int64_t , uint64_t

size_t aus stddef.h für sizeof und Array-Indizes.

Zuweisung: `int x; // x hat den Wert, was gerade im Speicher liegt.`

Explizite Typ Angaben

long (double)	l oder L
long long	ll oder LL
unsigned	u oder U
float	f oder F

3500l (oder 3500L), 123LLU, 123ULL , 0104270L , 0x8868L, 1.4142, 1e3F , 3.14159265359L

```
0101 = 1*8^2 + 0*8^1 + 1
\0: ASCII Code 0
```

```
12345  // signed int   (dec)   %d, %i
'A'    // unsigned int (char)  %u
'\101' // unsigned int (octal) %u
0101   // unsigned int (octal) %u
'\x41' // unsigned int (hex)   %u
0x41   // unsigned int (hex)   %u
1234.5 // double       (float) %f, %lf
1.6e-2 // double       (float) %f, %lf
```

Literale: Im Code eingefügte, unveränderliche Werte. Zeichenfolge, die zur direkten Darstellung der Werte von Basistypen (z. B. Ganzzahlen, Gleitkommazahlen, Zeichenketten, Funktionen) definiert bzw. zulässig ist:
`true, false, 1200, -12, 'a', "Zeichenkette", 12.e-34,` etc.

Bitoperatoren

Komplement: ~a Xor: a ^ b (Logisch: a != b)
Or: a | b Shift: a >> n, a << n
And: a & b

```
// unsigned int bit = 0 ... 31;
num |= (1 << bit);   //  setting a bit
num ^= (1 << bit);   // toggling a bit
num &= ~(1 << bit);  // clearing a bit
```

Präzision

```
double r; r = 3/2;     /* ergibt r = 1.0 */
int i; i = 3.1415;     /* ergibt i = 3   */
int i; i = 2.99 + 3;   /* ergibt i = 5   */
```

Boolean

Es gibt kein Datentyp boolean vor C99. Es gilt: **false wenn 0, sonst true**

```
typedef enum { false, true } bool;
bool b = false;
oder via stdbool.h
bool b = false;
```

C: Keywords

static: Nur in der eigenen Unit (oder Blockcode) sichtbar. Lebt die ganze Programm-Laufzeit. Statische Variablen können **nicht in anderen Quelldateien** verwendet werden. Eine statische Variable in einem Header File hätte zu Folge, dass jedes #include seine eigene Instanz dieser Variable hätte. Wird standardmässig mit 0 initialisiert (weil immer im globalen Kontext).

volatile: Wenn die Variable ausserhalb des Programms (Thread, Hardware, etc.) ändern kann. Der Compiler nimmt keine Optimierungen vor und leisst der Wert immer aus dem Speicher.

extern: Deklariert eine globale Variable, die in einer anderen Translation Unit deklariert ist. `extern` wird nur für die Deklaration einer Variable gebraucht. Eine globale Variable muss mit extern deklariert werden, wenn sie in einer anderen Source-Datei verwendet werden soll. Immer ein **code-smell.**

Variable definiert/deklariert **innerhalb** von Funktionen/Blöcken

Specifier	Speicherort	Impliziter Wert	Sichtbarkeit	Lebensdauer
`auto` (oder nichts)	Stack	Müll (undefiniert)	Bis zum Ende der **Funktion** oder des **Blocks**	Bis zum Ende des **Blocks**
`register`[1]	CPU Register			
`static`	Data Segment[3]	0		Von Begin bis zum Ende des **Programms**
`extern`[2]				

Variable definiert/deklariert **ausserhalb** von Funktionen

Specifier	Speicherort	Impliziter Wert	Sichtbarkeit	Lebensdauer
(nichts)[3]	Data Segment[4]	0	Bis zum Ende des **Moduls**	Von Begin bis zum Ende des **Programms**
`static`				
`extern`[2]				

C: Funktionen und Parameters

C ist eine **prozedurale** Programmiersprache. Funktionen statt Methoden. Methode = Funktion die einer KLasse zugehört: `obj.methode()` vs. `func()`

C Funktionen können keine Arrays zurückgeben. Stattdessen werden Pointers zurückgegeben.

```
// Parameternamen sind in der Deklaration optional:
int max(int, int);
int func(void) // Compiler prüft nach leere Parameterlist.
int func()     // Prüft nicht nach leere Parameterlist.
```

Call by reference: Die Adresse wird aber wie gewohnt by-value übergeben!

Tabellen und Zahlensysteme

Hexadezimal

Hexadecimal (Base 16)	Decimal (Base 10)	Binary (Base 2)	Hexadecimal (Base 16)	Decimal (Base 10)	Binary (Base 2)
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

Dezimal → Oktal

`0d1352` –binär→ $2^{10}(1024) + \sout{328}\ 2^{8}(256) + \sout{72}\ 2^{6}(64) + 2^{3}(8)$
`0b010|101|001|000` –decimal→ 2|5|1|0 –oktal→ `0o2510`

oder direkt

`0d1352` –oktal→ $2{\cdot}8^{3}(2{\cdot}512{=}1024) + \sout{328}\ 5{\cdot}8^{2}(5{\cdot}64{=}320) + 1{\cdot}8^{1} + 0{\cdot}8^{0}$

Oktal → Dezimal

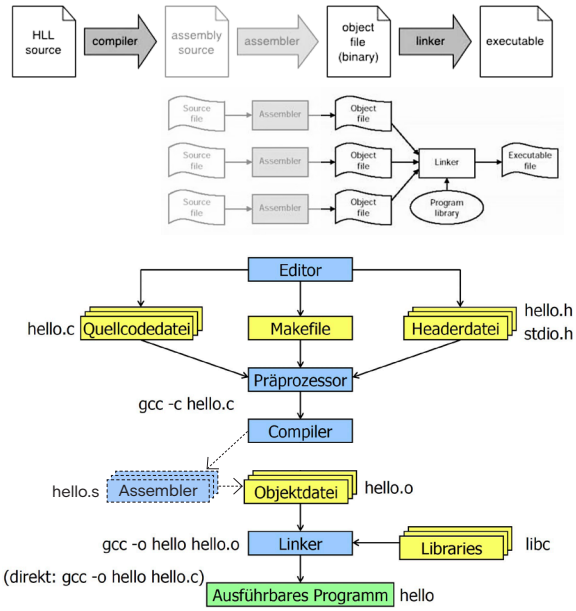
`0o4301` = $4{\cdot}8^{3} + 3{\cdot}8^{2} + 0{\cdot}8^{1} + 1{\cdot}8^{0} = 2241$

Bit-Nummerierung beginnt bei 0 von rechts. Bsp: **Bit 0**, **Bit 1**, etc. → 00**0**0

C: Modulare Programmierung

Ein Modul besteht aus einem .c-File und einem passenden .h-File.

