EUROSplus

Systemdienste

Ausgabe März 1999

Ein Produkt eines Unternehmens der PHYTEC Technologie Holding AG

Informieren Sie sich:

	EUROPA	NORD AMERIKA
Adresse:	PHYTEC Technologie Holding AG Robert-Koch-Str. 39 D-55129 Mainz GERMANY	PHYTEC America LLC 255 Ericksen Avenue NE Bainbridge Island, WA 98110 USA
Angebots Hotline:	+49 (800) 0749832 order@phytec.de	+1 (800) 278-9913 order@phytec.com
Technische Hotline:	+49 (6131) 9221-31 support@phytec.de	+1 (800) 278-9913 support@phytec.com
Fax:	+49 (6131) 9221-33	+1 (206) 780-9135
Web Seite:	http://www.phytec.de	http://www.phytec.com

1 Einführung

Die Systemaufrufe von EUROS (Supervisor Calls, SVCs) gestatten den Zugriff auf die Dienste des Betriebssystems und stellen die Programmierschnittstelle (API) für den Anwender dar. Aus Sicht der Anwendersoftware sind sie als Erweiterung der Programmiersprache zu verstehen, mit deren Hilfe komplexe Datenstrukturen, die über den normalen Sprachumfang hinausgehen, einfach zu handhaben sind.

Zwar existieren Programmiersprachen, die bestimmte Multitasking-Funktionen schon integriert haben (z.B. Ada), diese haben aber den Nachteil, noch nicht sehr verbreitet zu sein. Mit den EUROS-Systemaufrufen als Handwerkszeug ist der Programmierer in der Lage, mit der Standardsprache C die Verständigung getrennter Prozesse untereinander (im EUROS-Sprachgebrauch ‘Tasks’) auf vielfältige Art und Weise zu implementieren. EUROS als Betriebssystem erledigt dann im Hintergrund die Aufgaben der Kommunikation, Synchronisation und Koordination für den Anwender.

Im Lauf der Jahre hat sich C als Standard-Programmiersprache in vielen Bereichen durchgesetzt. Da sich C hervorragend zur Implementierung von Betriebssystemen eignet und außerdem viele Anwender mit C vertraut sind, wurde EUROS fast vollständig in ANSI-C implementiert. Aus diesem Grund basiert die Systemaufruf-Schnittstelle von EUROS auf Aufrufen von C-Funktionen.

Auf Anfrage kann eine Sprachenschnittstelle für andere Programmiersprachen wie ‘Pascal’ oder ‘Fortran’ erstellt werden.

Die C-Funktionen sind in einer Funktionsbibliothek zusammengefaßt. Die Funktionen der Bibliothek können wie bei jeder anderen Funktionsbibliothek durch Einbinden von Header-Dateien benutzt werden. Beim Erzeugen einer Anwendung werden vom Linker nur die Teile der Bibliothek zur Anwendung gebunden, die tatsächlich benötigt werden.

Das Betriebssystem EUROS setzt sich aus mehreren Funktionsbibliotheken zusammen. Die verschiedenen Funktionsbibliotheken sind die ‘Software-Komponenten’, aus denen EUROS gebildet wird. Diese Komponenten stellen logische Einheiten des Betriebsystems dar, die jeweils getrennt initialisiert werden und eigene Datenbereiche verwalten. Die Kapiteleinteilung dieses Handbuchs trägt dieser Aufteilung Rechnung und faßt die Dienste der einzelnen Komponenten jeweils zusammen.

Eine weitere Unterscheidung der Systemkomponenten gibt es nicht.

Die Systemaufrufe sind für Tasks (z.B. die Anwendung) und Treiber identisch. Die Beschreibung der Systemdienste gilt für beide gleichermaßen, auch wenn an vielen Stellen der Einfachheit halber nur von einer Task als anforderndem Programm gesprochen wird. Es existiert eine Reihe von Systemaufrufen, die speziell für die Belange von Treibern implementiert wurden.

Um dem Programmierer von Applikationen den Umgang mit den Systemaufrufen zu erleichtern und um die Systemdienste trotz ihrer Vielzahl überschaubar und einfach zu gestalten, wurden sie nach folgenden Regeln aufgebaut.

- Die Namen der Systemaufrufe bestehen aus zwei Teilen.
  - Der erste Teil ist ein Kürzel zur Kennzeichnung des Objekts oder der Komponente, mit der gearbeitet wird. Zum Beispiel `Object` für allgemeine Objekt-Dienste, `Task` für Task-Dienste und `Mailbox` für Mailbox-Dienste bzw. `Microkernel` für einen Dienst, der die Komponente Mikrokernel im allgemeinen betrifft.
  - Der zweite Teil ist ein Kürzel zur Kennzeichnung der Funktion, die der Systemaufruf bewirkt. Zum Beispiel `Create` für Dienste zum Erzeugen von Objekten, `Init` für Initialisierungs-Dienste, usw.
- Die Art und Reihenfolge der Parameter ist bei gleichartigen Systemdiensten für alle Objekte gleich. Der Programmierer muß sich also nur an ein einziges Schema gewöhnen, um die EUROS-Systemaufrufe zu beherrschen.

Die folgenden Abschnitte stellen die EUROS-Systemdienste als Übersicht tabellarisch zusammen: geord-

net nach Komponenten und Systemobjekttypen. Die Parameter sind in einer allgemeinen Auflistung anschließend erläutert.

2 Mikrokernel-Systemdienste

Der Mikrokernel stellt die unterste Schicht des Betreibssystems dar. Er verbirgt die Hardwareeigenschaften der Umgebung vor der Anwendersoftware und den restlichen EUROS-Komponenten. Der größte Teil des Mikrokernels ist jedoch hardwareunabhängig und dient zur Verwaltung der Systemobjekte. Alle anderen EUROS-Komponenten setzen den Mikrokernel voraus. Der Mikrokernel stellt auch ohne die anderen EUROS-Komponenten bereits ein EUROS-Minimalsystem dar.

2.1 Hardware-Initialisierungs-Dienste

Damit anwenderspezifische Initialisierungen für die Hardware-Bausteine einer Hardware-Plattform vorgenommen werden können, bietet EUROS die folgende Aufrufschnittstelle an.

Anwenderspezifische Hardware-Initialisierungen durchführen:

```
HardwareInit ();
```

2.2 Initialisierungs- und Statusfunktionen

Die Komponente Mikrokernel stellt eine Reihe von Initialisierungs- und Statusfunktionen zur Verfügung, die den Start des Betriebssystems vorbereiten und dem Anwender bestimmte Statusinformationen liefern. Die Funktionen sind in der Header-Datei `mk_prot.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Mikrokernel initialisieren:

```
Status = MicrokernelInit ();
```

Status des Mikrokernels abfragen:

```
Status = MicrokernelStatus (pStatus);
```

Status des Mikrokernels ausgeben:

```
DumpMicrokernelStatus ();
```

Betriebssystem starten:

```
EurosStart ();
```

Versionsnummer von EUROS abfragen:

```
Status = EurosVersion ();
```

Aktuelle `errno`-Fehlermeldung in Klartext ermitteln:

```
Status = GetErrnoStr (pBuffer);
```

2.3 Panik-Konsole-Dienste

Die Komponente Mikrokernel enthält einige elementare Ein-/Ausgabefunktionen, die unmittelbar auf der Systemhardware aufsetzen und für Systemmeldungen bzw. für den Notfall bestimmt sind. Sie ermöglichen dei Bedienung der Systemkonsole, ohne Funktionen des I/O-Systems zu benutzen. Die Funktionen sind hardwareabhängiger Bestandteil des Mikrokernels und arbeiten üblicherweise im Polling-Modus, d.h. ohne Interruptbetrieb des Schnittstellenbausteins. Aus diesem Grund setzen sie keine weitere Systemumgebung voraus und arbeiten auch unter stark eingeschränkten Bedingungen. Die Eingaben bzw. die Ausgaben erfolgen über die konfigurierten Standard-Kanäle. Die Funktionen sind in der Header-Datei `mk_prot.h` deklariert. Die folgenden Panik-Konsole-Aufrufe sind implementiert:

Panik-Konsole initialisieren:

```
Status = PanicInit ();
```

Ein Zeichen einlesen:

```
Character = GetChar ();
```

Ein Zeichen ausgeben:
```
Status = PutChar (Character);
```
Not-Ausgabe einer Textzeile:
```
Panic (pText);
```
Not-Ausgabe einer ganzen Zahl in Dezimaldarstellung:
```
PanicNum (Value);
```
Not-Ausgabe einer ganzen Zahl in Binärdarstellung:
```
PanicBin (Value, MinDigits);
```
Not-Ausgabe einer ganzen Zahl in Hexadezimaldarstellung:
```
PanicHex (Value, MinDigits);
```
Not-Ausgabe einer 64-Bit Zahl in Hexadezimaldarstellung:
```
PanicUInt64 (Value, MinDigits);
```
Not-Ausgabe eines Datenzeigers:
```
PanicDPtr (pData);
```
Not-Ausgabe eines Codezeigers:
```
PanicCPtr (pCode);
```
Not-Ausgabe einer Textzeile ähnlich `printf`:
```
PanicPrintf (pFormat, ...);
```

2.4 Dienste für die Zustandsübergänge

Die Komponente Mikrokernel stellt einige Funktionen zur Durchführung von Zustandsübergängen zur Verfügung. Die Funktionen sind in der Header-Datei `int.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Zustandsübergang von `I` nach `N` durchführen:
```
IntI2N ();
```
Zustandsübergang von `I` nach `S` durchführen:
```
IntI2S ();
```
Zustandsübergang von `I` nach `X` durchführen:
```
IntI2X ();
```
Zustandsübergang von `N` nach `S` durchführen:
```
IntN2S ();
```
Zustandsübergang von `N` nach `X` durchführen:
```
IntN2X ();
```
Zustandsübergang von `S` nach `X` durchführen:
```
IntS2X ();
```

2.5 Interrupt-Dienste

EUROS behandelt auch die Interrupts als dynamische Systemobjekte. Die Systemaufrufe zur Behandlung von Interrupts erlauben unter anderem eine effiziente Interruptverarbeitung auf Task-Ebene. Dadurch erübrigt sich in vielen Fällen die Entwicklung eines Treibers zur Bearbeitung eines Peripheriegerätes. Dies ist für Minimalkonfigurationen interessant: Es kann unter Umständen auf die Einbindung des Ein-/Ausgabe-Systems (IOS) verzichtet werden; daraus resultiert ein geringerer Speicherplatzbedarf.

Der Einsatz von Port-Treibern zur Interruptbearbeitung ist dann sinnvoll und notwendig, wenn eine ganze Klasse von Peripheriegeräten mit einem gemeinsamen Treiber und parallel arbeitenden Geräteeinheiten installiert werden soll (Koordinierung der Einheiten untereinander, wiedereintrittsfähiger Code, keine - bzw. koordinierte - statischen Datenbereiche) oder aber ein Anschluß von Peripheriegeräten an das Dateiver-

waltungssystem (FMS, File-Management-System) vorgesehen ist. Bei begrenzten Hardware-Konfigurationen genügen die Interrupt-Dienste des Mikrokernels.

Es wird zwischen 'Interrupt-Handlern' unterschieden, die in den Systemzuständen `I` und `N` sehr schnell zur Ausführung kommen, jedoch keine Systemaufrufe absetzen können, und solchen, die im `S`-Zustand ausgeführt werden und Zugriff auf fast alle Systemdienste haben. Einer Task, die von einem Interrupt direkt gestartet wird - bezeichnet als 'Interrupt-Task' - steht dagegen die ganze Palette der Systemaufrufe zur Verfügung.

Die Funktionen sind in der Header-Datei `int.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Interrupt-System sperren:
```
IntDisable ();
```
Interrupt-System freigeben:
```
IntEnable ();
```
Einzelne Interrupt-Quelle sperren:
```
Status = IntSourceDisable ();
```
Einzelne Interrupt-Quelle freigeben:
```
Status = IntSourceEnable ();
```
Interrupt-Handler anmelden:
```
Status = IntSetVector (Vector, pFunc);
```
Interrupt-Handler für einen unerwarteten Interrupt anmelden:
```
Status = IntUnexpected (Vector);
```
Adresse eines Interrupt-Handlers ermitteln:
```
pIntFunc = IntGetVector (Vector);
```
Interrupt-Vektor als belegt markieren:
```
Status = IntMarkAsUsed (Vector);
```
Interrupt-Vektor als frei verfügbar markieren:
```
Status = IntMarkAsUnused (Vector);
```

2.6 Prolog-Dienste

Eine mit einem Interrupt verbundene Behandlungsroutine wird nach Eintreffen des ausgewählten Interrupts gestartet. Die in diesem Augenblick gültige Systemumgebung ist für die Behandlungsroutine unbekannt. Eine Versorgung mit Parametern in Registern oder auf dem Stack ist nicht möglich. Dabei ist es sehr oft wünschenswert nach dem Eintreffen eines Interrupt eine definierte Umgebung vorzufinden. Dies könnte beispielsweise die Adresse eines Puffers sein, in den die zu empfangenden Daten zu transferieren sind oder im Falle eines Treibers die Adresse seiner Datenstruktur.