Compliation Process



Header-Files

Da ein Header-File potentiell in mehreren Files mit #include eingebetet wird, darf ein Header File **keine Funktion-** und **Variablen-Definitionen** enthalten.
Header-Files sind wie **Interfaces,** die implementiert werden müssen.
Jeder Name darf **nur eine Definition** im gesamen Programm haben.

(EN) The idea of header files is that source files can be compiled individually so that large programs can be built modularly and complilation time is faster.

Vorteile: Logik und Implementation aufgeteilt, schnelle Komplizierzeit
Nachteile: Boilerplate

```
void f1();
void f2() { f()1; } // kann f1 aufrufen, obwohl später definiert
void f1() { ... }
```

1. Präprozessor

`gcc -E main.c -o main.i`
Führt Text Ersetzungen im Quellcode mittels Makroanweisungen (`#include, #define, #ifdef`) durch.

2. Compiler

`gcc -S main.i -o main.s`
Wandelt den Quellcode in **Objekt-** oder optional zuerst in Assemblydateien um. **Assembler-Code** ist im Wesentlichen eine «für den Menschen» lesbare Version von Objektcode.

3. Assembler

`gcc -c main.s -o main.o`

(EN) Creates object code. An object files (.o) is structured machine code (header, data segment, symbol table, etc.).

Objektdateien beinhalten generierten Machinencode und u.a. eine Liste der externen/importierten Symbole, die noch gelint werden müssen.

4. Linker

`gcc main.o -o main`
(EN) Linker combines object files into an executable. The idea of the linker is that source files can be compiled individually (per-file compilation) so that large programs can be built modularly and complilation time is faster (concer and divide).
Überprüft ob mit **extern** deklarierte Variablen irgendwo definiert sind.

Header Guards

Verhindert, dass eine Header-Datei mehrmals in eingebettet wird.

```
#ifndef FILE_H
#define FILE_H
// code ...
#endif
```

C: Arrays and Pointers

Arrays in Funktionsparameter

Ein Array kennt seine Länge nicht. Eine Funktion kann die Länge eines Arrays nicht ermitteln. Deshalb muss die **Länge** des Arrays immer **übergeben** werden:

```
void access(int array[], size_t n) {
  for(size_t i = 0; i < n; i++) {
    ...
  }
}
int a[100] = { 0 };
size_t n = sizeof(a)/sizeof(a[0]);
access(a, n);
```

End-Marke

Genannt **Sentry,** Sentinel oder Guard

Ein spezieller reservierter Wert, z.B. -1, markiert das Ende des Arrays.
Sentry Ansatz ist fehleranfällig und sollte sehr gut dokumentiert oder sogar **vermieden** werden.

```
#define DATA_SENTRY (-1) // or any other value which cannot exist otherwise
int array[] = { 1, 2, 3, DATA_SENTRY };
...
for (size_t i = 0; array[i] != DATA_SENTRY; i++) {
 ... array[i] ...;
}
```

Strings

Strings sind nichts anderes als **char-Arrays.**
Sentry-Wert ist **'\0'**, dafür wird eine **zusätzliche Speicherstelle** benötigt.

String-Literale:

```
char s[] = "Hello"; // char array (length == 6)
char* a = "Hello";  // char-pointer to a (read-only) array
```

Pointer Basics und Deklarationen

Syntax Regeln
Priorität: Gruppe → Array → Pointer → Type

Priorität	Assoziativität	Elemente	Beschreibung	Beispiel
1	-	`(…)`	Gruppierung	`int (*a[5])(void);`
2	links-rechts `((x[…])[…])`	`x[…]` `x(…)`	Array Funktion	`int a[5];` `int f(void);`

3	rechts-links
	`(*(const x))`
		`const`
		`volatile`
		`*`
			konstant
			flüchtig
			Pointer
				`int * const a;`
				`int * volatile a;`

```
char* str[20]     // jagged array: str is an [] of 20 char pointers
int b[20];        // b is a pointer to an array of 20 int
int (*b)[4]       // b is a pointer to 'int[][4]'
int (*b)[3][4]    // == int (*b)[][3][4]
int* (*b)[4]      // pointer to 'int* [4]'
int* a[]          // a is an array of int-pointers
int** a           // == int* a[]
int* a(int)       // Function returning an int-pointer
float (*a)(int)   // a is ptr to a function 'float func(int)'
float (*a[5])(int) // a is [] of 5 ptr to a function 'float func(int)'

int* a, b;  ==  int* a; int b; // b ist ein int und nicht ein int*
sizeof(ptr) == 4  // immer 4, unabhängig vom typ
void* vp = &d;    // Auf einen beliebigen Typ zeigen (ohne Cast)
int* ptr = NULL;  // Uninitialisierter Pointer
y[n][m] == *(*(y + n) + m)
```

pointer size = double (8 bytes)

Jagged Arrays

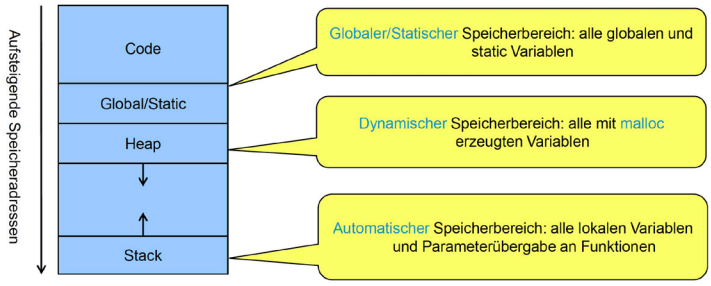
Eindimensionale Arrays von Pointern.

```
char* words[] = { "John", "Doe" }; // Strings haben untersch. Längen
words[0][4] == '\0' && words[1][3] == '\0'
printf("%zd", sizeof(words)); // 16 == 2 * 8 (pointer = 8 Bytes)
```

C: Dynamische Allozierung

Speicherorte von Daten

Der **Stack** garantiert, dass Daten immer «verdichtet» sind, **Fragmentierung** ist unmöglich. Beim **Heap** ist das Verhalten unberechenbar, was zu Fragmentierung und folglich zu neuen Memory Anfragen führt (*malloc → syscall).



Mit jedem Funktionsaufruf wird auf dem Stack automatisch Platz für lokale **automatische Variablen** reserviert (Stackframe). Der Compilierer berechnet den Platzbedarf. Grosse Datenmengen oder Daten mit unbekannter Grösse sollten auf dem Heap definiert werden. Sämtlicher Speicher wird immer freigegeben, wenn das Programm terminiert.

Segmentation Fault

Wenn versucht wird auf einen ungültigen Speicherbereich zuzugreifen.

Buffer-Overflow

Daten auf dem Stack weden überschrieben. Bei fehlerhafter Speicherverwaltung können Verwaltungsinformation auf dem Stack der eigenen oder anderer Funktionen zerstört werden.

```
char buf[5];
strcpy(buf, "12345\0"); // Kompiliererror
```

Anwender-Eingaben immer überprüfen, sichere Funktionen Vorbedingungen prüfen bevor Arrays beschrieben werden.

```
int main(int argc, char* argv[]) {
  char buf[5];
  strcpy(buf, argv[1]); // Kein Kompiliererror!
  return 0;
}
```

```
illegal hardware instruction ./main 12345
```

Stack-Overflow

Nicht mehr genügend Speciherplatz auf dem Stack.
Verhindern (Ansätze): Rekursion verbieten oder in der Tiefe limitieren.

Heap-Overflow

Verhindern: Fragmentierung z.B. durch Memory-Pools verhindern.
Ablauf anpassen damit nicht gleichzeitig zu viel Speicher benötigt wird.

Sichere Funktionen:

unsicher	besser
~~`gets`~~	`fgets()`: liest immer `\n` und informiert, falls der Buffer zu klein ist
~~`strcpy`~~, ~~`strcat`~~	`strncpy()` und `strncat()`: limitiert die Anzahl der zu kopierenden Zeichen
`sprintf`, `scanf`	immer mit Precision Specifiers verwenden (z.B. `"%.100s"`)
`malloc`, `calloc`, `realloc`	Erfolg prüfen (!= NULL), Array-Grenzen beachten bei Zugriff
`free`	genau ein mal verwenden pro allozierten Speicherbereich

Allgemeines Verhindern von Overflows:
Sichere Funktionen, Coding-Guidelines

C: Code Quality und Test

Testen bedeutet, Fehler zu finden und nicht die Korrektheit des Programms zu beweisen (ist per Definition nicht möglich).

Testhierarchie

System Test (testing the entire application)
> Integration Test (testing if different components work together)
> > Unit Test (testing individual components/modules)

Coding Guidelines: Vorgabe an Entwickler (um Fallstricke zu vermeiden)
Code Review: Nachweis der Anwendung von Guidelines

Werkzeuge

C Compiler: Mit Warnungs- und Debuggin-Flags compilieren. Schnell
Linter (Language Interpreter): Zusatzwerkzeug zum Compiler. Langsam.

Fallstricke

```
unsigned int a = 1;
int b = -1;
if(b < a)      // beides wird zu unsigned, if wird nie ausgeführt
if(b < (int)a) // korrekt
(i++) + a[i]   // undefiniert ob i++ oder a[i] zuerst aufgerufen wird
while(x = 1){ ... // kein error
while(1 = x){ ... // error. Besserer coding style
```

Core Dump

Ist ein Abbild des Speichers, z.B. zum Zeitpunkt wenn das Programm abstürzt. Der Dump wird üblicherweise im File «core» abgelegt und kann mit dem `gdb` (GNU-Debugger) gelesen werden:

```
gdb -c core executable.o
-c file – Use file as a core dump to examine.
Enable core dumps:  ulimit -c unlimited
Disable core dumps: ulimit -c 0
```

Wichtig: Damit bei Linux, das standardmaessig ABRT (Automatic Bug Reporting Tool) verwendet, ein «core» File erzeugt wird, muss:

```
sudo /bin/bash -c 'echo core > /proc/sys/kernel/core_pattern'
```

Bei SE-Linux verwendet (z.B. Fedora) muss auf Erzeugung des «core» Files verzichten werden.

Ausgabe analysieren:

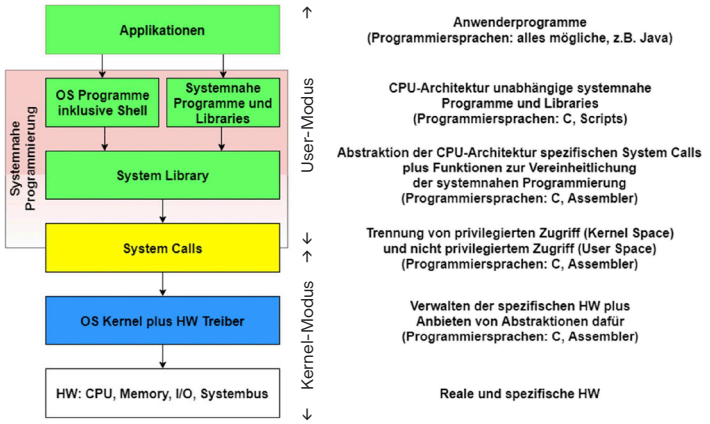
```
ulimit -c unlimited
sudo /bin/bash -c 'echo core > /proc/sys/kernel/core_pattern'
gdb ChildProcA7.e core  oder  gdb -c core executable.o
```