Die Bestimmung einer Interrupt-Umgebung ist nur dann möglich, wenn beim Starten der Interrupt-Handler selbst dafür Sorge trägt. Dies wird erreicht, indem zum Zeitpunkt der Verbindung der Behandlungsroutine mit dem Interruptvektor ein Vorspann zwischengeschaltet wird, der den CPU-Kontext sichert und bestimmte Register oder den Stack mit vordefinierten Werten initialisiert. Nicht die Interrupt-Behandlungsroutine selbst, sondern dieser Vorspann, wird direkt mit dem Interruptvektor verbunden. Auf diese Weise findet die Interrupt-Behandlungsroutine eine initialisierte Umgebung vor, mit der sie nach dem Vorspann aufgerufen wird.

Dieser Mechanismus wird im EUROS-Sprachgebrauch als Interrupt-Prolog bezeichnet.

Die Funktionen sind in der Header-Datei `int.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Prolog ohne Anbindung an einen Interrupt-Vektor erzeugen:

```
pCode = PrologCreate (ExecMode, pFunc, pArgs, pRegBank);
```

Prolog mit Anbindung an einen Interrupt-Vektor erzeugen:

```
pProlog = PrologAttach (Vector, ExecMode, pFunc, pArgs, pRegBank);
```

Prolog löschen:

```
Status = PrologDelete (pCode);
```

Prolog von einem Interrupt-Vektor abkoppeln:

```
Status = PrologDetach (pProlog);
```

2.7 Timer-Dienste

Die Timer-Dienste stellen Funktionen zur Verfügung, mit deren Hilfe die Integration von Timer-Bausteinen (Zeitgeber) in einer EUROS-Konfiguration vorgenommen wird. Die Aufruf-Schnittstelle ist zwar allgemeingültig gehalten, die ausführenden Routinen sind jedoch hardwareabhängig.

EUROS benötigt mindestens einen Timer, der als System-Timer bezeichnet wird. Dieser Timer dient dazu, die zeitbezogenen Aufträgen durchzuführen und die Wartezeiten bei Systemaufrufen mit `ExecMode = WAIT` zu überwachen.

Sind in einem System mehrere Timer-Bausteine vorhanden, so kann ein weiterer Zeitgeber dazu genutzt werden, die Watchdog-Dienste zu versorgen. Dieser Timer wird als Auxiliary-Timer bezeichnet. In diesem Fall müssen die Aufrufe des System-Timers - Abkürzung `SysTimer` - und die Aufrufe des Auxiliary-Timers - Abkürzung `AuxTimer` - verwendet werden.

Die Funktionen sind in der Header-Datei `timer.h` vereinbart. Die folgenden Systemaufrufe sind implementiert:

Common-System-Timer initialisieren:

```
Status = CommonSysTimerInit (Period);
```

Common-System-Timer anhalten:

```
Status = CommonSysTimerDisable ();
```

Common-System-Timer freigeben:

```
Status = CommonSysTimerEnable ();
```

Zeitintervall des Common-System-Timers setzen:

```
Status = CommonSysTimerSetPeriod (Period);
```

Zeitintervall des Common-System-Timers ermitteln:

```
Period = CommonSysTimerGetPeriod ();
```

Zeitstempel vom Common-System-Timer holen:

```
CommonSysTimerGetTimeStamp (pTimeStamp);
```

System-Timer initialisieren:

```
Status = SysTimerInit (Period);
```

System-Timer anhalten:

```
Status = SysTimerDisable ();
```

System-Timer freigeben:

```
Status = SysTimerEnable ();
```

Zeitintervall des System-Timers setzen:

```
Status = SysTimerSetPeriod (Period);
```

Zeitintervall des System-Timers ermitteln:

```
Period = SysTimerGetPeriod ();
```

Zeitstempel vom System-Timer holen:

```
SysTimerGetTimeStamp (pTimeStamp);
```

Auxiliary-Timer initialisieren:

```
Status = AuxTimerInit (Period);
```

Auxiliary-Timer anhalten:

```
Status = AuxTimerDisable ();
```

Auxiliary-Timer freigeben:

```
Status = AuxTimerEnable ();
```

Zeitintervall des Auxiliary-Timers setzen:

```
Status = AuxTimerSetPeriod (Period);
```

Zeitintervall des Auxiliary-Timers ermitteln:

```
Period = AuxTimerGetPeriod ();
```

2.8 Watchdog-Dienste

Unter einem Watchdog ist ein EUROS-Objekt zu verstehen, mit dem der Anwender Überwachungszeiten einrichten und handhaben kann. Ein Watchdog wird mit einem Zeitintervall versorgt und führt - getrieben durch einen Timer - einen eigenen Zähler. Die Watchdog können entweder vom Common-System-Timer oder vom Auxiliary-Timer versorgt werden. Eine Beschreibung der EUROS-Timer befindet sich im Kapitel ‘Timer-Dienste’ dieses Handbuchs.

Die Watchdog-Dienste stellen eine Art Ergänzung der Interrupt-Dienste dar. Gemeinsam mit diesen lassen sich auf einfache Art und Weise interruptgesteuerte Vorgänge auf Task-Ebene handhaben.

Die Funktionen sind in der Header-Datei `wdg.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Watchdog-Objekte initialisieren:

```
Status = WatchdogInit (ObjMode, Count, TimeLimit,
                       NumberOfWatchdogs);
```

Watchdog erzeugen:

```
WatchdogId = WatchdogCreate (pPath, ObjMode, Count, TimeLimit);
```

Watchdog löschen:

```
Status = WatchdogDelete (WatchdogId, ExecMode);
```

Watchdog zurücksetzen:

```
Status = WatchdogReset (WatchdogId, ExecMode);
```

Watchdog starten:

```
Status = WatchdogStart (WatchdogId, ExecMode);
```

Parameter eines Watchdogs setzen:

```
Status = WatchdogSetParam (WatchdogId, ExecMode, Count, TimeLimit);
```

Timeout-Handler mit einem Watchdog verbinden:

```
AttachId = WatchdogAttach (WatchdogId, ExecMode, pFunc, pArgs);
```

Timeout-Handler von einem Watchdog abkoppeln:

```
Status = WatchdogDetach (AttachId);
```

Status eines Watchdogs abfragen:

```
Status = WatchdogStatus (WatchdogId, pStatus);
```

2.9 Dienste für die Objektverwaltung

Eines der Grundprinzipien des Betriebssystems EUROS ist die dynamische Erzeugung aller Objekte (Tasks, Treiber, Mailboxes, Eventflags, usw.). Jedes Systemobjekt erhält bei der Erzeugung eine ID zuge-

wiesen, mit deren Hilfe EUROS bei allen bearbeitenden Systemaufrufen das Objekt identifiziert. Mit den Diensten für die Objektverwaltung kann ein Objekt anhand seines Pfadnamens identifiziert werden und mit der ID des Objekts die Objekt-Hierarchie durchlaufen werden. Die Funktionen sind in der Header-Datei `mk_prot.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

ID eines Objekts aus seinem Namen ermitteln:
```
Id = ObjectIdent (const char *pPath);
```
Name eines Objekts aus seiner ID ermitteln:
```
Length = ObjectName (int Id, char* pBuffer);
```
Wurzel-ID eines Objekts ermitteln:
```
RootId = ObjectRoot (int Id);
```
ID des ersten untergeordneten Objekts ermitteln:
```
ChildId = ObjectDown (int Id);
```
ID des übergeordneten Objekts ermitteln:
```
ParentId = ObjectUp (int Id);
```
ID des nächsten Objekts derselben Ebene ermitteln:
```
NextId = ObjectNext (int Id);
```
ID des vorangehenden Objekts derselben Ebene ermitteln:
```
PrevId = ObjectPrev (int Id);
```
ID einer Task aus ihrer Task-Struktur ermitteln:
```
Status = GetIdFromTaskPtr (pTask);
```

2.10 Hook-Dienste

Die Hook-Dienste gestatten es dem Anwender, eigene Funktionen bei bestimmten Systemdiensten oder bei bestimmten Ereignissen durch das System aufrufen zu lassen. Da die Ausführung von Hook-Routinen das Laufzeitverhalten von EUROS beeinflußt, muß der Einsatz der Hook-Dienste sorgfältig geplant werden. Insbesondere sollte eine Hook-Routine von kurzer Laufzeit sein. In einer Hook-Routine dürfen keine Systemdienste aufgerufen werden. Der Kontext, in dem die Hook-Routine abläuft, darf nicht verändert werden. Außerdem ist die Anzahl der möglichen Hook-Routinen pro Systemdienst bzw. pro Ereignis auf eine feste Anzahl begrenzt (derzeit 8).

Die Funktionen sind in den Header-Dateien `hooks.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Eine Routine anmelden, die bei jedem `TaskCreate`-Aufruf aufgerufen wird:
```
Status = HookAddTaskCreate (pFunc);
```
Bereits angemeldete `TaskCreate`-Routine entfernen:
```
Status = HookRemoveTaskCreate (pFunc);
```
Eine Routine anmelden, die beim Verdrängen von Tasks aufgerufen wird:
```
Status = HookAddTaskSwitch (pFunc);
```
Bereits angemeldete Routine beim Verdrängen von Tasks entfernen:
```
Status = HookRemoveTaskSwitch (pFunc);
```
Eine Routine anmelden, die beim beim `TaskDelete`-Aufruf aufgerufen wird:
```
Status = HookAddTaskDelete (pFunc);
```
Bereits angemeldete `TaskDelete`-Routine entfernen:
```
Status = HookRemoveTaskDelete (pFunc);
```
Eine Routine anmelden, die beim Löschen der aufrufenden Task aufgerufen wird:
```
Status = HookAddPrivateTaskDelete (pFunc);
```

Bereits angemeldete Routine beim Löschen der aufrufenden Task entfernen:

```
Status = HookRemovePrivateTaskDelete (pFunc);
```

3 Systemdienste des I/O-Systems

Die EUROS-Komponente I/O-System (Ein-/Ausgabesystem) ist die Schnittstelle, über die Tasks spezifische Anforderungen an Port-Treiber bzw. Resource-Manager stellen. Außerdem ist es den Tasks mit den Diensten des I/O-Systems möglich, Port-Treiber bzw. Resource-Manager zu installieren und Ein-/Ausgabe-Anforderungen auszugeben. Das I/O-System ist eine unabhängige Komponente, die in einer EUROS-Konfiguration nicht zwingend vorhanden sein muß. Das I/O-System hat keinerlei Bezüge zum Prozeß-Manager und kann deswegen auch ohne den Prozeß-Manager in die Konfiguration miteingebunden werden.

3.1 Initialisierungs- und Statusfunktionen

Die Komponente I/O-System stellt eine Reihe von Initialisierungs- und Statusfunktionen zur Verfügung, die das I/O-System vorbereiten und dem Anwender bestimmte Statusinformationen liefern.

Die Funktionen sind in der Header-Datei `io.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Initialisieren des I/O-Systems und seiner Wurzelobjekte:
```
Status = IoSysInit (NumObjects);
```
Statusinformationen des I/O-Systems abfragen:
```
Status = IoSysStatus (pStatus);
```

3.2 Allgemeine Ein-/Ausgabe-Dienste

Die allgemeinen Ein-/Ausgabe-Dienste des I/O-Systems umfassen Funktionen zur Handhabung von synchronen/asynchronen Lese- und Schreib-Operationen. Die Funktionen sind in der Header-Datei `io.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

I/O-Objekt erzeugen:
```
IoId = IoCreate (pPath, ObjMode, pIoConfig)
```
I/O-Objekt löschen:
```
Status = IoDelete (IoId, ExecMode)
```
Aufträge eines I/O-Objekts abbrechen und dessen Puffer löschen:
```
Status = IoFlush (IoId, ExecMode)
```
Auftrag eines I/O-Objekts abbrechen:
```
Status = IoCancel (IoId, IoRequestId)
```
I/O-Objekt öffnen:
```
IoId = IoOpen (pPath, ObjMode);
```
I/O-Objekt schließen:
```
Status = IoClose (IoId);
```
I/O-Objekt reservieren:
```
Status = IoReserve (IoId, pArgs);
```
I/O-Objekt freigeben:
```
Status = IoRelease (IoId);
```
Verbindung zur Hardware eines I/O-Objekts überprüfen:
```
Status = IoProbe (IoId, ExecMode);
```
Prüfen, ob im Puffer eines I/O-Objekts bereits Daten vorliegen:
```
Status = IoPeek (IoId, pBuffer, Maximum);
```
Puffer eines I/O-Objekts mit Daten füllen:
```
Status = IoPoke (IoId, pBuffer, Maximum);
```

Schreib-/Lesezeiger eines I/O-Objekts positionieren:

```
Status = IoLSeek (IoId, ExecMode, TimeLimit, pIoLSeek);
```

Daten von einem I/O-Objekt lesen:

```
Status = IoRead (IoId, ExecMode, TimeLimit, pIoRead);
```

Daten an ein I/O-Objekt schreiben:

```
Status = IoWrite (IoId, ExecMode, TimeLimit, pIoWrite);
```

Blockorientiertes I/O-Objekts formatieren:

```
Status = IoFormat (IoId, ExecMode, TimeLimit, pIoFormat);
```

Einstellungen eines I/O-Objekts inspizieren bzw. ändern:

```
Status = IoControl (IoId, ExecMode, TimeLimit, pIoControl);
```

Zwei kombinierbare I/O-Objekte miteinander verbinden:

```
Status = IoAttach (IoId, ExecMode, IoLinkId);
```

Zwei verbundene I/O-Objekte voneinander trennen:

```
Status = IoDetach (IoId, ExecMode);
```

Versionsnummer eines I/O-Objekts abfragen:

```
Version = IoVersion (IoId);
```

Statusinformationen eines I/O-Objekts abfragen:

```
Status = IoStatus (IoId, pStatus)
```

Standard-Kanal einer Task setzen:

```
Status = IoSetTaskStdIo (TaskId, StdId, IoId);
```

Standard-Kanal einer Task ermitteln:

```
Status = IoGetTaskStdIo (TaskId, StdId);
```

Globaler Standard-Kanal setzen:

```
StdId = IoSetGlobalStdIo (StdId, IoId);
```

Globaler Standard-Kanal ermitteln:

```
StdId = IoGetGlobalStdIo (StdId);
```

4 Systemdienste des Prozeß-Managers

Die EUROS-Komponente Prozeß-Manager enthält die Systemobjekte, die in einer Echtzeitumgebung zur Realisierung der eigentlichen Anwenderaufgabe verwendet werden: Tasks, Mailboxes, Ereignisflags, Semaphore, Speicherverwaltung, usw.