innerhalb von `gdb`

```
where
list
quit
```

Sys: Computer Systeme

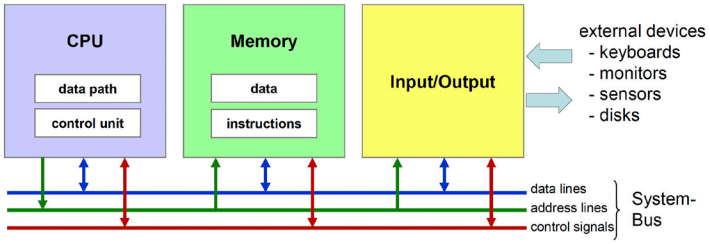
Schichtenmodel



Hauptaufgaben des OS
- Vermittler/Brücke zwischen Hardware und Programmen
- Verwalter Speicher (User-Space, Root-Space)

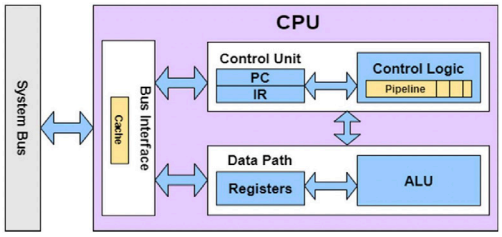
Elemente eines Computer Systems

CPU	**Central Processing Unit (Prozessor)**
Memory	**Speichert Daten und Instruktionen**
Input / Output	**Interface zu externen Devices**
System-Bus	**elektrische Verbindung der Komponenten**



System-Bus: Master-Slave

CPU



Aufgaben:
- Programmführung (Control Unit)
- Datenverarbeitung
- Master am Systembus

Programmführung (Control Unit): Programmausführung
- PC (Programm-Counter): Gibt an, wo im Memory die nächste Instruktion liegt.
- IR (Instruction-Register): Die aktuell ausgeführte Instuktion

Data Path: Rechnen mit Daten
- ALU (Arithemtic-Logic-Unit): Recheneinheit für Integer Daten.
- Register: Schneller Zwischenspeicher für Integer Daten.

Cache: Zwischengespeicherte Daten.

Memory

Memory-Zellen werden über Systembus-Adressen angesprichen

Aufgaben: Liefern von gespeicherten Daten

RAM (Random-Access Memory):
Gespeicherte Daten, nur solange Strom fliesst.
ROM (Read-Only Memory): Persistente Daten. Z.B. MAC-Adresse, Bootloader

I/O

Aufgaben:
- Anbindung des Computersystems an Peripherals (Maus, Tastatur, etc.)
- Lese-/Schreibe-Schnittstelle für externe Hardware (USB, HDD, etc.)

Systembus

Aufgaben: Verbindet zentrale Komponenten des Computersystems
Bidirektional: Read/Write, CPU verlangt Daten und liefert DAten über Bus.

Weitere Hardware Componenten:

- FPU (Floating-Point Unit): Recheneinheit für Gleitkommazahlen
- GPU (Graphics Processing Unit): Spezielle Graphik-Recheneinheit

Sys: Betriebssystem (OS)

Aufgaben:
- Verwalten von HW-Ressourcen (CPU, Memory, I/O, etc.)
- Ausführen von Tasks (Prozess oder Thread) → Siehe **Multitasking**

Sys: System Calls, System Libraries, System Shell

System Call

Fundamentales API zwischen Applikation und Kernen, bzw. Schnittstelle zwischen Kernel- und User-Mode. API für Aufrufe von User Tasks zum Kernel. Meistens muss dazu ein **Mode Switch** ausgeführt werden (Mode Switch ist gekapselt in System Call Funktionen).

Jede SysCall Funktion wird mit einer Nummer identifiziert. Diese Nummer und die notwendigen Argumente weden in CPU Registern geladen. Details des SysCalls werden im ABI (Application Binary Interface) definiert.

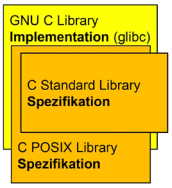
```
syscall(long number, ...)
```

Normalerweise mittels eines Wrappers (`glibc`) aufgerufen.

System Library

System-Libraries (POSIX) standardisieren und abstrahieren die OS-/Architek-tur-spezifischen SysCalls → unabhängige Programme lassen sich schreiben.

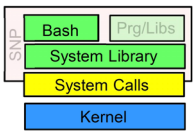
Bsp: `printf` ruft die `write` Funktion auf welche widerum ein Systemaufruf (`syscall`) macht. System-Calls ist das API vom OS: Programm → `syscall` → OS



Systemaufrufe sind «teuer».

```
stdio.h  unter /usr/include/stdio.h
```

System Shell



Ein User kann Anweisungen (durch Texteingabe) an das OS geben.

Shell: Systemnahes Programm (`/bin/sh`), welches die Kommandozeile interpretiert. Kann **Kernel-Funktionen** aufrufen, andere Programme starten und Zugriff auf die drei Standard I/O Streams **stdin, stdout** und **stderr** anbieten.

Prozess-ID der Shell: `echo $$`
Eine Eingabe wird von stdin gelesen und in Wörter (Tokens) aufgeteilt.

`/bin/sh`	System Startup Shell
`/bin/bash`	User Shell

POSIX

Portable **O**perating **S**ystem **I**nterface. Instruction manual on how to to create interoprobility between unix (and linux) operating systems.

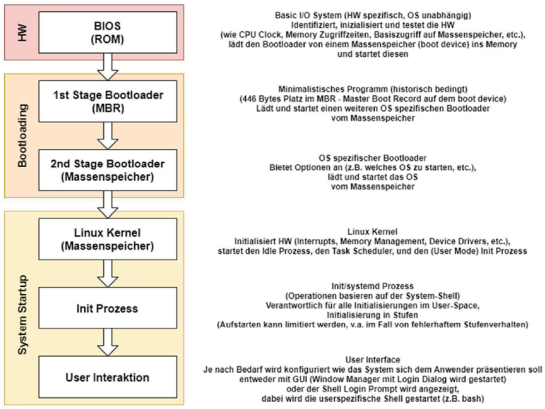
A Family of standards (collection of technical instruction manuals) specified by IEEE for keeping unix-like operating systems compatible with each other:

- c libraries, shell, utility applications (grep, ls, echo, cat, etc.)
- processes, signals, schedling, semaphores, message passing, threads, etc.
- memory, file (and directory) operations, I/O, port

Allows a programmer to make assumtion about (if a system is posix compliant) what is possible (and what **c library** is available) on that system.