4.1 Initialisierungs- und Statusfunktionen

Die Komponente Prozeß-Manager stellt eine Reihe von Initialisierungs- und Statusfunktionen zur Verfügung, die den Prozeß-Manager vorbereiten und dem Anwender bestimmte Statusinformationen liefern. Die Funktionen sind in der Header-Datei `pm_prot.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Prozeß-Manager initialisieren:
```
Status = ProcessManagerInit ();
```
Status des Prozeß-Managers abfragen:
```
Status = ProcessManagerStatus (pStatus);
```
Status des Prozeß-Managers ausgeben:
```
DumpProcessManagerStatus ();
```

4.2 Task-Dienste

Die Tasks stellen die eigentlichen Anwenderprogramme dar. Eine Task ist ein Programm mit eigenem Code- und Datenbereich, das über Systemaufrufe Dienste vom Betriebssystem anfordern kann. Aus Sicht der Task scheint ihr der Prozessor allein zu gehören; der Scheduler von EUROS sorgt dafür, daß derjenigen Task mit der jeweils höchsten Priorität Rechenzeit zugewiesen wird.

Die Funktionen sind in der Header-Dateie `task.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Task erzeugen:
```
TaskId = TaskCreate (pPath, ObjMode, pTdp);
```
Task erzeugen und sofort starten:
```
Status = TaskSpawn (pPath, ObjMode, ExecMode, TimeLimit, CpuId,
                    pFunc, StackSize, Prio, Options, Arg, pArg);
```
Task löschen:
```
Status = TaskDelete (TaskId, ExecMode);
```
Alle Restart-Aufträge einer Task abbrechen:
```
Status = TaskFlush (TaskId, ExecMode);
```
Task starten:
```
Status = TaskStart (TaskId, ExecMode, TimeLimit, Prio, Arg, pArg);
```
Task beenden und nach einem Zeitintervall erneut starten:
```
Status = TaskEndRestart (TimeLimit);
```
Task-Ausführung beenden:
```
Status = TaskTerminate ();
```
Task-Verdrängung sperren:
```
Status = TaskPreemptDisable ();
```
Task-Verdrängung zulassen:
```
Status = TaskPreemptEnable ();
```

CPU-Steuerung abgeben:
```
Status = TaskReschedule ();
```
Task suspendieren:
```
Status = TaskSuspend (TaskId, ExecMode);
```
Suspendierte Task fortsetzen:
```
Status = TaskResume (TaskId, ExecMode);
```
Task für ein Zeitintervall pausieren lassen:
```
Status = TaskSleep (TimeLimit);
```
Pause einer Task aufheben:
```
Status = TaskWakeup (TaskId, ExecMode);
```
Parameter einer Task setzen:
```
Status = TaskSetParam (TaskId, ExecMode, Arg, pArg);
```
Priorität einer Task setzen:
```
Status = TaskSetPrio (TaskId, ExecMode, Prio);
```
Priorität einer Task nach einem Zeitintervall erhöhen:
```
Status = TaskSetOverridePrio (TaskId, ExecMode, TimeLimit,
                              PrioInc, PrioMax);
```
Round-Robin-Intervall einer Task setzen:
```
Status = TaskSetRoundRobin (TaskId, ExecMode, Ticks);
```
Eigene Task-ID ermitteln:
```
TaskId = TaskGetId ();
```
Priorität einer Task ermitteln:
```
Prio = TaskGetPrio (TaskId);
```
Adresse der eigenen Task-Definitionsparameter ermitteln:
```
pTdp = TaskGetTdp ();
```
Status einer Task abfragen:
```
Status = TaskStatus (TaskId, pStatus);
```

4.3 Private-Eventflag-Dienste

Bei der Einrichtung einer Task stellt EUROS ihr automatisch private Eventflags (’PEF’) zur Verfügung. Die Verwaltungsdatenstruktur ist innerhalb der Task-Datenstruktur angeordnet und beansprucht daher keine zusätzlichen System-Resourcen. Die privaten Ereignisflags werden gemeinsam mit der Task erzeugt und auch mit dieser wieder gelöscht.

Die privaten Eventflags können für Task-interne Koordinierungsaufgaben eingesetzt werden. Eine nützliche Anwendung liegt z.B. darin, eine Ein-/Ausgabe-Anforderung an einen Treiber ohne synchrones Warten - aber mit Setzen eines Eventflags - auszuführen (Koordinierungsverfahren `ExecMode` = `C_SEF` bzw. `C_REF`). Die Task kann während der Ein-/Ausgabe-Operation ihre Abarbeitung weiter fortsetzen und am Zustand des betreffenden privaten Eventflags das Ende der Operation feststellen.

Die Funktionen sind in der Header-Datei `pef.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Private Eventflags setzen/rücksetzen:
```
Status = PrivFlagSet (Mask, Value);
```
Private Eventflags nach einem Zeitintervall setzen/rücksetzen:
```
TimeId = PrivFlagSetTime (ExecMode, TimeLimit, Mask, Value);
```

Verzögertes Setzen/Rücksetzen privater Eventflags aufheben:
```
Status = PrivFlagUntime (TimeId);
```
Auf das Setzen/Rücksetzen privater Eventflags warten:
```
WaitId = PrivFlagWait (ExecMode, TimeLimit, Mask, Value);
```
Warten auf private Eventflags abbrechen:
```
Status = PrivFlagUnwait (WaitId);
```
Private Eventflags zurücksetzen:
```
Status = PrivFlagFlush ();
```
Status der eigenen privaten Eventflags abfragen:
```
Status = PrivFlagStatus (pStatus);
```

4.4 Private-Signal-Dienste

Bei der Einrichtung einer Task stellt EUROS ihr automatisch private Signale (’PSG’, private signals) zur Verfügung. Die Verwaltungsdatenstruktur ist innerhalb der Task-Datenstruktur angeordnet und beansprucht daher keine zusätzlichen System-Resourcen. Die privaten Signale werden gemeinsam mit der Task erzeugt und auch mit dieser wieder gelöscht. Private Signale können für Task-interne Koordinierungsaufgaben eingesetzt werden (Koordinationsmodus `ExecMode` = `C_PSG`). Eine nützliche Anwendung der privaten Signale liegt z.B. in der Koordinierung von Systemaufrufen (ähnlich den privaten Eventflags).

Die Funktionen sind in der Header-Datei `sig.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Privates Signal mit einer Routine verbinden:
```
AttachId = PrivSignalAttach (ExecMode, Mask, pFunc, pArg);
```
Routine von einem privaten Signal abkoppeln:
```
Status = PrivSignalDetach (AttachId);
```
Privates Signal senden:
```
Status = PrivSignalSend (Mask);
```
Privates Signal nach einem Zeitintervall senden:
```
TimeId = PrivSignalSendTime (TimeLimit, Mask);
```
Verzögertes Senden eines privaten Signals aufheben:
```
Status = PrivSignalUntime (TimeId);
```
Privates Signal empfangen:
```
ReceiveId = PrivSignalReceive (ExecMode, TimeLimit, Mask);
```
Auftrag zum Empfang von privaten Signals aufheben:
```
Status = PrivSignalUnreceive (ReceiveId);
```
Alle Aufträge einer privaten Signal-Gruppe abbrechen:
```
Status = PrivSignalFlush ();
```
Status der eigenen privaten Signals abfragen:
```
Status = PrivSignalStatus (pStatus);
```

4.5 System-Signal-Dienste

Die System-Signale weisen Ähnlichkeit zu den privaten Signalen auf. Wie diese sind sie private Objekte der Tasks und werden automatisch bei der Erzeugung der Task angelegt. Die Verwaltungsdatenstruktur ist innerhalb der Task-Datenstruktur angeordnet und beansprucht daher keine zusätzlichen System-Resourcen. Die System-Signale werden gemeinsam mit der Task erzeugt und auch mit dieser wieder gelöscht.

Sie dienen zur Behandlung von System-Ablaufbesonderheiten (Exceptions); jede Task hat die Möglichkeit, ihre eigenen Exception-Handler zu installieren. Jedem Bit der System-Signale ist eine bestimmte Ex-

ception zugeordnet (Division durch Null, unzulässiger Opcode, usw.), und die Task kann mit jedem dieser Signal-Bits eine Behandlungsroutine verbinden - z.B. zur Ausgabe einer Warnmeldung, sich selbst beenden oder sonstige Aktivitäten zur gezielten Fehlerbehandlung.

Die Funktionen sind in der Header-Datei `sig.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

System-Signal mit einer Routine verbinden:
```
AttachId = SysSignalAttach (ExecMode, Mask, pFunc, pArg);
```
Routine von einem System-Signal abkoppeln:
```
Status = SysSignalDetach (AttachId);
```
System-Signal senden:
```
Status = SysSignalSend (Mask);
```
System-Signal nach einem Zeitintervall senden:
```
TimeId = SysSignalSendTime (TimeLimit, Mask);
```
Verzögertes Senden eines System-Signals aufheben:
```
Status = SysSignalUntime (TimeId);
```
System-Signal empfangen:
```
ReceiveId = SysSignalReceive (ExecMode, TimeLimit, Mask);
```
Auftrag zum Empfang von System-Signals aufheben:
```
Status = SysSignalUnreceive (ReceiveId);
```
Alle Aufträge einer System-Signal-Gruppe abbrechen:
```
Status = SysSignalFlush ();
```
Status der eigenen System-Signal-Gruppe abfragen:
```
Status = SysSignalStatus (pStatus);
```

4.6 Mega-Pool-Dienste

Ein Mega-Pool ist ein zusammenhängender Speicherbereich mit einer beliebigen Länge, jedoch maximal 4 GByte. Der Anwender kann Speicher beliebiger Länge anfordern und zurückgeben, jedoch maximal 4 GByte. EUROS verwaltet die aktuelle Belegung dieses Speichers mit einer Bit-Tabelle (Bitmap). Jedes Bit repräsentiert dabei einen Block fester Länge.

Eine Speicheranforderung beliebiger Länge reserviert immer einen Bereich, der auf ganze Blocklängen aufgerundet ist. Es ergibt sich ein ‘Verschnitt’, der für die Zeit der Belegung nicht nutzbar ist. Bei Applikationen, die in größerem Umfang mit kleinen Speichereinheiten arbeiten, bieten sich als Alternative die Buffer-Pool- oder Memory-Pool-Dienste an.

Die Funktionen sind in der Header-Datei `meg.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Mega-Pool-Objekte initialisieren:
```
Status = MegPoolInit (ObjMode, pBuffer, Length, BlockSize);
```
Mega-Pool erzeugen:
```
MegId = MegPoolCreate (pPath, ObjMode, pBuffer, Length, BlockSize);
```
Mega-Pool löschen:
```
Status = MegPoolDelete (MegId, ExecMode);
```
Alle Aufträge eines Mega-Pools abbrechen:
```
Status = MegPoolFlush (MegId, ExecMode);
```

Speicherplatz von einem Mega-Pool anfordern:
```
GetId = MegPoolGet (MegId, ExecMode, TimeLimit, Length,
                    ppBlock, pMegId);
```
Speicheranforderung an einen Mega-Pool abbrechen:
```
Status = MegPoolUnget (GetId);
```
Speicherbereich in einem Mega-Pool reservieren:
```
Status = MegPoolReserve (MegId, Length, pBuffer);
```
Größe eines bereits angeforderten Speicherblocks ändern:
```
Status = MegPoolRealloc (MegId, ExecMode, pOldBlock, OldSize,
                         ppNewBlock, NewSize);
```
Speicherplatz an einen Mega-Pool zurückgeben:
```
Status = MegPoolPut (MegId, Length, ppBlock);
```
Status eines Mega-Pools abfragen:
```
Status = MegPoolStatus (MegId, pStatus);
```

4.7 Memory-Pool-Dienste

Ein Memory-Pool ist ein zusammenhängender Speicherbereich, der bei Architekturen mit segmentierter Adressierung, die Seitenadressen beim Adressieren verwenden, eine architekturabhängige maximale Länge hat (z.B. 64 KByte in der Intel-16-Bit-Version). Benötigt eine Task für eine begrenzte Zeit einen statischen Speicherbereich, so braucht dieser Bereich nicht fest angelegt zu werden - er würde zur Laufzeit ständig Adreßraum belegen, der anderen Tasks verlorengeht. Die Task holt sich stattdessen den gewünschten Speicher aus einem Pool, den sie mit den anderen Tasks teilt: Die Folge ist eine effiziente Nutzung des vorhandenen Speichers.

Die Funktionen sind in der Header-Datei `mem.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Memory-Pool-Objekte initialisieren:
```
Status = MemPoolInit (ObjMode, pBuffer, Length);
```
Memory-Pool erzeugen:
```
MemId = MemPoolCreate (pPath, ObjMode, pBuffer, Length);
```
Memory-Pool löschen:
```
Status = MemPoolDelete (MemId, ExecMode);
```
Alle Aufträge eines Memory-Pools abbrechen:
```
Status = MemPoolFlush (MemId, ExecMode);
```
Speicherplatz von einem Memory-Pool anfordern:
```
GetId = MemPoolGet (MemId, ExecMode, TimeLimit, Length, ppBuffer,
                    pMemId);
```
Speicheranforderung an einen Memory-Pool abbrechen:
```
Status = MemPoolUnget (GetId);
```
Speicherplatz an einen Memory-Pool zurückgeben:
```
Status = MemPoolPut (MemId, ppBuffer);
```
Status eines Memory-Pools abfragen:
```
Status = MemPoolStatus (MemId, pStatus);
```

4.8 Buffer-Pool-Dienste

Unter dem Begriff Buffer-Pool (auch ‘Fixed-Size-Buffer’ genannt) ist ein Speicherbereich zu verstehen, der ähnlich den Memory-Pools oder Mega-Pools vom Anwender für vorübergehend benötigte Datenberei-

che genutzt werden kann. Buffer-Pools sind - im Gegensatz zu den beiden anderen Typen der Speicherverwaltung - in gleichgroße Speicherblöcke eingeteilt, deren Länge der Anwender bei der Erzeugung vorgibt. Eine Speicheranforderung holt jeweils nur einen einzelnen Block. Der Vorteil liegt in der vereinfachten Verwaltungsarbeit durch EUROS und einer entsprechend kürzeren Ausführungszeit der zugehörigen Systemaufrufe.

Die Funktionen sind in der Header-Datei `buf.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Buffer-Pool-Objekte initialisieren:
```
Status = BufPoolInit (ObjMode, pBuffer, BufLength, BlockSize);
```
Buffer-Pool erzeugen:
```
BufId = BufPoolCreate (pPath, ObjMode, pBuffer, BufLength,
                       BlockSize);
```
Buffer-Pool löschen:
```
Status = BufPoolDelete (BufId, ExecMode);
```
Alle Aufträge eines Buffer-Pools abbrechen:
```
Status = BufPoolFlush (BufId, ExecMode);
```
Speicherblock von einem Buffer-Pool anfordern:
```
GetId = BufPoolGet (BufId, ExecMode, TimeLimit, ppBlock, pBufId);
```
Speicheranforderung an einen Buffer-Pool abbrechen:
```
Status = BufPoolUnget (GetId);
```
Speicherblock an einen Buffer-Pool zurückgeben:
```
Status = BufPoolPut (BufId, ppBlock);
```
Status eines Buffer-Pools abfragen:
```
Status = BufPoolStatus (BufId, pStatus);
```

4.9 Shared-Memory-Dienste

Eine spezielle Form der Speicherverwaltung stellen die Shared-Memory-Dienste dar. Ein Shared-Memory-Objekt ist ein Speicherbereich fester Länge, über den verschiedene Tasks miteinander kommunizieren können; die Zugriffe werden dabei über einen integrierten Semaphor-Mechanismus koordiniert.

Die Funktionen sind in der Header-Datei `shm.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Shared-Memory-Objekte initialisieren:
```
Status = SharedMemInit (ObjMode, pBuffer, Size);
```
Shared-Memory-Bereich erzeugen:
```
ShmId = SharedMemCreate (pPath, ObjMode, pBuffer, Size);
```
Shared-Memory-Bereich löschen:
```
Status = SharedMemDelete (ShmId, ExecMode);
```
Alle Aufträge eines Shared-Memory-Bereich abbrechen:
```
Status = SharedMemFlush (ShmId, ExecMode);
```
Shared-Memory-Bereich anfordern:
```
GetId = SharedMemGet (ShmId, ExecMode, TimeLimit, ppBuffer,
                      pShmId);
```
Speicheranforderung an einen Shared-Memory-Bereich abbrechen:
```
Status = SharedMemUnget (GetId);
```

Shared-Memory-Bereich zurückgeben:

```
Status = SharedMemPut (ShmId, ppBuffer);
```

Status eines Shared-Memory-Bereichs abfragen:

```
Status = SharedMemStatus (ShmId, pStatus);
```

4.10 Eventflag-Dienste

Ereignisflag-Objekte sind Gruppen von jeweils 16 Bit, die dem Anwender für vielfältige Koordinierungs- und Synchronisierungszwecke zur Verfügung stehen. EUROS unterscheidet zwischen den Ereignisflag-Objekten ('CEF', CommonEventflags oder auch Eventflags) und den privaten Ereignisflaggruppen ('PEF', private Eventflags). Die Ereignisflags-Objekte sind dynamische Systemobjekte des Prozeß-Managers, die zur Laufzeit eingerichtet werden; die privaten Ereignisflags sind fest an eine Task gebunden und können auch nur von dieser benutzt werden.

Die Funktionen sind in der Header-Datei `cef.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Eventflag-Objekte initialisieren:

```
Status = EventFlagInit (ObjMode);
```

Eventflag-Gruppe erzeugen:

```
EventId = EventFlagCreate (pPath, ObjMode);
```

Eventflag-Gruppe löschen:

```
Status = EventFlagDelete (EventId, ExecMode);
```

Alle Aufträge einer Eventflag-Gruppe abbrechen:

```
Status = EventFlagFlush (EventId, ExecMode);
```

Eventflags setzen/rücksetzen:

```
Status = EventFlagSet (EventId, ExecMode, Mask, Value);
```

Eventflags nach einem Zeitintervall setzen/rücksetzen:

```
TimeId = EventFlagSetTime (EventId, ExecMode, TimeLimit, Mask,
                           Value);
```

Verzögertes Setzen/Rücksetzen von Eventflags aufheben:

```
Status =  EventFlagUntime (TimeId);
```

Auf das Setzen/Rücksetzen von Eventflags warten:

```
WaitId = EventFlagWait (EventId, ExecMode, TimeLimit, Mask, Value);
```

Warten auf Eventflags abbrechen:

```
Status = EventFlagUnwait (WaitId);
```

Status einer Eventflag-Gruppe abfragen:

```
Status = EventFlagStatus (EventId, pStatus);
```

4.11 Signal-Objekt-Dienste

Signal-Objekte sind Gruppen von jeweils 16 Bit, die der Anwender zur schnellen Benachrichtigung von Tasks und Treibern bei bestimmten Ereignissen verwenden kann. Der Empfänger kann einzelne Bits mit einer eigenen Behandlungsroutine verknüpfen; wenn der Sender ein Signal absendet, wird die zugehörige Routine des Empfängers aktiviert.

EUROS unterscheidet zwischen den Signal-Objekten ('CSG', Common-Signal-Groups oder auch Signal-Objekte) und den privaten Signalen ('PSG', Private Signals). Die Signal-Objekte sind dynamische Systemobjekte im EUROS-Sinne, die zur Laufzeit eingerichtet werden; die privaten Signale sind fest an eine Task gebunden und können auch nur von dieser benutzt werden.

Die Funktionen sind in der Header-Datei `sig.h` deklariert. Die folgenden Systemaufrufe sind implemen-

tiert:

Wurzelobjekt der Signal-Objekte initialisieren:
```
SigId = SignalInit (ObjMode);
```
Signal-Gruppe erzeugen:
```
SigId = SignalCreate (pPath, ObjMode);
```
Signal-Gruppe löschen:
```
Status = SignalDelete (SigId, ExecMode);
```
Alle Aufträge einer Signal-Gruppe abbrechen:
```
Status = SignalFlush (SigId, ExecMode);
```
Signal mit einer Routine verbinden:
```
AttachId = SignalAttach (SigId, ExecMode, Mask, pFunc, pArg);
```
Routine von einem Signal abkoppeln:
```
Status = SignalDetach (AttachId);
```
Signal senden:
```
Status = SignalSend (SigId, ExecMode, Mask);
```
Signal nach einem Zeitintervall senden:
```
TimeId = SignalSendTime (SigId, ExecMode, TimeLimit, Mask);
```
Verzögertes Senden eines Signals aufheben:
```
Status = SignalUntime (TimeId);
```
Signal empfangen:
```
ReceiveId = SignalReceive (SigId, ExecMode, TimeLimit, Mask);
```
Auftrag zum Empfang von Signals aufheben:
```
Status = SignalUnreceive (ReceiveId);
```
Status einer Signal-Gruppe abfragen:
```
Status = SignalStatus (SigId, pStatus);
```

4.12 Mailbox-Dienste

Eine Mailbox ist ein Systemobjekt, das als universelles Kommunikationsmedium für Tasks, Resource-Manager, Port-Treiber und Interrupt-Handler genutzt werden kann. Eine Mailbox nimmt - nach Prioritäten geordnet - Botschaften (Nachrichten, Messages) eines sendenden Programms auf, die von einem Empfänger abgeholt werden; hierbei stehen eine Reihe von Synchronisations- und Koordinierungsmechanismen zur Verfügung.

Die Funktionen sind in der Header-Datei `mailbox.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Mailbox-Objekte initialisieren:
```
Status = MailboxInit (ObjMode, MaxMessages, MaxTasks, MessageSize);
```
Mailbox erzeugen:
```
MailboxId = MailboxCreate (pPath, ObjMode, MaxMessages, MaxTasks,
                           MessageSize);
```
Mailbox löschen:
```
Status = MailboxDelete (MailboxId, ExecMode);
```
Alle Aufträge einer Mailbox abbrechen:
```
Status = MailboxFlush (MailboxId, ExecMode);
```

Nachricht an eine Mailbox senden:
```
SendId = MailboxSend (MailboxId, ExecMode, TimeLimit, Prio,
                      pMessage);
```
Sendeauftrag an eine Mailbox abbrechen:
```
Status = MailboxUnsend (SendId);
```
Nachricht an eine Mailbox nach einem Zeitintervall senden:
```
TimeId = MailboxSendTime (MailboxId, ExecMode, TimeLimit, Prio,
                          pMessage);
```
Auftrag zum verzögeten Senden aufheben:
```
Status = MailboxUntime (TimeId);
```
Broadcast-Nachricht an eine Mailbox senden:
```
Status = MailboxBroadcast (MailboxId, ExecMode, pMessage);
```
Nachricht von einer Mailbox empfangen:
```
ReceiveId = MailboxReceive (MailboxId, ExecMode, TimeLimit, Prio,
                            pMessage);
```
Empfangsauftrag von einer Mailbox abbrechen:
```
Status = MailboxUnreceive (ReceiveId);
```
Status einer Mailbox abfragen:
```
Status = MailboxStatus (MailboxId, pStatusBuf);
```

4.13 Pipeline-Dienste

Eine EUROS-Pipe ist ein Ringpuffer, in den Tasks Daten schreiben können und aus dem Tasks Daten lesen können. Wenn der Pipe-Puffer leer ist, können lesende Tasks warten. Wenn der Pipe-Puffer voll ist, können schreibende Tasks warten. Bei nicht völlig gefülltem Puffer sind beide Warteschlangen leer.

Ein Schreibauftrag (Senden) blockiert immer so lange, bis alle Daten geschrieben werden konnten. Ein Leseauftrag (Empfangen) blockiert nur so lange, bis wenigstens ein Byte gelesen wurde. Beide Aufrufe liefern die Anzahl der tatsächlich übertragenen Bytes.