Linux System Startup

BIOS → Bootloader → Kernel → Login → User-Interaktion



Sys: Filesystem

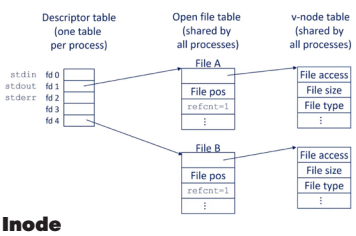
File: Directory, Hardware Devices, Zugriff auf netzwerk (Sockets) und Austuasch von Daten zwischen Programmen.

Filedescriptor

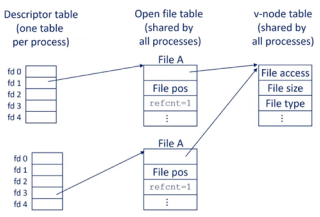
Filedescriptor = Integer, dass eine Nummer einem File zuweist. Wie ein «Ticket», dass man dem OS übergibt, um auf das File zuzugreifen.
OS weist jedem File eine systemweite eindeutige Nummer im Descriptor Table zu. Jedes öffnen eines Files, kreiert ein Eintrag im Open File Table.
refcnt = reference counter.

Der Kernel verwaltet geöffnete Files anhand einer **Integer-ID,** das ist der File Deskriptor (fd). System Calls teilen diese File Deskriptoren mit dem User Programmen, um mit tatsächlichen Files zu verknüpfen.

process with two files open



sharing files - different positions



Inode

(v-node für FreeBSD). Verwaltungseinheit eines Files. Table für File-**Metadata.** Ist die eigentliche Verwaltungseinheit eines Files und wird vom Kernel verwaltet. Jede Einheit hat eine eindeutige i-Nummer (ino). Enthält Informationen über letzter Zugriffszeitpunkt (access timestamp), Besitzer (owner), Länge, physikalischer Ort des Files auf dem Datenträger.

Ein Filename gehört nicht zum File, es ist nur ein Mittel, um ein File dem User zu präsentieren.

Files

= Unstrukturierter Byte-Stream oder Block von Bytes.
«Everything is a file».

Reguläre Files: Stream von Bytes.
File-Position: Bestimmte Byte-Stelle im Stream.
Verzeichnis (Directory): Ein File dessen Inhalt eine Liste von Inodes ist = Directory-Eintrag = **Hard-Link**
Symbolischer-Link: Ein File dessen Inhalt ein Pfad zu einem File ist (auch nicht existierende Files/Directories).

Spezielle Files

- **Character Devices:** Zugriff als Stream von Bytes (z.B. Tastatur)
- **Block Devices:** Zugriff als Byte-Buffer (z.B. Massenspeicher)
- **Named Pipes:** Für Inter-Process-Communication (IPC)
- **Sockets:** Kommunikation über ein Netzwerk

Der Kernel kümmert sich nicht um die **Synchronistation,** die User-Programme sind dafür zuständig.

Andere Filesysteme können mittels `mount` (`unmount`) eingebunden werden.

stdio.h

POSIX, portierbar

Öffnen/kreieren, schliessen, löschen, umbenennen
- `fopen(), freopen(), tmpfile()`
- `fclose(), fflush()`
- `remove()`
- `rename()`

Zustand und Zugriffs-Position abfragen und setzen
- `ftell(), fgetpos()`
- `fseek(), rewind(), fsetpos()`
- `feof(), ferror(), clearerr()`
- `setbuf(), setvbuf()`

Lesen und schreiben (unformatiert)
- `fread(), fwrite()`
- `fgetc(), getc(), getchar(), ungetc(), fgets()`
- `fputc(), putc(), putchar(), fputs(), puts()`

Lesen und schreiben (formatiert)
- `fscanf(), scanf(), sscanf()`
- `fprintf(), printf(), snprintf()`

Lesen und schreiben (formatiert, va_list Argument, <stdarg.h>)
- `vfscanf(), vscanf(), vsscanf()`
- `vfprintf(), vprintf(), vsnprintf()`

Error Message Output
- `perror()`

Verwendung nicht empfohlen
- `tmpnam()` : Eindeutigkeit nicht garantiert
- `sprintf(), vsprintf()` : Gefahr von array-bound-write
- `gets()` : nicht mehr unterstützt

I/O System Call

Hardware, nicht portierbar

Öffnen/kreieren, schliessen, löschen
- `open(), creat(),`
- `close()`
- `link(), unlink()`

Zustand abfragen und Zugriffs-Position setzen
- `stat(), fstat()`
- `lseek()`

Lesen und schreiben
- `read(), write()`
- `recv(), send()`

File Attribute
- `chown(), chmod()`
- `umask()`

Inter-Process-Communication[1)]
- `pipe()`
- `socket(), bind(), listen(), accept(), connect()`

Other Utilities
- `mount(), mknod(), fcntl(), mmap(), dup()`

Sys: Multitasking

Task

Prozess oder Thread. Eine Aufgabe, die von der CPU abgearbeitet wird. Batch: Anstehende Tasks werden hintereinander ausgeführt.

Batch-Ausführung: Anstehende Tasks werden **hintereinander** ausgeführt.

Multi-Tasking: Taks werden (quasi-)**nebeneinander** ausgeführt

Kontext-Switch

CPU zwischen Tasks hin und her. Quasi-parallele Ausführung.
Maps virtuelle Memory Location zur richtigen Memory.