Die Funktionen sind in der Header-Datei `pipe.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Pipe-Objekte initialisieren:
```
PipeId = PipeInit (ObjMode, BufSize);
```
Pipe erzeugen:
```
PipeId = PipeCreate (pPath, ObjMode, BufSize);
```
Pipe löschen:
```
Status = PipeDelete (PipeId, ExecMode);
```
Alle Aufträge einer Pipe abbrechen:
```
Status = PipeFlush (PipeId, ExecMode);
```
Daten an eine Pipe senden:
```
SendId = PipeSend (PipeId, ExecMode, TimeLimit, pData, pDataLen);
```
Sendeauftrag an eine Pipe abbrechen:
```
Status = PipeUnsend (SendId);
```
Daten von einer Pipe empfangen:
```
ReceiveId = PipeReceive (PipeId, ExecMode, TimeLimit, pBuffer,
                         pBufferLen);
```

Empfangsauftrag von einer Pipe abbrechen:

```
Status = PipeUnreceive (ReceiveId);
```

Status einer Pipe abfragen:

```
Status = PipeStatus (PipeId, pStatus);
```

4.14 Semaphor-Dienste

Ein Semaphor ist ein Systemobjekt, daß zur Koordinierung der Zugriffe auf gemeinsam von verschiedenen Tasks benutzte Betriebsmittel dient. Da in einer Multitasking-Umgebung ein Taskwechsel prinzipiell zu jedem Zeitpunkt erfolgen kann, muß der Anwender dafür Sorge tragen, daß es an gemeinsam verwendeten Betriebsmitteln nicht zu Kollisionen kommt. Dies ist wichtig z.B. bei Prozeduren, die nicht ‘reentrant’ (wiedereintrittsfähig) aufgebaut sind, oder bei Speicherbereichen, die für den Zeitraum der Bearbeitung inkonsistent sein können.

Mit den Semaphore-Diensten können Überschneidungen verhindert und kritische Ablaufphasen geschützt werden. Semaphore sind bei EUROS als Zähler implementiert, deren Grenzwert bei der Erzeugung vorgegeben wird. Der Wert 1 bestimmt ein binäres Semaphor: nur jeweils einer Task ist der Zutritt gestattet. Bei größeren Werten werden entsprechend mehrere Tasks zugelassen, bevor weitere Anforderungen warten müssen. Die ‘Anforderung’ oder ‘Belegung’ eines Semaphors bewirkt ein Erniedrigen (Herunterzählen) des Semaphor-Levels, das ‘Freigeben’ ein Erhöhen (Heraufzählen). Ein Semaphor ist ‘belegt’, wenn der Level den Wert 0 erreicht hat.

Die Funktionen sind in der Header-Datei `sem.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Semaphor-Objekte initialisieren:

```
SemId = SemInit (ObjMode, Count);
```

Semaphor erzeugen:

```
SemId = SemCreate (pPath, ObjMode, Count);
```

Semaphor löschen:

```
Status = SemDelete (SemId, ExecMode);
```

Alle Aufträge eines Semaphores abbrechen:

```
Status = SemFlush (SemId, ExecMode);
```

Semaphor Testen/Reservieren:

```
WaitId = SemWait (SemId, ExecMode, TimeLimit, pId);
```

Warteauftrag zur Belegung eines Semaphors aufheben:

```
Status = SemUnwait (WaitId);
```

Semaphor freigeben:

```
Status = SemSignal (SemId, ExecMode);
```

Status eines Semaphor abfragen:

```
Status = SemStatus (SemId, pStatus);
```

4.15 Datum-/Uhrzeit-Dienste

EUROS führt mit Hilfe des Systemtakt-Interrupts eine Systemuhr, die intern als 64-Bit-Zähler implementiert ist. Dieser Zähler wird in Nanosekunden geführt und für jeden Systemtakt um diejenige Anzahl von Nanosekunden erhöht, die ein Systemtakt dauert. Die Systemzeit kann vom Anwender in vier verschiedenen Formaten genutzt werden.

Die Funktionen sind in der Header-Datei `pm_prot.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Systemzeit setzen:
```
Status = TimeSet (Format, pBuffer);
```
Systemzeit lesen:
```
Status = TimeGet (Format, pBuffer);
```

4.16 Bit-Processing-Dienste

Die Bit-Processing-Dienste erlauben die Handhabung bitorientierter Ein-/ Ausgabe-Peripherie auf logischer Ebene. Eine Gruppe von 8 bzw. 16 Bits bildet ein EUROS-Systemobjekt (‘Diskrete Signalgruppe’), das zur Laufzeit dynamisch eingerichtet wird und über Systemaufrufe auf vielfältige Weise bearbeitet werden kann. Die physikalische Adresse und die Eigenschaften der Bit-Ein-/Ausgabe müssen nur bei der Einrichtung des Systemobjekts bekannt sein; im weiteren Verlauf der Applikation arbeiten die Tasks ausschließlich mit hardwareunabhängigen Systemdiensten - ein Vorteil, der bei Änderungen der Hardwarekonfiguration oder bei der Portierung auf ein anderes System zu einer Reduzierung der Umstellungsarbeiten führt.

Die Bit-Processing-Dienste besitzen eine große Ähnlichkeit zu den Eventflag-Objekten; der wesentliche Unterschied liegt darin, daß sie mit festen Adressen (bzw. Port-Kanälen) verbunden sind. EUROS führt ein internes Abbild der Bitzustände, so daß auch nicht-rücklesbare Ausgaben bzw. nicht-schreibbare Eingaben unterstützt werden.

Ein weiterer Unterschied zu den Eventflags liegt darin, daß Eingänge sich asynchron durch Einfluß von außen ändern; bei Eingaben, die keine Interrupts erzeugen, kann das Betriebssystem Änderungen nur durch zyklisches Abfragen der Eingänge erfahren (siehe hierzu `BitScanTime`).

Die Funktionen sind in der Header-Datei `bit.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Wurzelobjekt der Bit-Gruppen-Objekte initialisieren:
```
Status = BitInit (ObjMode, pAddr, BitOptions);
```
Bit-Gruppe erzeugen:
```
BitId = BitCreate (pPath, ObjMode, pAddr, BitOptions);
```
Bit-Gruppe löschen:
```
Status = BitDelete (BitId, ExecMode);
```
Alle Aufträge einer Bit-Gruppe aufheben:
```
Status = BitFlush (BitId, ExecMode);
```
Bits einer Bit-Gruppe setzen/rücksetzen:
```
Status = BitSet (BitId, ExecMode, Mask, Value);
```
Bits einer Bit-Gruppe nach einem Zeitintervall setzen/rücksetzen:
```
TimeId = BitSetTime (BitId, ExecMode, TimeLimit, Mask, Value);
```
Bits einer Bit-Gruppe zyklisch abtasten:
```
TimeId = BitScanTime (BitId, ExecMode, TimeLimit, Mask);
```
Zeitauftrag an einer Bit-Gruppe aufheben:
```
Status = BitUntime (TimeId);
```
Warten auf das Setzen/Rücksetzen von Bits einer Bit-Gruppe:
```
WaitId = BitWait (BitId, ExecMode, TimeLimit, Mask, Value);
```
Warteauftrag einer Bit-Gruppe aufheben:
```
Status = BitUnwait (WaitId);
```
Status einer Bit-Grupp abfragen:
```
Status = BitStatus (BitId, pStatus);
```

5 Systemdienste des Dateisystems

5.1 Initialisierungs- und Statusfunktionen

Mit den Initialisierungsfunktionen wird die Vorraussetzung für das korrekte Arbeiten der Datei- und Verzeichnisfunktionen der FMS-Komponente geschaffen.

Die Funktionen sind in der Header-Datei `fms.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

```
Status = FmsInit(Files, Descriptors);
```
FMS initialisieren

```
Status = FmsAutoMount(DriverId, FileBuffers);
```
Anmelden aller Partitionen einer Platte

```
Status = FmsMount(pName, StartSector, Flags, DriverId, FileBuffers,
                  FileBufferSize);
```
Anmelden einer Partition

5.2 Dateifunktionen

Diese Funktionen werden für das Arbeiten mit Dateien verwendet. Für ein korrektes Arbeiten der Funktionen muß das FMS initialisiert und mindestens eine Partition angemeldet sein.

```
Status = FmsClose (tDescriptor *pDescriptor);
```
Schließen einer Datei

```
pDirRecord = FmsGetStatus (pDescriptor, pDirRecord );
```
Status einer Datei abfragen

```
Offset = FmsLSeek(pDescriptor, Offset, Whence);
```
Positionieren des Dateizeigers

```
pDescriptor = FmsOpen(pName, Flags);
```
Öffnen einer Datei

```
Number = FmsRead(pDescriptor, pBuffer, MaxBytes);
```
Lesen von einer Datei

```
Status = FmsRemove(pName);
```
Löschen einer Datei

```
Status = FmsRename(pOldName, pNewName);
```
Umbenennen einer Datei

```
FmsRewind(pDescriptor);
```
Zurücksetzen des Dateizeigers

```
Status = FmsSetStatus (pDescriptor, pDirRecord);
```
Status einer Datei setzen

```
Number = FmsWrite(pDescriptor, pBuffer, NBytes);
```
Schreiben in eine Datei

5.3 Verzeichnisfunktionen

Diese Funktionen werden für das Arbeiten mit Verzeichnissen verwendet. Für ein korrektes Arbeiten der Funktionen muß das FMS initialisiert und mindestens eine Partition angemeldet sein.

```
Status = FmsCloseDir(pDescriptor);
```
Schließen eines Verzeichnisses

```
Status = FmsMkDir(pName, Mode);
```
Anlegen eines Verzeichnisses

```
Status = FmsChDir(pPath);
```
Setzen eines Arbeitsverzeichnisses

```
pDescriptor = FmsOpenDir(pName);
```
Öffnen eines Verzeichnisses

```
pName = FmsGetCurDir(void);
```
Ermitteln des aktuellen Arbeitsverzeichnisses

```
pRoots = FmsGetRoots(void);
```
Ermitteln der angemeldeten Partitionen

```
pDirRecord = FmsReadDir(pDescriptor, pDirRecord );
```
Lesen eines Verzeichniseintrages

```
          FmsRewindDir(pDescriptor);
```
Rücksetzen des Verzeichniszeigers

```
Status = FmsRmDir(pName);
```
Löschen eines Verzeichnisses

6 Systemdienste der C-Laufzeit-Bibliothek

Die C-Laufzeit-Bibliothek ist eine EUROS-Komponente und enthält die C-Funktionen gemäß ANSI-Draft-Standard X3.159-1989 bzw. ISO-Standard ISO/IEC 9899:1990. Die C-Laufzeit-Bibliothek ist reentrant, multitaskingfähig und wird nur einmal zur Applikation gebunden. Die C-Laufzeit-Bibliothek von EUROS ersetzt compilerspezifische C-Laufzeit-Bibliotheken.

6.1 Klassifizierung und Umwandlung von Zeichen

Diese Funktionen dienen zur Klassifizierung und Umwandlung von Zeichen nach bestimmten Kriterien. Die Funktionen zur Klassifizierung beginnen mit dem Präfix “is”, die Funktionen zur Umwandlung beginnen mit dem Präfix “to”. All diesen Funktionen kann ein Argument vom Typ `int` übergeben werden, dessen Wert entweder ein für Variablen des Datentyps `unsigned char` zulässiger Wert oder `EOF` sein muß. Wird ein anderer Wert übergeben, so ist das Verhalten der Funktionen undefiniert. Eine Ausnahme sind die Funktionen `isascii` und `toascii`, für diese Funktionen sind auch alle anderen für Variablen des Datentyps `int` zulässigen Werte erlaubt. Die Funktionen sind in der Header-Datei `ctype.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

7-Bit-ASCII-Zeichen klassifizieren: (EUROS)
```
Status = isascii(Character);
```
Steuerzeichen klassifizieren:
```
Status = iscntrl(Character);
```
Wait-Space Zeichen klassifizieren:
```
Status = isspace(Character);
```
Druckbare Zeichen klassifizieren:
```
Status = isprint(Character);
```
Sichtbare Zeichen klassifizieren:
```
Status = isgraph(Character);
```
Satzzeichen klassifizieren:
```
Status = ispunct(Character);
```
Alphanumerische Zeichen klassifizieren:
```
Status = isalnum(Character);
```
Hexadezimalziffer klassifizieren:
```
Status = isxdigit(Character);
```
Ziffer klassifizieren:
```
Status = isdigit(Character);
```
Buchstaben klassifizieren:
```
Status = isalpha(Character);
```
Kleinbuchstaben klassifizieren:
```
Status = islower(Character);
```
Großbuchstaben klassifizieren:
```
Status = isupper(Character);
```
Integer-Werte in 7-Bit-ASCII-Code umwandeln: (EUROS)
```
AsciiCode = toascii(Character);
```
Groß- in Kleinbuchstaben umwandeln:
```
LowerChar = tolower(Character);
```

Klein- in Großbuchstaben umwandeln:
```
UpperChar = toupper(Character);
```

6.2 String-Operationen

Mit String-Operationen lassen sich Zeichenketten überprüfen und bearbeiten. Die Funktionen sind in den Header-Dateien `string.h` und `stdlib.h` vereinbart. Die folgenden Systemaufrufe sind implementiert:

Zeichenkette in eine double-Zahl umwandeln:
```
Result = atof (pString);
```
Zeichenkette in eine int-Zahl umwandeln:
```
Result = atoi (pString);
```
Zeichenkette in eine long-Zahl umwandeln:
```
Result = atol (pString);
```
Zeichenkette in eine double-Zahl umwandeln:
```
Result = strtod (pString, ppRest);
```
Zeichenkette in eine long-Zahl umwandeln:
```
Result = strtol (pString, ppRest, Base);
```
Zeichenkette in eine unsigned-long-Zahl umwandeln:
```
Result = strtoul (pString, ppRest, Base);
```
Länge eines Multibyte-Zeichens ermitteln:
```
Length = mblen (pMbChar, Maximum);
```
Multibyte-Zeichen in ein “langes” Zeichen umwandeln:
```
Number = mbtowc (pWideChar, pMbChar, Maximum);
```
Multibyte-Zeichenkette in eine Folge “langer” Zeichen umwandeln:
```
Number = mbstowcs (pWideChars, pMbChars, Maximum);
```
Ein “langes” Zeichen in ein Multibyte-Zeichen umwandeln:
```
Number = wctomb (pMbChar, WideChar);
```
Eine Folge “langer” Zeichen in eine Multibyte-Zeichenkette umwandeln:
```
Number = wcstombs (pMbChars, pWideChars, Maximum);
```
Zeichenkette an eine anderen Zeichenkette anhängen:
```
pDest = strcat (pDest, pSource);
```
Zeichen in einer Zeichenkette suchen:
```
pCharacter = strchr (pString, Character);
```
Zweier Zeichenketten ohne Berücksichtigung der Spracheinstellung vergleichen:
```
Status = strcmp (pString1, pString2);
```
Zweier Zeichenketten mit Berücksichtigung der Spracheinstellung vergleichen:
```
Status = strcoll (pString1, pString2);
```
Zeichenkette in eine andere Zeichenkette kopieren:
```
pDest = strcpy (pDest, pSource);
```
Die nicht Übereinstimmung zweier Zeichenketten ermitteln:
```
Length = strcspn (pString1, pString2);
```
Länge einer Zeichenkette ermitteln:
```
Length = strlen (pString);
```
Maximale Anzahl Zeichen einer Zeichenkette an eine andere anhängen:
```
pDest = strncat (pDest, pSource, Maximum);
```

Maximale Anzahl Zeichen zweier Zeichenketten vergleichen:
```
Status = strncmp (pDest, pSource, Maximum);
```
Maximale Anzahl Zeichen einer Zeichenkette in eine andere kopieren:
```
pDest = strncpy (pDest, pSource, Maximum);
```
Nach dem ersten Auftreten eines Zeiches in einer Zeichenkette suchen:
```
pCharacter = strpbrk (pString1, pString2);
```
Nach dem letzten Auftreten eines Zeichens in einer Zeichenkette suchen:
```
pCharacter = strrchr (pString, Character);
```
Länge einer Teilzeichenkette ermitteln, die ausschließlich aus Zeichen einer anderen Zeichenkette besteht:
```
Length = strspn (pString1, pString2);
```
Die erste Übereinstimmung zweier Zeichenketten ermitteln:
```
pString = strstr (pString1, pString2);
```
Nach der ersten, durch Trennzeichen begrenzten, Zeichenfolge in einer Zeichenkette suchen:
```
pToken = strtok (pString, pDelimitter);
```
Maximale Anzahl Zeichen einer Zeichenkette in eine andere umwandeln und kopieren:
```
Length = strxfrm (pString1, pString2, Maximum);
```

6.3 Speicher-Operationen

Mit Speicher-Operationen können Zeichen kopiert sowie Speicherbereiche verglichen oder beschrieben werden. Die Funktionen sind in der Header-Datei `string.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Nach einem Zeichen in einem Speicherbereich suchen:
```
pCharacter = memchr (pAddr, Character, Length);
```
Zwei Speicherbereiche vergleichen:
```
Status = memcmp (pAddr1, pAddr2, Length);
```
Zeichen vom Quellbereich in den Zielbereich kopieren:
```
pDest = memcpy (pDest, pSource, Length);
```
Zeichen vom Quellbereich in den Zielbereich kopieren:
```
pDest = memmove (pDest, pSource, Length);
```
Speicherbereich mit einem Zeichen füllen:
```
pAddr = memset (pAddr, Character, Number);
```

6.4 Ein-/Ausgabe-Funktionen

Die umfangreichste Funktionsklasse der C-Bibliothek wird von den Ein-/ Ausgabe-Operationen gebildet. Sie enthält Funktionen, mit denen die Ein- und Ausgabe von C-Programmen aus durchgeführt werden kann, ebenso Funktionen zur Überprüfung und Formatierung von Ein-/Ausgaben sowie zur Dateiverwaltung. Die Funktionen sind in den Header-Dateien `stdio.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Fehler-Status eines Streams löschen:
```
           clearerr (pStream);
```
Alle ungeschriebenen Daten eines Streams schließen und schreiben:
```
Status = fclose (pStream);
```
Abfragen, ob das Dateiende eines Streams erreicht ist:
```
Status = feof (pStream);
```

Fehler-Status eines Streams abfragen:
```
Status = ferror (pStream);
```
Alle ungeschriebenen Daten eines Streams schreiben:
```
Status = fflush (pStream);
```
Zeichen aus einem Stream lesen:
```
Character = fgetc (pStream);
```
Aktuelle Position des Dateizeigers eines Streams ermitteln:
```
Status = fgetpos (pStream, pPosition);
```
String aus einem Stream lesen:
```
pString = fgets (pString, Number, pStream);
```
Datei öffnen und den zugehörigen Stream zurückgeben:
```
pStream = fopen (pFilename, pMode);
```
Formatierte Ausgabe in einen Stream schreiben:
```
Number = fprintf (pStream, pFormat, ...);
```
Zeichen in einen Stream schreiben:
```
Character = fputc (Character, pStream);
```
String in einen Stream schreiben:
```
Status = fputs (pString, pStream);
```
Datenblock aus einem Stream lesen:
```
Number = fread (pArray, MemberSize, Members, pStream);
```
Ein bestehender Stream zu einer anderen Datei zuordnen:
```
pStream = freopen (pFilename, pMode, pStream);
```
Formatierte Eingabe aus einem Stream lesen:
```
Number = fscanf (pStream, pFormat, ...);
```
Dateizeiger eines Streams positionieren:
```
Status = fseek (pStream, Offset, Origin);
```
Dateizeiger eines Streams setzen:
```
Status = fsetpos (pStream, pPosition);
```
Aktuelle Position des Dateizeigers eines Streams ermitteln:
```
Position = ftell (pStream);
```
Datenblock in einen Stream schreiben:
```
Number = fwrite (pArray, MemberSize, Members, pStream);
```
Zeichen aus einem Stream lesen:
```
Character = getc (pStream);
```
Zeichen aus dem Standard-Input-Kanal lesen:
```
Character = getchar ();
```
String aus dem Standard-Input-Kanal lesen:
```
pString = gets (pString);
```
Formatierte Ausgabe in den Standard-Output-Kanal schreiben:
```
Number = printf (pFormat, ...);
```
Zeichen in einen Stream schreiben:
```
Character = putc (Character, pStream);
```
Zeichen in den Standard-Output-Kanal schreiben:
```
Character = putchar (Character);
```

String in den Standard-Output-Kanal schreiben:
```
Status = puts (pString);
```
Datei löschen:
```
Status = remove (pFilename);
```
Datei umbenennen:
```
Status = rename (pOldname, pNewname);
```
Dateizeiger an den Anfang eines Streams zurücksetzen:
```
rewind (pStream);
```
Formatierte Eingabe aus dem Standard-Input-Kanal lesen:
```
Number = scanf (pFormat, ...);
```
Puffer zu einem I/O-Stream zuweisen:
```
setbuf (pStream, pBuffer);
```
Puffer zu einem I/O-Stream zuweisen:
```
Status = setvbuf (pStream, pBuffer, Mode, Size);
```
Formatierte Ausgabe in einen String schreiben:
```
Number = sprintf (pString, pFormat, ...);
```
Formatierten Eingabe aus einem String lesen:
```
Number = sscanf (pString, pFormat, ...);
```
Temporäre Datei anlegen:
```
pFile = tmpfile ();
```
Eines neuen noch nicht verwendeten Dateinamens erzeugen:
```
pString = tmpnam (pString);
```
Zeichen in einen Stream zurücklegen:
```
Character = ungetc (Character, pStream);
```
Formatierte variablen Argumentliste in einen Stream schreiben:
```
Number = vfprintf (pStream, pFormat, pArgs);
```
Formatierte variablen Argumentliste in den Standard-Output-Kanal schreiben:
```
Number = vprintf (pFormat, pArgs);
```
Formatierte variablen Argumentliste in einen String schreiben:
```
Number = vsprintf (pString, pFormat, pArgs);
```

6.5 Variable Argumente

Diese Funktionen werden benötigt, um Funktionen mit einer variablen Anzahl von Argumenten wie `printf` zu realisieren. Die Funktionen sind in den Header-Dateien `stdarg.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Variable Argumentliste auf ihr erstes Element zurücksetzen:
```
va_start (pArgs, LastFixedArg);
```
Den Wert des nächsten Arguments der variablen Argumentliste ermitteln:
```
Value = va_arg (pArgs, Type);
```
Am Ende der Verarbeitung der variablen Argumentliste aufräumen:
```
va_end (pArgs);
```

6.6 Speicherverwaltung

Diese Funktionen dienen zur Verwaltung des Hauptspeichers. Die Funktionen sind in der Header-Datei `stdlib.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Speicher für eine Anzahl von Elementen mit einer angegebenen Größe anfordern:
```
pAddr = calloc (Number, Size);
```
Speicher ab einer angegebenen Adresse freigeben:
```
free (pAddr);
```
Speicher mit einer angegebenen Größe anfordern:
```
pAddr = malloc (Size);
```
Größe eines allozierten Speicherblocks ändern:
```
pAddr = realloc (pAddr, NewSize);
```

6.7 Mathematische Funktionen

Der ANSI-Standard beinhaltet ein Reihe von portablen mathematischen Funktionen. Ein Teil dieser Funktionen kann nur auf ganzzahlige Werte angewendet werden. Sie sind in der Header-Datei `stdlib.h` deklariert. Die Gleitkomma-Funktionen sind in der Header-Datei `math.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Absolutwert einer int-Zahl berechnen:
```
Result = abs (j);
```
Arcuscosinus einer double-Zahl berechnen:
```
Result = acos (x);
```
Arcussinus einer double-Zahl berechnen:
```
Result = asin (x);
```
Arcustangens einer double-Zahl berechnen:
```
Result = atan (x);
```
Arcustangens im korrekten Quadranten für den Bruch y/x zweier double-Zahlen berechnen:
```
Result = atan2 (y, x);
```
Die nächstgrößere ganze Zahl einer double-Zahl berechnen:
```
Result = ceil (x);
```
Cosinus einer `double`-Zahl berechnen:
```
Result = cos (x);
```
Cosinushyperbolicus einer `double`-Zahl berechnen:
```
Result = cosh (x);
```
Ganzzahlige Division zweier `int`-Zahlen berechnen:
```
Result = div (Numerator, Denominator);
```
Exponentialfunktion e^x einer double-Zahl berechnen:
```
Result = exp (x);
```
Absolutwerts einer double-Zahl berechnen:
```
Result = fabs (x);
```
Nächstkleinere ganze Zahl einer double-Zahl berechnen:
```
Result = floor (x);
```
Divisionsrests zweier double-Zahlen berechnen:
```
Result = fmod (x, y);
```
Normalisierte Mantisse und des binären Exponenten einer double-Zahl ermitteln:
```
Result = frexp (Value, pExponent);
```
Absolutwert einer long-Zahl berechnen:
```
Result = labs (j);
```

$x * 2^{Exponent}$ zweier double-Zahlen berechnen:
```
Result = ldexp (x, Exponent);
```
Ganzzahlige Division zweier long-Zahlen berechnen:
```
Result = ldiv (Numerator, Denominator);
```
Natürlichen Logarithmus einer double-Zahl berechnen:
```
Result = log (x);
```
Logarithmus zur Basis 10 einer double-Zahl berechnen:
```
Result = log10 (x);
```
Den ganzzahligen und den Nachkommaanteil einer double-Zahl ermitteln:
```
Result = modf (Value, pIntPart);
```
Potenz x^y zweier double-Zahlen berechnen:
```
Result = pow (x, y);
```
Eine ganzzahlige Pseudozufallszahl erzeugen:
```
Number = rand ();
```
Sinus einer double-Zahl berechnen:
```
Result = sin (x);
```
Sinushyperbolicus einer double-Zahl berechnen:
```
Result = sinh (x);
```
Quadratwurzel einer double-Zahl berechnen:
```
Result = sqrt (x);
```
Initialisierungswert für Pseudozufallszahlen setzen:
```
           srand (Seed);
```
Tangens einer double-Zahl berechnen:
```
Result = tan (x);
```
Tangenshyperbolicus einer double-Zahl berechnen:
```
Result = tanh (x);
```

6.8 Zeit- und Datumsfunktionen

Mit diesen Funktionen können Zeit- und Datumsangaben in verschiedene Formate konvertiert und für die lokale Zeitzone angepaßt werden. Die Funktionen sind in der Header-Datei `time.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

In Komponenten aufgeteilten Zeit in einen String umwandeln:
```
pString = asctime (pTime);
```
Die Zeit ermitteln, die seit Beginn der Ausführung eines Programms vergangen ist:
```
Clock = clock ();
```
Kalenderzeit in einen String umwandeln:
```
pString = ctime (pTimer);
```
Differenz zwischen zwei Kalenderzeiten bilden:
```
Difference = difftime (Timer1, Timer0);
```
Kalenderzeit in eine in Komponenten aufgeteilte Greenwich-Zeit umwandeln:
```
pGmt = gmtime (pTimer);
```
Kalenderzeit in eine in Komponenten aufgeteilte lokale Ortszeit umwandeln:
```
pLocalTime = localtime (pTimer);
```

Einer in Komponenten aufgeteilten Zeit in eine lokale Kalenderzeit umwandeln:

```
Timer = mktime (pTime);
```

Formatierte Ausgabe von Datum und Uhrzeit in einen String erzeugen:

```
Number = strftime (pString, MaxSize, pFormat, pTime);
```

Die aktuelle Kalenderzeit des Systems ermitteln:

```
Timer = time (pTimer);
```

6.9 Lokalisierung

Die Lokalisierung definiert sogenannte Locale-Pakete, die der Anpassung von Programmen an länderspezifische Besonderheiten dienen. Die Funktionen sind in der Header-Datei `locale.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Länderspezifische Einstellungen auswählen bzw. ändern:

```
pCategory = setlocale(int Category, const char *pLocale);
```

Zahlenformate der aktuellen länderspezifischen Einstellungen ermitteln:

```
pSettings = localeconv();
```

6.10 Steuerfunktionen

Die Steuerfunktionen werden zur Festlegung von Abbruchbedingungen benötigt. Die Funktionen sind in den Header-Dateien `stdlib.h` und `assert.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Das Signal `SIGABRT` an die aufrufende Task senden:

```
abort ();
```

Eine Bedingung prüfen und bei Nichtzutreffen die Task abbrechen:

```
assert (Condition);
```

Routinen festlegen, die am Ende einer Task aufgerufen werden sollen:

```
Status = atexit (pFunc);
```

Die aufrufende Task beenden:

```
exit (Status);
```

6.11 Signalbehandlung

Mit diesen Funktionen ist es möglich, eigene Funktionen als Behandlungsroutinen (Signal-Handler) für verschiedene Ereignisse festzulegen. Tritt ein Ereignis ein, so wird ein Signal an die gerade aktive Task geschickt, d. h. die für dieses Signal zugewiesene Funktion wird aufgerufen. Es ist auch möglich, eine Standardbehandlung für das Signal zu vereinbaren oder es ganz zu ignorieren. Die Funktionen zur Signalbehandlung sind in der Header-Datei `signal.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Kontrolle an einen Signal-Handler übergeben:

```
Status = raise(Signal);
```

Signal-Handler installieren:

```
pFunc = signal(Signal,pFunc);
```

6.12 Nichtlokale Sprünge

Diese Funktionen finden Verwendung bei einem Rücksprung aus verschachtelten Routinen, falls es zu einer Fehlerbedingung gekommen ist. Mithilfe der Anweisung `longjmp` wird die Taskausführung beim letzten Aufruf von `setjmp` fortgesetzt. Die Funktionen sind in den Header-Dateien `setjmp.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Setzen des Sprungziels für einen späteren nichtlokalen Sprung:

```
Status = setjmp (Environment);
```

Ausführen eines nichtlokalen Sprungs:

```
Status = longjmp (Environment, Returnvalue);
```

6.13 Fehlerbehandlung

Die Funktionen zur Fehlerbehandlung sind in den Header-Dateien `stdio.h` und `string.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Fehlermeldung auf den Standard-Error-Kanal für den aktuellen Wert von errno ausgeben:

```
perror (pString);
```

Einen einfachen Fehlertext für eine Fehlernummer generieren:

```
pString = strerror (Errno);
```

6.14 Sonstige Funktionen

Hier sind eine Reihe von Funktionen zusammengefaßt, die sich nicht eindeutig einer Kategorie zuordnen lassen. Die Funktionen sind in den Header-Dateien `stddef.h` und `stdlib.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Nach einem Element in einem sortierten Feld suchen:

```
pElement = bsearch (pKey, pBase, Number, Size, pCompare);
```

Den Wert einer Umgebungsvariablen ermitteln:

```
pString = getenv (pName);
```

Den Abstand einer Komponente vom Strukturanfang ermitteln:

```
Offset = offsetof(Type, Member);
```

Die Elemente eines Feldes sortieren:

```
qsort (pBase, Number, Size, pCompare);
```

Ein Kommando durch den Kommandointerpreter ausführen:

```
Status = system (pCommand);
```

7 Systemdienste des Netzwerk-Managers

Die EUROS-Komponente Netzwerk-Manager implementiert die TCP/IP-Protokolle und die Socket-Schnittstelle.

7.1 Initialisierungs- und Statusfunktionen

Die Komponente Netzwerk-Manager stellt eine Reihe von Initialisierungs- und Statusfunktionen zur Verfügung, die den Netzwerk-Manager vorbereiten und dem Anwender bestimmte Statusinformationen liefern. Die folgenden Systemaufrufe sind implementiert:

```
Status = NetInit(NumSockets, NumClusters, NumBuffers, NumPcb,
                 NumUtilBlocks);
```
Initialisierung des Netzwerk-Managers

```
Status = netctl(Option, pData, Size);
```
Optionen des Netzwerk-Managers setzen und abfragen

```
Status = gethostname(name, namelen);
```
Hostname abfragen

```
Status = sethostname(name, namelen);
```
Hostname setzen

7.2 Socket-Funktionen

Die Socket-Funktionen werden verwendet, um Sockets zu erzeugen, zu schließen, zu verbinden und Verbindungen zu lösen. Folgende Funktionen sind implementiert:

```
Socket = socket(domain, type, protocol);
```
Erzeugt einen Socket

```
Status = soclose(socket);
```
Schließt einen Socket

```
Status = connect(socket, name, namelen);
```
Verbindet einen Socket

```
Status = shutdown(socket, mode);
```
Schließt einen Teil einer Vollduplex-Verbindung

```
Status = bind(socket, name, namelen);
```
Bindet einen Socket an eine lokale Adresse

```
Status = listen(socket, backlog);
```
Versetzt Socket in den Listen-Zustand

```
Socket = accept(socket, addr, addrlen);
```
Akzeptiert Verbindung mit einem Socket

7.3 Datentransfer

Die Funktionen zum Datentransfer werden verwendet, um Daten über Sockets zu senden oder zu empfangen. Folgende Funktionen sind implementiert:

```
Number = recv(socket, buffer, len, flags);
```
Empfängt Daten über einen Socket

```
Number = recvfrom(socket, buffer, len, flags, from, fromlen);
```
Empfängt Datagramm

```
Number = send(socket, message, len, flags);
```
Sendet Daten über einen Socket

```
Number = sendto(socket, message, len, flags, to, tolen);
```
Sendet Datagramm

7.4 Änderung der Byte-Reihenfolge

Diese Funktionen werden verwendet, um die Reihenfolge der Bytes in einem 16- oder 32-Bit-Wort zu vertauschen. Folgende Funktionen sind implementiert:

```
Value = htonl(Value);
```
32-Bit-Wert von Host-Darstellung in Netz-Darstellung umwandeln

```
Value = htons(Value);
```
16-Bit-Wert von Host-Darstellung in Netz-Darstellung umwandeln

```
Value = ntohl(Value);
```
32-Bit-Wert von Netz-Darstellung in Host-Darstellung umwandeln

```
Value = htohs(Value);
```
16-Bit-Wert von Netz-Darstellung in Host-Darstellung umwandeln

```
Value = bswap(Value);
```
Bytes eines 16-Bit-Werts vertauschen

```
Value = lswap(Value);
```
Bytes eines 32-Bit-Werts vertauschen

7.5 Socket-Hilfsfunktionen

Mit den Socket-Hilfsfunktionen kann man Adressen von Sockets abfragen und Socket-Optionen setzen und abfragen. Folgende Funktionen sind implementiert:

```
Status = getpeername(socket, name, namelen);
```
Adresse der Gegenseite eines Sockets abfragen

```
Status = getsockname(socket, name, namelen);
```
Adresse der lokalen Seite eines Sockets abfragen

```
Status = getsockopt(socket, level, optname, optval, optlen);
```
Socket-Option abfragen

```
Status = setsockopt(socket, level, optname, optval, optlen);
```
Socket-Option setzen

```
Status = soioctl(socket, cmd, data);
```
I/O-Modus eines Sockets setzen

7.6 Konvertierung von Internet-Adressen

Mit diesen Funktionen können Internet-Adressen manipuliert sowie zwischen Textdarstellung und Zahlendarstellung konvertiert werden. Folgende Funktionen sind implementiert:

```
Addr = inet_addr(cp);
```
Konvertiert Textdarstellung in Adresse

```
Status = inet_aton(cp, addr);
```
Konvertiert Textdarstellung in Adresse

```
Addr = inet_lnaof(in);
```
Ermittelt lokale Netzadresse einer Internet-Adresse

```
Addr = inet_netof(in);
```
Ermittelt Netzwerkadresse einer Internet-Adresse

```
Addr = inet_makeaddr(net, host);
```
Erstellt Internet-Adresse aus Netzadresse und Hostadresse

```
Addr = inet_network(cp);
```
Konvertiert Textdarstellung einer Netzadresse in Zahlendarstellung

```
pText = inet_ntoa(in);
```
Konvertiert Zahlendarstellung einer Internet-Adresse in Textdarstellung

```
pText = inet_ntoa_r(in, pDest);
```
Konvertiert Zahlendarstellung einer Internet-Adresse in Textdarstellung (reentrant)

8 POSIX-Schnittstelle

Die POSIX-Schnittstelle, die als eine EUROS-Komponente implementiert ist, stellt dem Anwender einen Teil der im ISO-Standard IEEE Std 1003.1-1990 festgelegten Standards zur Verfügung.

8.1 Initialisierungs- und Statusfunktionen

Die POSIX-Schnittstelle ist eine EUROS-Komponente und stellt eine Reihe von Initialisierungs- und Statusfunktionen zur Verfügung. Diese Funktionen bereiten die POSIX-Schnittstelle vor und liefern dem Anwender bestimmte Statusinformationen. Sie sind *nicht* Teil des POSIX-Standards. Die Funktionen sind in der Header-Datei `posix.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

POSIX-Schnittstelle initialisieren:

```
Status = PosixInit ();
```

Status der POSIX-Schnittstelle abfragen:

```
Status = PosixStatus (pStatus);
```

8.2 Ein-/Ausgabe-Funktionen

Zur Bearbeitung von Dateien stellt die POSIX-Schnittstelle nur einen Teil der im POSIX-Standard festgelegten Funktionen bereit. Die Funktionen sind in der Header-Datei `unistd.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Datei öffnen:

```
Fd = open (pPath, OpenFlags, Permissions);
```

Datei schließen:

```
Status = close (Fd);
```

Dateizeiger positionieren:

```
Status = lseek (Fd, Offset, Origin);
```

Datei lesen:

```
Number = read (Fd, pBuffer, Maximum);
```

Datei schreiben:

```
Number = write (Fd, pBuffer, Maximum);
```

9 Hilfsdienste

EUROS bietet eine Reihe von Hilfsdiensten, die den Anwender von der Notwendigkeit befreien, eigene Routinen zu programmieren, die diesen Zweck erfüllen. Außerdem berücksichtigen die Hilfsdienste die Besonderheiten der zugrundeliegenden Hardware-Architektur.

Zu den Hilfsdiensten gehören Funktionen zur Bearbeitung von doppelt verketteten Listen, Puffern und Ring-Puffern.

9.1 Dienste zur Puffer-Bearbeitung

Die Dienste zur Puffer-Bearbeitung gehören zu den von EUROS angebotenen Hilfsdiensten. Die Funktionen sind in der Header-Datei `buffer.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Die ersten n Stellen eines Puffers mit einem Byte füllen:
```
Status = FillBytes (pBuffer, Number, Byte);
```
Die ersten n Stellen eines Quell-Puffers bytewise in den Ziel-Puffer kopieren:
```
Status = CopyBytes (pSource, pDest, Number);
```
Die ersten n Stellen eines Quell-Puffers wortweise in den Ziel-Puffer kopieren:
```
Status = CopyWords (pSource, pDest, Number);
```
Die ersten n Stellen eines Quell-Puffers doppelt-wortweise in den Ziel-Puffer kopieren:
```
Status = CopyLongs (pSource, pDest, Number);
```
N Worte vom Quell- in den Ziel-Puffer kopieren, wobei High- und Low-Bytes vertauscht werden:
```
Status = CopySwappedBytes (pSource, pDest, Number);
```
Die ersten n Byte-Stellen zweier Puffer vertauschen:
```
Status = SwapContents (pBuffer1, pBuffer2, Number);
```
Die Reihenfolge der ersten n Byte-Stellen eines Puffers invertieren:
```
Status = InvertOrder (pBuffer, Number);
```

9.2 Dienste zur Verwaltung doppelt verketteter Listen

Die Dienste zur Verwaltung doppelt verketteter Listen gehören zu den von EUROS angebotenen Hilfsdiensten. Die Funktionen sind in der Header-Datei `list.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Liste initialisieren:
```
Status = ListInit (pList);
```
Liste leeren:
```
Status = ListFree (pList);
```
Liste an das Ende einer anderen Liste anhängen:
```
Status = ListAppend (pList, pAppendList);
```
Liste aus einer anderen Liste extrahieren:
```
Status = ListExtract (pList, pStartNode, pEndNode, pExtractList);
```
Anzahl der Elemente einer Liste ermitteln:
```
Status = ListCount (pList);
```
Feststellen, ob eine Liste leer ist:
```
Status = ListIsEmpty (pList);
```
Knoten am Ende einer Liste anhängen:
```
Status = ListAppendNode (pList, pNewNode);
```

Knoten in eine Liste hinter dem angegebenen Knoten einfügen:
```
Status = ListInsertNode (pList, pPrevNode, pNewNode);
```
Knoten aus einer Liste entfernen:
```
Status = ListRemoveNode (pList, pNode);
```
Knoten in einer Liste suchen:
```
Status = ListFindNode (pList, pNode);
```
Den ersten Knoten in einer Liste ermitteln:
```
pNode =ListFirstNode (pList);
```
Den ersten Knoten in einer Liste ermitteln und entfernen:
```
pNode =ListGetNode (pList);
```
Den letzten Knoten in einer Liste ermitteln:
```
pNode =ListLastNode (pList);
```
Den nächsten Knoten in einer Liste ermitteln:
```
pNode =ListNextNode (pNode);
```
Den vorhergehenden Knoten in einer Liste ermitteln:
```
pNode =ListPrevNode (pNode);
```
Den Knoten in einer Liste an der angegebenen Position ermitteln:
```
pNode =  ListNthNode (pList, Position);
```
Den Knoten in einer Liste anhand des angegebenen Knotens und Offsets ermitteln:
```
pNode =  ListSkipNodes (pNode, Offset);
```

9.3 Dienste zur Ring-Puffer-Verwaltung

Die Dienste zur Verwaltung von Ring-Puffern gehören zu den von EUROS angebotenen Hilfsdiensten. Die Funktionen und Makros sind in der Header-Datei `ringbuf.h` deklariert. Die folgenden Systemaufrufe sind implementiert:

Ring-Puffer erzeugen:
```
pRingBuf = RingCreate (Size);
```
Ring-Puffer löschen:
```
Status = RingDelete (pRingBuf);
```
Ring-Puffer in den leeren Zustand zurücksetzen:
```
Status = RingFlush (pRingBuf);
```
Anzahl Bytes aus einem Ring-Puffer lesen und in einen Puffer kopieren:
```
Status = RingGet (pRingBuf, pBuffer, Maximum);
```
Anzahl Bytes von einem Puffer in einen Ring-Puffer schreiben:
```
Status = RingPut (pRingBuf, pBuffer, Maximum);
```
Ein Byte schreiben, ohne den Schreib-Zeiger des Ring-Puffers zu aktualisieren:
```
Status = RingPutAhead (pRingBuf, Byte, Offset);
```
Schreib-Zeiger des Ring-Puffers um n Stellen nach vorne positionieren:
```
Status = RingMoveAhead (pRingBuf, Number);
```
Abfragen, ob ein Ring-Puffer leer ist:
```
Status = RingIsEmpty (pRingBuf);
```
Abfragen, ob ein Ring-Puffer voll ist:
```
Status = RingIsFull (pRingBuf);
```
Abfragen, wieviele Bytes im Puffer frei sind:
```
Status = RingBytesFree (pRingBuf);
```

Abfragen, wieviele Bytes im Puffer belegt sind:

```
Status = RingBytesUsed (pRingBuf);
```

10 Parameter und Ergebniswerte

10.1 Standardparameter

Die EUROS-Systemaufrufe zur Anforderung von Operationen mit Systemobjekten weisen eine Reihe von wählbaren Zeit- und Prioritätsbedingungen und Optionen auf. Um dem Anwender eine möglichst einheitliche Aufrufschnittstelle anzubieten, werden Standardparameter vereinbart, die bei allen Systemobjekten den gleichen Aufbau haben.

10.1.1 `ObjMode` - Objekt-Modus

Der Parameter `ObjMode` erscheint bei allen `Create`-Aufrufen als zweiter Parameter hinter dem Pfadnamen; er legt grundsätzliche Eigenschaften des neuen Objekts fest. Der Parameter `ObjMode` ist ein vorzeichenloser 16-Bit-Wert vom Typ `uint16`, der mit einer Auswahl der folgenden Makros bestückt werden kann:

`EXCL`	Objekt exklusiv für die erzeugende Task anlegen.
`NODEL`	Objekt gegen Löschen schützen.
`FIFO`	Objekt-Warteschlangen nach der Reihenfolge der Auftragseingänge verwalten.
`FIFO_MSG`	Nachrichten-Warteschlange nach der Reihenfolge der Auftragseingänge verwalten.

10.1.2 `ExecMode` - Ausführungsmodus

Der Parameter `ExecMode` ist ein vorzeichenloser 16-Bit-Wert vom Typ `uint16`, mit dem die Arbeitsweise vieler Systemdienste gesteuert werden kann. Er kann mit einer Auswahl der folgenden Makros bestückt werden:

`SINGLE`	Nur das eine angegebene Objekt wird angesprochen.
`CLUSTER`	Das eine angegebene und alle darunterliegenden Objekte werden angesprochen.
`AND_TEST`	Alle getesteten Bits müssen übereinstimmen.
`OR_TEST`	Mindestens eines der getesteten Bits muß übereinstimmen.
`FIRST_MATCH`	Mindestens ein Objekt des Clusters muß die Bedingung erfüllen bzw. eine Aktion wird nur für das erste passende Objekt ausgeführt.
`ALL_MATCH`	Alle Objekte des Clusters müssen die Bedingung erfüllen bzw. eine Aktion wird für jedes Objekt des Clusters ausgeführt.
`DEL_COND`	Belegte Objekte werden nur als gelöscht markiert, falls sie nicht sofort gelöscht werden können.
`DEL_FORCED`	Es werden alle noch ausstehenden Aufträge entsorgt und anschließend wird das vormals belegte Objekt gelöscht. Hat das Objekt Kinder, so wird es nur als gelöscht markiert.
`START_QUEUED`	Die Task starten, wenn sie `DORMANT` ist, sonst einen Auftrag an die Restart-Queue der Task anhängen.
`START_DORMANT`	Die Task starten, wenn sie `DORMANT` ist, sonst nicht.
`ALLOC_ASC`	Speicher vom Anfang des Pools her suchen und anfordern.
`ALLOC_DESC`	Speicher vom Ende des Pools her suchen und anfordern.
`INVERT`	Die maskierten Bits werden bei jedem Aufruf invertiert.
`NO_INVERT`	Die maskierten Bits werden nicht invertiert.

WT_START/WT_STRT	Synchron warten, bis die erste Task gestartet ist.
WT_END	Synchron warten, bis sich die gestartete Task beendet hat.
WAIT/WT	Synchron warten, bis der Auftrag beendet ist.
CHECK/ASYNC/CONT	Auftrag ohne Warten asynchron ausführen.
C_SEF	Asynchron warten und Auftragsende durch Setzen eines der privaten Eventflags signalisieren.
C_REF	Asynchron warten und Auftragsende durch Rücksetzen eines der privaten Eventflags signalisieren.
C_PSG	Asynchron warten und Auftragsende durch Senden eines der privaten Signale signalisieren.

10.1.3 `TimeLimit` - Zeitüberwachung / Zeitverzögerung

Der Parameter `TimeLimit` ist ein vorzeichenloser 32-Bit-Wert vom Typ `uint32`, der bei allen Systemaufrufen eingesetzt wird, die mit der Angabe eines Zeitintervalls verbunden sind. Es lassen sich hier zwei grundsätzliche Anwendungsfälle unterscheiden:

- Zeitverzögerungen:
  Es wird ein Intervall vorgegeben, nach dessen Ablauf die eigentliche Aktion durch das Betriebssystem ausgeführt werden soll (z.B. `SignalSendTime`: Senden eines Signals nach Zeitintervall, `EventFlagSetTime`: Setzen / Rücksetzen von Eventflags nach Zeitintervall). Solche Systemaufrufe (auch ‘zeitverzögerte Dienste’ genannt) setzen ihren Auftrag an das Betriebssystem ab, dieser wird in einer Warteschlange vermerkt, und die anfordernde Task setzt ihre Arbeit fort. Ein endloses Zeitintervall durch Angabe von `TimeLimit = FOREVER` ist in nicht zulässig.
- Zeitüberwachungen:
  Es wird ein Timeout-Wert vorgegeben, um die Ausführung eines Systemdienstes zeitlich zu überwachen. Davon sind fast alle Systemaufrufe betroffen, die den Parameter `ExecMode` (Ausführungsmodus) haben. Enthält `ExecMode` den Wert `WAIT` (d.h. die Task wird blockiert bis der Aufruf befriedigt werden kann), so begrenzt `TimeLimit` die maximale Wartezeit - bei allen anderen Werten von `ExecMode` hat `TimeLimit` keine Bedeutung. Läuft die Zeitüberwachung `TimeLimit` ab, ohne daß die Anforderung befriedigt werden konnte, so wird die Task aus dem blockierten Zustand wieder entlassen und durch eine entsprechende Fehlermeldung benachrichtigt (Rückgabewert `FAIL` und `errno = ETIME`). Bei Zeitüberwachungen ist ein endloses Warten durch Angabe von `TimeLimit = FOREVER` erlaubt.
  Der wesentliche Vorteil des Parameters `TimeLimit` zur Überwachung von Wartezuständen liegt darin, daß bei jedem einzelnen Systemaufruf ein spezifisches, den jeweiligen Anforderungen angepaßtes Zeitintervall gewählt werden kann. Die Länge der Zeitüberwachung ist also keine pauschale Eigenschaft einer Task bzw. eines Systemobjekts (oder gar der gesamten Applikation).

Der Parameter `TimeLimit` kann sowohl als absolute als auch als relative Zeit angegeben werden. Eine absolute Zeitangabe mit Datum und Uhrzeit kann durch Angabe von Jahr, Monat, Tag, Stunde, Minute und Sekunde erfolgen. Eine relative Zeitangabe kann in Nanosekunden, Mikrosekunden, Millisekunden, Sekunden, Minuten, Stunden oder Tagen erfolgen. Die genauen Rahmenbedingungen und weiteren Möglichkeiten sind ausführlich im Kapitel ‘TimeLimit’ im ‘Reference Manual’ beschrieben.

10.1.4 `Prio` - Priorität

Der Parameter `Prio` (priority) tritt in Verbindung mit Systemaufrufen auf, die mit der Festlegung von Prioritäten verbunden sind. `Prio` ist ein vorzeichenloser 16-Bit-Wert, der aus zwei Bestandteilen zusammengesetzt ist:

- einer Kennung für das Verfahren, nach dem der Prioritätswert bestimmt wird, und

- einer Zahl von 0 bis 255.

Der Wertebereich der Prioritäten für Anwendertasks reicht von 0 (niedrigste Priorität) bis 255 (höchste Priorität); größere Werte sind für systeminterne Zwecke reserviert.

Folgende Kennungen für die Prioritätsfestlegung sind definiert:

`PRIO_VALUE`	Den angegebenen Wert als Priorität verwenden.
`PRIO_URGENT`	Die Priorität ist höher als bei allen anderen normalen Tasks.
`PRIO_RQ_CUR`	Die aktuelle Priorität der aufrufenden Task verwenden.
`PRIO_RQ_INH`	Die ursprüngliche Priorität der aufrufenden Task verwenden.
`PRIO_TAR_INH`	Die ursprüngliche Priorität der Zieltask verwenden.
`PRIO_LARGE`	Von ursprünglicher Priorität der Zieltask und aktueller Priorität der aufrufenden Task die größere als Priorität verwenden.
`PRIO_INC`	Die aktuelle Priorität um den angegebenen Wert erhöhen.
`PRIO_DEC`	Die aktuelle Priorität um den angegebenen Wert erniedrigen.

10.2 Rückgabewerte

Die meisten EUROS-Systemaufrufe sind als Funktionen implementiert, die einen Wert zurückliefern. Dieser Rückgabewert dient dazu festzustellen, ob eine Funktion erfolgreich abgearbeitet wurde, oder ob die Abarbeitung der Funktion wegen eines Fehlers abgebrochen wurde. Die wenigen Funktionen, die keinen Rückgabewert liefern, arbeiten immer erfolgreich oder haben keine Möglichkeit festzustellen, ob ein Fehler aufgetreten ist.

Im Fall, daß eine Funktion erfolgreich abgearbeitet wurde, ist die Bedeutung des Rückgabewertes sehr stark von der aufgerufenen Funktion abhängig und kann im zugehörigen Kapitel bei dieser Funktion nachgeschlagen werden. Im Fehlerfall ist der Rückgabewert eindeutig festgelegt. Bei Funktionen, die einen Zeiger beliebiger Art liefern ist der Rückgabewert im Fehlerfall gleich `NULL`. Alle anderen Funktionen, die einen Rückgabewert liefern, geben im Fehlerfall `FAIL` zurück.

Die genaue Fehlerursache ist in der Standard-Fehlervariable `errno` hinterlegt. Eine genaue Beschreibung von `errno` ist im Kapitel ‘Fehlermeldungen in errno’ im ‘Reference Manual’ zu finden.

Published by

SYSTEC

Ordering No. L-430d_1
Printed in Germany