Kooperaktiv: Jeder Task entscheidet selbst, wann er die Kontrolle an den nächsten Task abgibt.
Präemptiv: Kontrollübergabe wird erzwungen (durch Scheduler). Jeder Task wird vom OS nach einem Zeitinterval unterbrochen.
Scheduler: Enscheidet welcher Task an der Reihe ist Scheduling-Algorithmen:
- Round-robin: Einer nach dem anderen
- Priority-drive: Task mit höchster Priorität

Herausforderung: Kein Task soll ewig blockiert sein.

User-Mode und Kernel-Mode

Übergang mittels Interrupts, Traps
Kernel-Mode: Alle Operationen sind erlaubt, voller Zugriff auf die Hardware.
User-Mode: Zugriff auf System Libraries, User Programme, Syscalls

Virtuelles Memory

Gibt einer Applikation privaten Speicher. Kernel bietet dem Prozess die Illusion, dass er die alleinige Kontrolle über den gesamten Speicher und CPU hätte.

Memory Management (MMU): Hat die Aufgabe, das virtuelle Memory (logische Adresse) auf physikalische Adresse zu übersetzen.

Prozess

= Instanz eines Programms in Ausführung. Programm = Sequenz von Maschineninstruktionen (von CPU abgearbeitet). Eigenes virtuelles Memory. Schwergewichtig, verursacht beim **Kontext-Switch** Kosten (Umkonfigurieren des virtuellen Memories). Eigenes virtuelles Memory, wird vom OS verwaltet. Ein Prozess enthält mindestens ein Thread (Main-Thread).

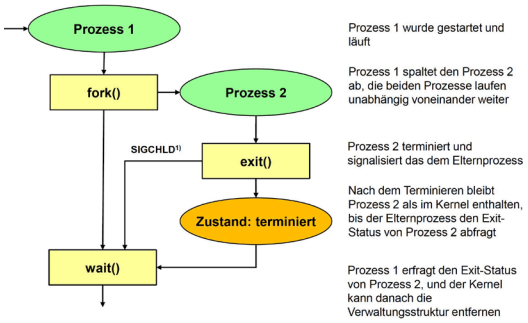
Jeder Prozess ist in einem definierten Ausführungszustand: running/active, ready, blocked (wartet z.B. auf HW Event), terminated.

fork(): Kopie des Elternprozesses. A system call that creates a child process which runs concurrently with the caller. Both processes will execute the next instruction after `fork()`. The child process **copies the resources** of the parent but uses the same pc (program counter), same CPU registers, and same open files.

exec(): Ersetzt einen laufenden Prozess durch einen neuen. Codezeilen im aufrufendes Programm unterhalb exec werden nur im Fehlerfall aufgerufen. Wird hauptsächlich im einem Fork aufgerufen.

system(): Execute the given line as a shell command.

Prozess Lebenszyklus:



Zombie: Child-Prozess ist bereits terminiert Zustand, Parent ist aber noch am laufen. Der Kindprozess exisiterit nur noch als Verwaltungsstrutur im OS, bis der Elternprozess wait() aufruft.

Will der Parent nicht warten, kann der Child-Prozess am Parent ein Signal zurückgeben und der Parent kann der Child-Prozess sofort beenden.

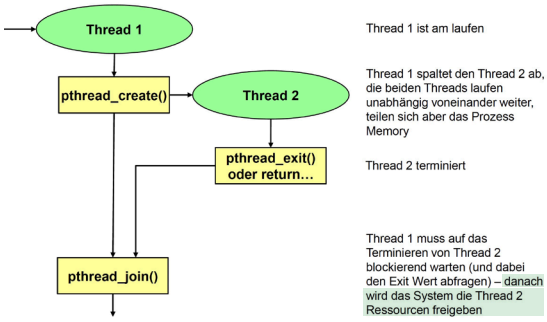
Thread

Leichtgewichtig. Exisitierung nur innerhalb eines Prozess. Threads greifen auf die selben Resourcen (e.g. den selben Array) zu bzw. Threads desselben Prozesses teilen sich die übergebenen Ressourcen: **Ressourcen werden geteilt** (und nicht kopiert) bzw. teils sich Memory mit Eltern-Prozess.

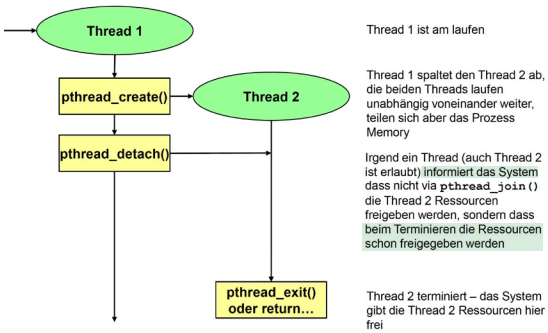
Fehlendes join/detach führt dazu, dass Thread-Resourcen nie freigegeben werden.

Multithreaded und fork(): Ein Prozess mit mehreren Threads ruft fork() auf → Lösung: Der Kindprozess muss unmittelbar nach dem fork() ein neues Image mit exec() starten, dadurch werden alle laufenden Threads vor dem start des neuen Images zerstört.

join() Lebenszyklus:



detach() Lebenszyklus:



Sys: Synchronisation

Problemstellung: Tasks (Threads oder Prozesse) sind auf gemeinsame Ressourcen abhängig.

Übersicht Synchronisationsarten

Mutex: Resource sharing, Mutual Exclusion, gegenseitiger Ausschluss, Sperren, frei/besetzt) → **Critical Section (Atomarer Zugriff):** Zeitlich exklusiver Zugriff auf gemeinsame Ressourcen ist gewährleistet.

Semaphore: Signaling, Signalisieren und Warten auf Tasks,

Mutex

Ist ein Objekt und ein spezifischer «Binary Semaphore». Zwei Operationen **lock** und **unlock.**

Probleme:
- Hohe Runtime-Kosten. Synchronisation kostet Zeit.
- Fehleranfällig, z.B. fehlendes Unlock bei Early-Return aus einer Funktion wegen Error-Handling
- Rekursion: Anzahl erlaubte Locks muss explizit gesetzt werden
  `#define MUTEXATTR PTHREAD_MUTEX_RECURSIVE`

Semaphoren

Ein Semaphore ist ein atomischer positiver Integer, die von verschiedenen Threads gemeinsam genutzt wird.

Zwei atomische Operationen:
sem_post(&sem): Erteile Freigabe (inkrementiert Wert).
sem_wait(&sem): Warte auf Freigabe (dekrimiert Wert).
Blockiert solange Wert = 0 und soblad >0 dekrimiert Wert sogleich wieder. wait() braucht passende Anzahl post().

Semaphore-Typen:

Counting Semaphore: Unlocks wenn ein gewisser Zählerwert erreicht is bzw. wartet auf mehrere Freigaben bzw. Posts.

Binary Semaphore: Zählerwert ist 0 oder 1.
Unlocks wenn einmal freigegeben (sem_post).

Probleme:
- Fehleranfällig: Nicht passende Anzahl Wait/Post z.B: wegen Early-Return.
- Rekursion: Nicht möglich.
- Deadlock, Livelock

POSIX Semaphore:
Unnamed Semaphore: In-Memory Semaphore (typischerweise zw. Threads)
```
sem_t sem;
sem_init(&sem, 0, initial_count); // in memory semaphore
sem_wait(&sem);
sem_destroy(&sem); // cleanup
```

Named Semaphore: Über Semaphore File (typischerweise zw. Prozessen)
```
sem_t* sem;
sem_open("/name", O_CREAT); // init IPC semaphore
sem_wait(sem);
sem_close(sem); // cleanup
sem_unlink("/name"); // remove file
```

Race Condition

Der Bereicht bzw. die Statements, welche geteilte Ressourcen modifizieren, dürfen nicht durch einen Kontext-Switch unterbrochen werden.

Unerwünschte Effekte

Deadlock: Gegenseitige Bockade von mehreren Locks. Ursache: Wenn die Reihenfolge der Zugriffe auf die Locks in Tasks unterschiedlch ist (Circular wait). **Livelock:** Tasks leisten keine produktive Arbeit mehr, weil sie so damit beschäftigt sind, Zugriffskonflikte anderer zu lösen und sich aus dem Weg zu gehen. **Starvation:** Ein blockierter Task kommt nie an die Reihe → OS sollte das verhindern. **Priority Inversion:** Task mit tieferer Priorität blockiert ein Task mit höherer → OS sollte das erkennen und verhindern

Producer-Consumer

```
Semaphore space_left = capacity(fifo);
Semaphore space_used = 0;
Mutex mutex;
// Producer
while(1) {
  item = produce_item();
  // blocks until space left for item
  wait(space_left); // block or dec
    lock(mutex);
      insert(fifo, item);
    unlock(mutex);
  post(space_used); // signal or inc
}
```

```
// Consumer
while(1) {
  // blocks until one item available
  wait(space_used); // block or dec
    lock(mutex);
      item = get(fifo);
    unlock(mutex);
  post(space_left); // signal or inc
  consume_item(item);
}
```

Sys: Inter Prozess Kommunikation

= Die Fähigkeit des Kerneln, Benachrichtigungen und daten zwischen parallel ausgeführten Prozessen auszutauschen. Motivation: Wie können Prozesse untereinander (mittels Events) Daten austauschen?

Übersicht

Unstrukturiert (Byte-basiert): Pipes, Sockets, Shared Memory, Shared Files
Strukturiert: Messages Queues

Implizite Synchronisation: Pipe, Message Queue, Socket
Explizite Synchronisation: Semaphore, Mutex, FIle Locking

Signals (legacy)

Asynchrone Notifikation (z.B, SIGKILL), Unterbrechung von Prozessen.
Ein Prozess kann einem anderen Prozess ein Signal (Integer) senden.
Signal = Zahl mit systemweit definierter Bedeutung. Kernel sorgt sich um die Benarchrichtigung.

Pipe

Unidirektional, unstrukturiert (Byte-Stream), FIFO Byte-Buffer (fixer Grösser).
Synchronisiert.

Anonymous Pipe
pipe() öffnet ein Paar von assoziierten File Deskriptoren. Danch muss mit fork() ein zweiter Prozess gestartet werden. Jeder der beiden Threads/Prozesse muss den einen File Deskriptor (übers Kreuz) mit close() schliessen, so dass die Pipe nutzbar ist. Dann kann write() bzw. read() genutzt werden.

Named Pipe (mit mkfifo)
```
$ mkfifo /tmp/my-named-pipe -m 06666
$ echo Hello > /tmp/my-named-pipe
$ cat /tmp/my-named-pipe && rm /tmp/my-named-pipe
Hello
```

Unterschied zu `touch` ist, dass mkfifo synchronisiert ist und nach dem Lesen das File automatisch geleert wird.

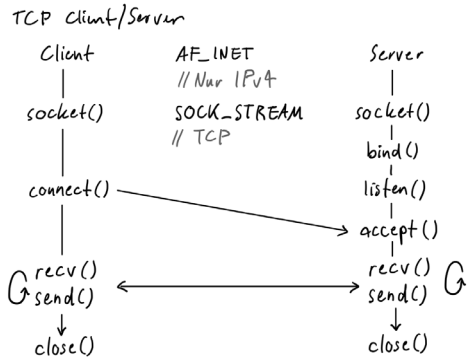
Message Queues

Bidirektional, Multi-Prozess, read/write synchronisiert, kann mehrere Leser und Schreiber haben. Jede Meldung wird beim Lesen aus der Queue entfernt. Die Message Queue exisitert auf dem virtuellen Filesystem `/dev/mqueue`. Jede Message hat eine **Priorität** (first-come, first-server bei gleicher Priorität). **Strukturiert** nach Anzahl-Messages *mal* Message-Grösse.

Socket

Daten sind **unstrukturiert** (Byte-Stream), **bidirektional, synchronisiert.** Netzwerkkommunikation durch die IP-Adresse der Netzwerkkarte und der Port Nummer.

Protokolle:
UDP: Verbindungsorientiert, Pakete ohne Fehler mit Sequenzkontrolle
TCP: Verbindungslos, Streams



Daemon

Dämonen oder englisch Daemons sind eine spezielle Art von Prozessen, die vollständig unabhängig arbeiten, d.h. ohne direkte Interaktion mit dem Anwender. Dämonen sind Hintergrundprozesse und terminieren i.A. nur, wenn das System heruntergefahren wird oder abstürzt. Dämonen erledigen meist Aufgaben, die periodisch ausgeführt werden müssen, z.B. Überwachung von Systemkomponenten, abfragen, ob neue Mails angekommen sind, etc.
Meist kann nur ein Dämon pro Aufgabe aktiv sein

Sys: Bash

Berechtigungen

```
ls -l
drwxr-xr-x. 1 domizai  staff  68 Jun 13 20:25 dirname
```

Filetype:
d directory
– regular file
l symbolic link

rwx-Permissions of: owner group other
r read: 4
w write: 2
x execute: 1
– disabled: 0

Owner: rwx = 4 + 2 + 1 = 7
Group: r-- = 4 + 0 + 0 = 4
Others: r-- = 4 + 0 + 0 = 4

```
chmod 744 file.txt
-rwxr--r--. 1 domizai  staff  68 Jun 13 20:25 file.txt
```

'.': Indicates a file with an SELinux security context (GNU `ls`).
'1': Hard link count (usually 1)
User owner, Group owner
'68': File size

List Processes

```
ps f – Display a tree view of parent to child processes
top – display sorted information about processes
```

Redirects und andere Kommandos

stdin: Kann mit **Filedeskriptor 0** referenziert werden. fully-buffered
stdiout: Kann mit **Filedeskriptor 1** referenziert werden. fully-buffered
stderr: Kann mit **Filedeskriptor 2** referenziert werden. nicht fully-buffered

```
... < file    # file to stdin
... > file    # stdout to file
... 1> file   # stdout to file
... >> file   # appends stdout to file
... 2> file   # stderr to file
... >& file   # stdout and stderr to file

cmd1 | cmd2   # stdout of cmd1 to stdin of cmd2

cmd1; cmd     # Ein Kommando nach dem anderen

cmd1 && cmd2  # Ruft cmd2 nur auf, wenn cmd1 Exit Code 0 hat
cmd1 || cmd2  # Ruft cmd2 nur auf, wenn cmd1 nicht Exit Code 0 hat

cmd &         # Startet ein Kommando als Background Prozess
```

Track Time (Performance)

```
$ time ./main
real   0m0, 193s
user   0m0, 193s
sys    0m0, 000s
```

ASCII Table

Dez	Hex	Okt		Dez	Hex	Okt	
0	0x00	000	NUL	64	0x40	100	@
1	0x01	001	SOH	65	0x41	101	A
2	0x02	002	STX	66	0x42	102	B
3	0x03	003	ETX	67	0x43	103	C
4	0x04	004	EOT	68	0x44	104	D
5	0x05	005	ENQ	69	0x45	105	E
6	0x06	006	ACK	70	0x46	106	F
7	0x07	007	BEL	71	0x47	107	G
8	0x08	010	BS	72	0x48	110	H
9	0x09	011	TAB	73	0x49	111	I
10	0x0A	012	LF	74	0x4A	112	J
11	0x0B	013	VT	75	0x4B	113	K
12	0x0C	014	FF	76	0x4C	114	L
13	0x0D	015	CR	77	0x4D	115	M
14	0x0E	016	SO	78	0x4E	116	N
15	0x0F	017	SI	79	0x4F	117	O
16	0x10	020	DLE	80	0x50	120	P
17	0x11	021	DC1	81	0x51	121	Q
18	0x12	022	DC2	82	0x52	122	R
19	0x13	023	DC3	83	0x53	123	S
20	0x14	024	DC4	84	0x54	124	T
21	0x15	025	NAK	85	0x55	125	U
22	0x16	026	SYN	86	0x56	126	V
23	0x17	027	ETB	87	0x57	127	W
24	0x18	030	CAN	88	0x58	130	X
25	0x19	031	EM	89	0x59	131	Y
26	0x1A	032	SUB	90	0x5A	132	Z
27	0x1B	033	ESC	91	0x5B	133	[
28	0x1C	034	FS	92	0x5C	134	\
29	0x1D	035	GS	93	0x5D	135	]
30	0x1E	036	RS	94	0x5E	136	^
31	0x1F	037	US	95	0x5F	137	_
32	0x20	040	SP	96	0x60	140	`
33	0x21	041	!	97	0x61	141	a
34	0x22	042	"	98	0x62	142	b
35	0x23	043	#	99	0x63	143	c
36	0x24	044	$	100	0x64	144	d
37	0x25	045	%	101	0x65	145	e
38	0x26	046	&	102	0x66	146	f
39	0x27	047	'	103	0x67	147	g
40	0x28	050	(	104	0x68	150	h
41	0x29	051	)	105	0x69	151	i
42	0x2A	052	*	106	0x6A	152	j
43	0x2B	053	+	107	0x6B	153	k
44	0x2C	054	,	108	0x6C	154	l
45	0x2D	055	-	109	0x6D	155	m
46	0x2E	056	.	110	0x6E	156	n
47	0x2F	057	/	111	0x6F	157	o
48	0x30	060	0	112	0x70	160	p
49	0x31	061	1	113	0x71	161	q
50	0x32	062	2	114	0x72	162	r
51	0x33	063	3	115	0x73	163	s
52	0x34	064	4	116	0x74	164	t
53	0x35	065	5	117	0x75	165	u
54	0x36	066	6	118	0x76	166	v
55	0x37	067	7	119	0x77	167	w
56	0x38	070	8	120	0x78	170	x
57	0x39	071	9	121	0x79	171	y
58	0x3A	072	:	122	0x7A	172	z
59	0x3B	073	;	123	0x7B	173	{
60	0x3C	074	<	124	0x7C	174	|
61	0x3D	075	=	125	0x7D	175	}
62	0x3E	076	>	126	0x7E	176	~
63	0x3F	077	?	127	0x7F	177	DEL

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NUL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL