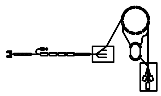

M O P S
Das Micro Operating System

Technisches Handbuch

2. Auflage

Ludwig Hechler

GSI, Gesellschaft für Schwerionenforschung mbH
Planckstraße 1, D-64291 Darmstadt
Tel.: 06159/71-0

Issued: 25. Jan 1996
Key: B-MOPS-05
Printed: 28. Mai 2009

Dieses technische Handbuch ist die Beschreibung von MOPS*, der Systemsoftware auf der Repräsentationsebene des Beschleunigerkontrollsystems.*
Die 2. Auflage bezieht sich auf die Entwickungsumgebung Organon *der Firma CAD-UL. Nur wo Rückbezüge auf das alte Entwicklungssystem der Firma* Oregon *notwendig sind, wird dieses, soweit nötig, erwähnt.*
Eine genaue Beschreibung des MOPS auf Basis von Oregon befindet sich in der 1. Auflage des technischen Handbuchs vom August 1990.

Document Revision History		
Date	Name	Comment
27. Aug. 87	E. Schaffner	Dokument erstellt
16. Jan. 90	G. Schwarz	Auf den neuesten Stand gebracht
8. Aug. 90	L. Hechler	Für neuen Compiler: Tausch A5 mit A6
15. Aug. 90	L. Hechler	In TEX-Form gebracht
18. Mar. 93	L. Hechler	Zweite, komplett überarbeitete Auflage in Zusammenhang mit dem neuen Entwicklungssystem Organon von CAD-UL. (Schlüssel: B-MOPS-05)
1. Mar. 95	L. Hechler	Erklärung von XLIB$GBLVAR
7. Jun. 06	L. Hechler	Übersetzbar auf dem ASL-Cluster. Keine inhaltlichen Anpassungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Überblick

1.1 Einbindung des Gruppenmikros in das Kontrollsystem

Das Beschleunigerkontrollsystem lässt sich hinsichtlich Aufgabenbereich und Funktionalität in drei logische Ebenen untergliedern.

- In die *obere* Ebene der Beschleunigerphysik und der Operatingalgorithmen mit den entsprechenden Benutzerschnittstellen zu den Kontrollsystemdiensten, die auf den VMS-Knotenrechnern angesiedelt sind.

- In die *mittlere* Ebene der Umsetzung der Aufträge der oberen Ebene (Gerätemodell) in Ablaufsteuerungen, Datenkonversionen und -verdichtungen und Kommunikation mit der unteren Ebene. Diese Ebene ist durch die Gruppenmikroprozessor-Systeme (GuP) abgedeckt, die als *Master* in den VME-Bus-Rahmen implementiert sind. Die Systemsoftware auf den GuPs inklusive ihrer Schnittstellen zu den anwendungsspezifischen Programmen auf dieser Ebene wird mit dem Namen *Micro Operating System* (MOPS) bezeichnet.

- In die *untere* Ebene des zeitkritischen, eventgetriggerten Zugriffs auf die Prozessdaten. Diese Ebene wird durch die sogenannten *Steuereinheiten* (SEs) abgedeckt, die als *Slaves* mit der Systemsoftware *Equipment Control Monitor* (ECM) in den VME-Bus-Rahmen implementiert sind.

MOPS ist das Objekt dieser Dokumentation.

1.2 Anwendungen unter MOPS

MOPS ist eine Schale, die kontrollsystem-spezifische Dienstleitungen zur Verfügung stellt. Die wesentliche Funktion dieser Schale ist die Bereitstellung einer einfachen und komfortablen Schnittstelle für die Anwendungsprogramme. Einfach und komfortabel heißt hier, dass ein Anwendungsprogramm mit minimalen Kenntnissen der Abläufe und Strukturen von MOPS integriert werden kann, dass komplizierte MOPS-Interna keinen Niederschlag im Code und in den Abläufen des Anwenderprogramms finden, dass neue Anwenderprogramme leicht hinzugefügt und existierende leicht geändert werden können. Die Verwaltung der Anwenderprogramme, d. h., der Aufruf dieser Programme, die Speicherverwaltung für ihre Empfangs- und Sendedaten und die Verknüpfung mit äußeren Ereignissen (Events und Interrupts) werden von MOPS abgewickelt und sind dem Anwender verborgen. Das Ansprechen dieser Verwaltungsfunktionalitäten von MOPS ist auf die höhere Ebene des Kontrollsystems (Alphas) verlagert, wo dafür eine (ebenfalls einfache und komfortable) Schnittstelle namens *Userface* zur Verfügung steht.

1.3 USRs und SSRs

Die einfache Konstruktion der Anwenderprogramme, die daraus resultiert, ist folgende: Die Anwenderprogramme sind als Prozeduren implementiert. Die Schnittstelle zu MOPS ist ein Standardrahmen mit einigen Support-Routinen für die Prozeduren und für jede Prozedur ein (mitzuliefernder) Kontrollblock mit einigen Verwaltungsparametern (standardisierte Numerierung der Prozedur, Name der Prozedur, Bytecount der Antwortdaten etc).
Die Anwenderprogramme werden als User Service Routinen (USRs) bezeichnet, die dazugehörigen Kontrollblöcke als USR Control Blocks (UCBs).
Von MOPS wird diese Anwenderprogrammstruktur selbst verwendet. Für Kontrollsystemdienstleistungen wie das Hochladen einer lokalen Datenbasis zu einer Alpha oder das Feststellen einer vom GuP kontrollierten Hardware-Konfiguration sind System Service Routines (SSRs) mit SSR Control Blocks (SCBs) implementiert. Diese haben die gleichen Formate und Anbindungen wie die USRs und können ebenfalls wie die USRs von der VMS-Ebene angesprochen werden.

1.4 Die Tasks des MOPS

Eine Task ist eine ablauffähige Einheit mit einer virtuellen, isolierten Umgebung, die durch den pSOS+-Kern vorgegeben wird. In dieser Umgebung konkurriert sie um die verfügbaren Ressourcen (CPU, Speicherplatz, I/O-Devices etc).
Eine Kollektion von Tasks ist die adäquate Organisationsform für die Aufgaben von MOPS und die Umgebung, in die es eingebettet ist.

- Der gesamte MOPS-Code ist so umfangreich, dass er in kleinere Einheiten aufgeteilt werden muss. Diese Einheiten haben eine beschränkte, definierte Funktion. Wenn eine solche Einheit als pSOS+-Task implementiert ist, dann kann sie sich auf die Erfüllung dieser Funktion beschränken, da ihr aus ihrer Sicht der Rechner allein gehört und ihre Konkurrenz zu den anderen Tasks von pSOS+ verwaltet wird.

- Tasks sind mit Prioritäten versehen. Das heißt, die Behandlung wichtiger Ereignisse wird der Behandlung weniger wichtiger Ereignisse vorgezogen, oder sie unterbricht diese sogar.

- Tasks können Zeitfunktionen aufrufen, sich z. B. für eine gewisse Zeit suspendieren oder ein Timeout aufsetzen. Während dieser Zeit können andere Tasks die CPU benutzen. Mit dieser Möglichkeit können mehrere (Anwender-) Aufträge, die durch Wartezeiten an äußere Ereignisse gebunden sind, quasi-parallel bearbeitet werden.

Die Unterteilung des Systems in kleine Bearbeitungseinheiten hat zur Folge, dass diese Tasks mit gemeinsamen Daten zu tun haben. Für gemeinsame Daten gibt es zwei Organisationsmöglichkeiten. Sie können in einem Speicherbereich liegen, auf den mehrere Tasks zugreifen, oder sie können zwischen Tasks über Messagequeues ausgetauscht (gesendet und empfangen, d. h. kopiert) werden. In MOPS sind beide Strukturen implementiert. Die Arbeitsdaten für die Anwendungen in MOPS, d. h., die Sendepakete von den Alphas und die Antwortpakete an die Alphas, werden in einem gemeinsamen Speicherbereich gehalten. Wegen des beträchtlichen Umfangs (Multipakete bis zu 8,5kB) wäre ein Hin- und Herkopieren dieser Daten wenig effektiv. Die Bearbeitungsaufträge zwischen den Tasks in MOPS werden dagegen über Messagequeues abgewickelt, weil die Auftragsinformation selbst nur wenige Bytes lang ist und weil durch die Messagequeues das Queuing und die Sequentialisierung von Aufträgen durch pSOS+ verwaltet wird.
Der prinzipielle Ablauf aller MOPS-Tasks ist gleich. Sie werden kreiert und gestartet, führen Initialisierungsaktionen aus und warten dann auf Aufträge. Diese Aufträge werden durch Event Flags, Messages oder Timer übermittelt. Nach der Abarbeitung eines Auftrags geht die Task dann wieder in ihren Wartezustand. Die Tasks des MOPS werden im Folgenden aufgeführt und ihre Funktionen kurz erklärt.

ROOT ist die Wurzel des Systems. Sie wird von pSOS+ kreiert und gestartet, kreiert und startet alle anderen Tasks (bis auf IDLE) und übernimmt die für MOPS notwendigen Initialisierungen.

RECV übernimmt die empfangenen Ethernetpakete vom Communication Processor (CP) und übergibt sie via QRCV an einen Dispatcher. RECV wird über einen Interrupt vom CP gestartet.

XEND wird ebenfalls vom CP via Interrupt gestartet, wenn dieser ein Sendepaket verschickt hat. XEND hat die Aufgabe, das Memory, das das Sendepaket enthielt, wieder freizugeben.

DInn sind die Dispatcher, von denen es 7 gibt (DI01. . . DI07). Sie sind verantwortlich für die Auftragsverwaltung und den Aufruf der entsprechenden XSRs. Alle Dispatcher warten an der Messagequeue QRCV auf Aufträge.

PERI ist die Task, die die Queue für periodische Aufträge verwaltet. Sie insertiert neue und löscht alte Aufträge und schickt Aufträge, die reif sind für die Bearbeitung, via QRCV an einen Dispatcher. PERI wird durch pSOS+-Events, die von bestimmten SSRs oder von der Timer Interrupt Service Routine gesendet wurden, gestartet.

RUPT ist die Task, die die Queue für interrupt-konnektierte Aufträge verwaltet. Sie insertiert neue und löscht alte Aufträge und schickt Aufträge, die durch Interrupts der SE initiiert wurden, via QRCV an einen Dispatcher. RUPT wird durch pSOS+-Events, die von bestimmten SSRs oder von der entsprechenden ECM Interrupt Service Routine gesendet wurden, gestartet.

ALRM ist die Task, die von allen Komponenten des MOPS Alarme entgegennimmt, sie eventuell zwischenspeichert und verschickt. ALRM wartet an der Messagequeue QALR auf Alarme.

TIMR ist die Task, die ALRM veranlasst zwischengespeicherte Alarm zu verschicken.

SHOW ist die Task, die für die gesamte lokale Bedienung des MOPS verantwortlich ist.

EHDL ist die Task, die das Errorhandling des MOPS übernimmt. Fast alle Fehler, die MOPS abfängt, landen letzenendes im Errorhandler EHDL.

IDLE ist eine pSOS+-eigene Task, die nur läuft, wenn keine andere Task `ready` ist. Sie führt eine STOP-Instruktion aus, was dazu führt, dass der Prozessor bis zum nächsten Interrupt oder Reset angehalten wird.

1.5 Die Bibliotheken des MOPS

Bibliotheken liefern dem Benutzer vorgefertigte Prozeduren und Funktionen, die in den verschiedensten Anwendungen immer wieder gebraucht werden. Sie können grundsätzlich in zwei Arten realisiert sein.
Einmal gibt es die Möglichkeit, die gewünschten Routinen zur Linkzeit der Anwendung aus der Bibliothek zu holen und sie zur Anwendung dazuzubinden. Das benötigt zum Teil viel Speicherplatz, wenn die Anwendung aus vielen, unabhängigen Teilen[1] besteht, da *jedes* Teil die benötigten Bibliotheksroutinen dazugelinkt haben muss. Zudem müssen alle Teile neu gelinkt werden, wenn eine Routine der Bibliothek geändert wurde, weil sie zum Beispiel einen Bug enthielt.
Die zweite Möglichkeit ist die der Laufzeitbibliothek. Diese wird in MOPS angewendet. Die Laufzeitbibliothek befindet sich sozusagen schon im Hauptspeicher der Anwendung, also hier des MOPS. Sie besitzt spezielle Eingänge, sogenannte Sprungtabellen, die auf festen Adressen liegen. Erst in diesen Eingängen werden die eigentlichen Bibliotheksroutinen aufgerufen. Eine Anwendung wird zur Linkzeit *nur* mit diesen Eingängen (mit der Sprungtabelle) gelinkt und nicht mit den Routinen selbst. Die Verbindung zwischen Anwendung und Bibliotheksroutine wird erst zur Laufzeit über die Bibliothekseingänge hergestellt.
Dies hat wesentliche Vorteile gegenüber einer „Linkzeit"-Bibliothek. Da für die *gesamte* Anwendung jede Bibliotheksroutine nur genau einmal im Speicher liegt, kann Platz gespart werden. Allerdings liegen auch Routinen in der Laufzeitbibliothek, die von *keiner* Anwendung gebraucht werden. Dies aus dem Grund, weil man zur Zeit der Generierung der Bibliothek noch nicht genau wissen kann, welche Routinen von Anwendungen (später einmal) gebraucht werden und welche nicht.
Ein weiterer Vorteil ist die für Anwendungen transparente Möglichkeit der Änderung von Routinen der Bibliothek. Solange die Schnittstelle erhalten bleibt, können Bibliotheksroutinen völlig umgeschrieben und im Speicher verschoben werden, ohne dass das für die Anwendung zu Erkennen wäre. Es besteht auch keine Notwendigkeit die Anwendungen neu zu linken. Ein (kompatibler) Austausch der Laufzeitbibliothek führt automatisch dazu, dass *alle* Anwendungen sofort die neue Bibliothek benutzen.
Auf dem Gruppenmikro gibt es zwei Laufzeitbibliotheken. Die eine ist die compiler- und systemspezifische (Pascal, pSOS+), die andere die MOPS-spezifische Bibliothek. Beide Bibliotheken haben jeweils eine eigene Sprungtabelle und können von den Anwendungen (XSRs) und natürlich auch von MOPS selbst benutzt werden. Anwendungen haben allerdings nur eingeschränkten Zugriff, der durch entsprechende Definitionsmodule realisiert ist.
Alle Prozeduren und Funktionen der Laufzeitbibliothek müssen wiedereintrittsfähig (re-entrant) sein. Näheres zu diesem *wichtigen* Thema findet sich in Abschnitt 3.6.

[1] Damit sind Einheiten gemeint, die unabhängig voneinander gelinkt werden. So sind zum Beispiel alle Tasks des MOPS unabhängige Linkeinheiten.

Der genaue Mechanismus der Generierung und des Aufrufs einer Laufzeitbibliothek ist in Abschnitt 3.5 erklärt.
Die Schnittstelle zur MOPS-spezifischen Bibliothek ist in den Dateien `USR$SUPPORT$DEF.PIN` und `ALARM$SUPPORT.PIN` definiert und in [7] dokumentiert.

2 Elemente

2.1 Der Betriebssystemkern pSOS+

MOPS basiert auf dem Echtzeit-Betriebssystemkern pSOS+. Im Gegensatz zu dem System MOPS, das spezifische Dienstleistungen für das Kontrollsystem zur Verfügung stellt, ist pSOS+ ein klassisches Multitasking-Betriebssystem mit System Services für Ein/Ausgabe, Memory-, Message-, Semaphore- und Eventflag-Verwaltung, Task-Scheduling, Zeitverwaltung und so weiter.
pSOS+ ist ein Produkt der Software Components Group, Santa Clara, Kalifornien. Die Gründe, gerade das hierzulande nicht sehr bekannte pSOS+ zu wählen, waren folgende:

- Durch den jahrelangen Einsatz in amerikanischen Unternehmen wie IBM, Boeing und General Electric ist pSOS+ ist ein ausgereiftes und sicheres System ohne Kinderkrankheiten. Im Rahmen seines Einsatzes für MOPS wurde bisher noch kein pSOS+-Fehler festgestellt, ebenfalls noch keine Diskrepanz zwischen der Beschreibung im Users Manual und dem realen Verhalten.

- pSOS+ ist ein sehr schnelles System. Ob es, wie vom Hersteller behauptet, 2 bis 5 mal schneller als alle vergleichbaren Konkurrenzsysteme ist, sei dahingestellt. Eigene vergleichende Messungen zwischen pSOS[2] und dem hier populäreren System OS-9 haben diese Faktoren aber bestätigt.

- pSOS+ hat einen recht kleinen Codeumfang (rund 15 Kilobyte), und der Kern ist nicht an eine bestimmte Hardware angepasst. Die Hardwareanpassung wird über eine einfache Konfigurationstabelle durchgeführt. Treiber für I/O-Bausteine müssen entsprechend der Standard-pSOS+-Schnittstelle selbst geschrieben werden. Die Eigenschaft, den Betriebssystemkern ohne Probleme bei GSI selbst an die eigene (sich zukünftig auch ändernde) Hardware anpassen zu können, ist eine unabdingbare Notwendigkeit.

Eine Beschreibung der Strukturen, Abläufe und Systemaufrufe von pSOS+ findet sich in [13].

2.2 Der System Debugger pROBE+

Für pSOS+ ist eine systemnahe Entwicklungsumgebung verfügbar. Der System Debugger und Monitor pROBE+ bietet zusätzlich zu der üblichen Funktionalität eines Debuggers noch die Möglichkeit Multitasking-, Speicherverwaltungs-, Messagequeue- und weitere Zustände von pSOS+ zu analysieren und auch zu verändern. pROBE+ hat die Möglichkeit des *Remote Debugging*. Diese erlaubt es, anstatt mit einem Terminal, mit einem Hochsprachen-Debugger (wie z. B. dem XDB, siehe Abschnitt 16.3) zu kommunizieren.

2.3 Die Runtime Library pREPC+

Die C Runtime Library pREPC+ ist eine mit dem ANSI Standard X3J11 kompatible Software. Sie wird wie pSOS+ und pROBE+ von Software Components Group Inc. als Binärcode (Motorola S-Record Format) geliefert. Sie enthält 91 Funktionen (zum Teil so C-typische wie `printf`, `scanf`) die sich in I/O-Funktionen, Utility-Funktionen, Character-Funktionen, String-Funktionen und eine Reihe weiterer allgemeiner Funktionen unterteilen. Alle Funktionen der pREPC+ sind für ein Multitaskingsystem ausgelegt, das heißt, sie sind voll re-entrant-fähig.

[2] Diese Messungen wurden im Jahre 1986 noch auf Basis des ersten an GSI gelieferten pSOS und dem damals aktuellen OS-9 gemacht.

pREPC+ kann auch mit dem Pascal-Compiler von Organon benutzt werden. Im MOPS ist pREPC+ Teil der Systemsoftware (wie pSOS+ selbst). Sie kann über Sprungtabellen erreicht werden. Ein Interface zwischen Compiler und pREPC+ (LJM_68.* auf ORGA$LIB) wurde von CAD-UL mitgeliefert, liegt im Quellcode vor und ist ebenfalls Teil der Systemsoftware.
Mehr zu pREPC+ findet sich in [11].

2.4 Die Implementierungssprache von MOPS

MOPS ist in Pascal geschrieben. Nicht nur der Code der Tasks, auch die Interrupt Service Routinen sind in Pascal implementiert. Nur einige wenige kleine Routinchen mussten in Assembler geschrieben werden.
Dass MOPS in einer höheren Sprache und nicht in Assembler implementiert wurde, bedarf wohl keiner Begründung. Diese höhere Sprache sollte eine neuere, *moderne* Sprache sein, mit Blockstrukturen und Typenüberprüfung (type checking), mit weitgehend selbsterklärendem Code und daraus folgender leichter Wartbarkeit, kurz, mit allem, was die letzten 10 bis 15 Jahre an Fortschritten auf diesem Gebiet gebracht haben. MODULA-2 ist die Sprache, die diesen Anforderungen am nächsten kommt (das Kontrollsystem auf den Alphas ist in MODULA-2 geschrieben, mit sehr guten Erfahrungen). Leider war kein passender MODULA-2 Compiler verfügbar (keine Anpassung an einen Realtime-Betriebssystemkern, schlechte Debug-Möglichkeiten). ADA ist, wenn man es mit MODULA-2 oder Pascal vergleicht, viel zu kompliziert und schwer verständlich, außerdem ist es noch schlechter verfügbar als MODULA-2. Die Sprache C bietet die Möglichkeit, noch unverständlicher als in Assembler zu programmieren, was angesichts der doch recht großen Wahrscheinlichkeit, den Code anderer Leute übernehmen zu müssen, den Einsatz dieser Sprache ausschließt. Pascal blieb da als zweitbeste Lösung übrig.

2.5 Compiler, Assembler, Linker

Bereits Anfang des Jahres 1992 hatte sich deutlich herausgestellt, dass das Entwicklungssystem der Firma Oregon zwei entscheidende Nachteile hat:

1. Kein HLL-Debugging. Mit der Weiterentwicklung des Kontrollsystems und dem Anwachsen der *Lines of Code* wurde es immer wichtiger, die Debug-Zeiten zu minimieren. Das Debuggen von Pascal-Programmen auf der Assembler-Ebene mit Hilfe von pROBE erforderte einiges an Aufwand. So musste man immer drei Listings zur Hand haben (Pascal-, Assembler- und Map-Listing), musste wissen, wie der Compiler Hochsprache in Assembler übersetzt, wo Variablen abgelegt sind usw. Dies erforderte viel Zeit und einiges – für das eigentliche Problem unnötige – Detailwissen.

2. Seit Ende 1991 gab es für dieses Entwicklungssystem keinen Support mehr. Das war sehr unbefriedigend, da der Compiler immer noch Fehler enthielt, um die wir nach wie vor „herumprogrammieren“ mussten, ganz abgesehen davon, dass man den Anschluss an sinnvolle und nützliche Weiterentwicklungen, die der Markt bietet, verlor (z. B. die Weiterentwicklung von pSOS -> pSOS+).

Aus diesen Gründen wurde Ende 1992 ein neues Entwicklungssystem beschafft. *Organon* von der Firma CAD-UL ist ein integriertes System, das sowohl einen Compiler mit den zugehörigen Tools als auch einen Hochsprachen-Debugger enthält. Neben der Möglichkeit des Debuggens in Hochsprache und der Anpassung an das neueste pSOS+, hat dieses System zwei wesentliche Vorteile:

1. Der Compiler versteht den Slang des Oregon Pascal-Compilers. Dies ist ein riesen Vorteil, da bei einer Umstellung der Software von Oregon auf Organon im besten Falle nicht eine Zeile Code geändert werden muss. Jede Änderung einer bestehenden Software birgt die Gefahr in sich, dass während der Umstellung neue Fehler eingebaut werden. Der neue Compiler minimiert diese Gefahr wesentlich.

2. Der Hersteller des Compilers ist eine deutsche Firma in Ulm. Dies gewährleistet einen sehr direkten Kontakt zu den *Entwicklern* des Systems und damit zu schnellen Antwortzeiten bei Problemen oder Fehlern im System.

Ein weiterer Vorteil, der allerdings im Moment von untergeordneter Bedeutung ist, ist die Kompatibilität zum C-Compiler der gleichen Firma. Falls es in Zukunft nötig wird auf die Sprache C umzusteigen, ist damit ein sanfter Übergang möglich. Die Compiler sollen sich nach Aussage von CAD-UL nur in ihrer äußeren Schale, dem Parser, unterscheiden. Alle darunterliegenden Ebenen seien gleich. Damit kann man von gleicher Datenrepräsentation (zumindest der einfachen Datentypen), gleichem Alignment (?) und gleichen Parameter-Übergabemechanismen der beiden Compiler ausgehen. Damit sollten in C geschriebene Module mit Modulen, die in Pascal geschrieben wurden kompatibel sein.
Eine Beschreibung der Werkzeuge von Organon befindet sich in Abschnitt 16.2 in diesem Dokument.
Eine genaue Beschreibung der Werkzeuge von Oregon befinden sich in der 1. Auflage des technischen Handbuchs vom August 1990 ([5]).

3 Grundlagen

3.1 Die Hardware-Anpassung von pSOS+, pROBE+ und pREPC+

Wie für pSOS+ so ist auch für den System Debugger pROBE+ und die Runtime Library pREPC+ die Anpassung an die Hardware selbst zu erstellen. Die Anpassungsstrukturen sind die gleichen. Für jedes Modul müssen die Konfigurationstabellen ausgefüllt und – wo nötig – die Treiberprozeduren geschrieben werden. Die Konfigurationstabelle für pSOS+ ist in [13] beschrieben. Die Treiberroutinen für den I/O-Baustein sind entsprechend den Spezifikationen in [13] in I/O-Jumptables eingebunden. Nur der serielle Port 1 wird von pSOS+ (als Device 0) mit diesen Routinen bedient. Der I/O-Baustein wird im Interruptmodus betrieben, um ein reibungsloses Multitasking zu gewährleisten. Die genaue Funktionsweise des pSOS+ Terminal-Treibers ist im folgenden Abschnitt 3.2 beschrieben.
Eine Beschreibung der Konfigurationstabelle für pROBE+ findet sich in [12]. Für die Ein- und Ausgabe von pROBE+ sind die Routinen mit den in [12] vorgegebenen Parametern geschrieben worden. pROBE+ und pSOS+ benutzen für ihre Ein- und Ausgaben den gleichen I/O-Baustein mit dem damit verbundenen Terminal, die pROBE+-Routinen pollen aber im Gegensatz zu den pSOS+-Routinen. Die pROBE+-Routinen bedienen beide seriellen Ports, da der zweite Port, die Host-Schnittstelle, für das Downline Load von Programmcode von der Alpha zum GµP, als auch zur Kommunikation mit dem XDB benutzt wird.
Eine Beschreibung der Konfigurationstabelle für pREPC+ findet sich in [11].
Die einzelnen Tabellen sind über den *System Anchor* und die *Node Configuration Table* verknüpft. Eine Beschreibung dieser Komponenten befindet sich ebenfalls in [13].
Einen angenehmen Vorteil bietet die Tatsache, dass alle drei Module nicht nur im Binärcode, sondern auch in einem Pseudo-Assemblercode (jeweils das gesamte Modul mit DC.B definiert) vorliegen. Dies bietet die Möglichkeit, die Module zu assemblieren[3], so dass sie beim Erstellen des Systems mit gelinkt werden können. Davon machen MOPS[4] als auch ECM und CP Gebrauch.
Für die Zeitverwaltung in pSOS+ wird der Timer des Bausteins MK68230 (MP1001) bzw. DP8570 (FIC8230) verwendet. Dieser generiert alle 20ms bzw. alle 10ms einen Interrupt. Die damit verknüpfte Interrupt Service Routine ruft den pSOS+ System Service tm_tick (Announce Tick) auf, der die interne Zeitverwaltung von pSOS+ bedient. Zusätzlich übernimmt sie die Zeitverwaltung der periodischen Queue. In Abschnitt 8.1 auf Seite 34 wird darauf näher eingegangen.

3.2 pSOS+ Terminal-Treiber

Alle Tasks des MOPS haben im Prinzip die Möglichkeit auf den Bildschirm zu schreiben. Zur Zeit liest nur die Task SHOW von der Tastatur. Die Einheit, die bei Ein- bzw. Ausgabe nicht unterbrochen werden darf, ist der String. Weitere Tasks, die zur gleichen Zeit einen String einlesen oder ausgeben wollen, müssen warten, bis der aktuelle String vollständig bearbeitet ist.

[3] siehe XLIB$DIR:MAKE$PSOS.COM
[4] siehe z. B. MOPS$DIR:MAKE$SYSTEM.COM

Das Terminal muss also verwaltet werden, so dass immer nur eine Task zu einer Zeit diese Resource benutzt. Die Verwaltung muss zudem so geschehen, dass das Multitasking nicht behindert wird, andere Tasks also weiterlaufen können, wenn eine oder mehrere Tasks auf die Resource Terminal warten.

Ein- und Ausgabe am Terminal ist ein sehr langsamer Prozess im Vergleich zur Geschwindigkeit der CPU. Nach der Ausgabe eines Zeichens muss lange gewartet werden, bis das nächste Zeichen ausgegeben werden kann. Für die Eingabe gilt das in noch viel extremerem Ausmaß. Dieses Warten darf in einem Multitaskingsystem nicht durch pollen geschehen (zum Beispiel durch ständiges Abfragen auf »TX-Buffer empty?« in einer Schleife), da dadurch die Resource CPU durch nutzloses Warten verschwendet wird und sie nicht mehr für andere Tasks zur Verfügung steht.

Aus diesen Gründen wurden die pSOS+-Terminaltreiber als interruptgetriebene Prozeduren mit Resourcenverwaltung durch Semaphoren realisiert. Am Beispiel einer Bildschirmausgabe soll die Funktionsweise kurz erklärt werden.

Im Device Init (`pSOS_de_init`) werden zwei Semaphoren kreiert und ihr Initwert gesetzt. `WREQ = 1` »Write Request« zur Verwaltung der Resource Bildschirm mit dem Initwert „Bildschirm ist frei“ und `WRDY = 0` »Write Ready« für die Fertigmeldung der Interrupt Service Routine mit dem Initwert „ISR ist nicht fertig“.

Möchte eine Task einen String am Bildschirm ausgeben, so wird in `pSOS_de_write` zunächst die Semphore »Write Request« angefordert, `P(WREQ)`. Ist sie bereits vergeben, der Bildschirm also an eine andere Task vergeben, wird die anfordernde Task in den Zustand `blocked` gesetzt, bis die Semaphore und somit der Bildschirm erhältlich ist. Warten bereits Tasks auf den Bildschirm, so wird die neue Task in eine FIFO-Queue eingereiht. Sie erhält den Bildschirm, wenn die Tasks vor ihr ihre Ausgabe beendet haben.

Ist der Bildschirm frei, erhält die anfordernde Task die Semaphore und nachfolgende Tasks müssen auf die Freigabe der Semaphore warten.

Anschließend wird der Interruptmode eingeschaltet. Das heißt, jedesmal, wenn ein Zeichen ausgegeben wurde, wird ein Interrupt generiert. Um dies zu Starten wird das Zeichen `NUL` ausgegeben. Die Ausgabe des eigentlichen Strings übernimmt die ISR `isr_write`. `pSOS_de_write` wartet auf die Fertigmeldung der ISR indem es die Semaphore »Write Ready« anfordert, `P(WRDY)`.

Ist das Zeichen `NUL`, das am Bildschirm nichts bewirkt, oder das zuletzt bearbeitet Zeichen des Strings ausgegeben, wird der Interrupt »TX-Buffer empty« generiert und die ISR `isr_write` (erneut) aufgerufen. Diese gibt das nächste Zeichen aus und ist damit beendet bis zum nächsten Interrupt. Ist der String komplett ausgegeben, schaltet die ISR den Interruptmode aus und signalisiert das sie fertig ist, indem sie die Semaphore »Write Ready« freigibt, `V(WRDY)`.

Hat `pSOS_de_write` die Semaphore »Write Ready« erhalten, wird die Semaphore »Write Request« wieder freigegeben, `V(WREQ)`. Damit ist der Bildschirm frei für die nächste Task.

In ganz ähnlicher Weise funktioniert die Tastaureingabe mit Hilfe der Semaphoren `RREQ` und `RRDY`.

Ein Problem entsteht dadurch, dass sich `pROBE+` und `pSOS+` die Hardwareschnittstelle zur Ein- und Ausgabe teilen. Um zu vermeiden, dass sich beide gegenseitig Zeichen „wegnehmen“, besonders bei der Eingabe, wurden Flags implementiert, die das verhindern sollen. Es ist fraglich, ob diese Implementierung völlig sicher ist. Bisher wurden allerdings keine Probleme festgestellt.

Die Menübedienung der Task SHOW erfolgt nicht über das `pSOS_de_read`. Sie macht sich den Poll-Mechanismus von `pROBE+` zunutze. Dies ermöglicht die ständige Auffrischung des Bildschirms und die gleichzeitige, asynchrone Eingabe eines Zeichens. Mit `READLN` und `pSOS_de_read` ist das nicht möglich. Alle anderen Eingaben erfolgen via `READLN`, also interruptgetriebenem Input.

3.3 Unterschiede zwischen pSOS und pSOS+

Das neue `pSOS+` unterscheidet sich von `pSOS` in vielen Dingen. Diese Unterschiede wirkten sich, mit wenigen Ausnahmen, nur gering auf die Umstellung des MOPS von `pSOS` auf `pSOS+` aus und sollen hier nicht dargestellt werden.

An dieser Stelle soll nur auf die wenigen Ausnahmen eingegangen werden, die es erforderlich machten, eine wesentliche Anpassung der Struktur des MOPS vorzunehmen.

3.3.1 Löschen einer Task

Altes pSOS: Hier musste sich der Benutzer (Process) nicht darum kümmern, dass allokierte Resourcen (Memory, Special Messages) vor dem Löschen wieder an `pSOS` zurückgegeben wurden. Diese Aufgabe übernahm `pSOS` selbst (siehe [10], Abschnitt 2.6.11, Death of a Process). Damit war es für den Error Handler (EHDL) einfach möglich, einen fehlerhaften Dispatcher (DInn) zu löschen ohne sich um allokierte Resourcen kümmern zu müssen.

Neues pSOS+: Dieses unterstützt die automatische Deallokierung von Resourcen nicht mehr. Es liegt allein in den Händen des Benutzers (Task), allokierte Resourcen zurückzugeben (siehe [13], Abschnitt 2.4.7, Death of a Task). Dies hat weitgehende Auswirkungen auf das Error Handling.

(Hier soll die Beschreibung der neuen Methoden des Error Handlings hin! Zur Zeit (1.12.94) fehlt mir noch jede vernünftige Idee.)

3.4 Die Register A5 und A6

Für spezielle prozedur- bzw. modul-übergreifende Mechanismen, wie relative Adressierung von globalen Variablen oder Aufbau eines Procedure Stack Frame benutzt der Pascal-Compiler bestimmte Register. Dies muss beim Schreiben z. B. von Interrupt Service Routinen oder Assemblermodulen berücksichtigt werden, um zu vermeiden, dass diese Register überschrieben werden.

A5: Pascal benutzt als Basispointer für globale Variable das Register `A5`, d. h., globale Variable werden mit einem Offset zu `A5` referiert. Jede MOPS-Task ist im Sinne von Pascal ein eigenes, unabhängiges *PROGRAM*, hat eigene globale Variable und damit ein eigenes `A5`.

Der Bereich für die globalen Variablen der Task wird zur Linkzeit festgelegt und beim Start der Task bereitgestellt (allokiert). Die Allokierung des Memory für die globalen Variablen und die Zuweisung des Pointers darauf an Register `A5` geschieht in den Assemblermodulen `ROOTHEAD.ASM` bzw. `TASKHEAD.ASM`. Im Abschnitt on page 19 wird darauf genau eingegangen.

In diesem Zusammenhang verdienen System und User Service Routinen (XSRs), Bibliotheksroutinen als auch Interrupt Service Routinen (ISRs) eine besondere Beachtung.

XSRs sind Prozeduren, die von einem der n Dispatcher aufgerufen werden. Jeder Dispatcher ist eine eigenständige Task und arbeitet damit in seinem eigenen Kontext, mit seinem eigenen `A5`.

Bibliotheksroutinen können von jeder Task aus aufgerufen werden. Jede Task arbeitet in ihrem eigenen Kontext, mit ihrem eigenen `A5`.

Einer ISR fehlt zur Laufzeit ebenfalls eine fest zugeordnete Taskumgebung. Sie arbeitet im Kontext der gerade unterbrochenen `running` Task, findet also *deren* `A5` vor.

Allen ist gemeinsam, dass sie kein fest zugeordnetes Register `A5` haben, mit dem sie arbeiten könnten. Damit haben sie keine Möglichkeit, globale Variablen zu verwalten. Aus diesem Grund dürfen XSRs, Bibliotheksroutinen und ISRs *keine* globalen Variablen benutzen! Die einzige Ausnahme sind Variablen auf festen Adressen, die mit den Anweisungen `ORIGIN` oder `USE` definiert wurden.

A6: Jede Pascalroutine benutzt als Framepointer für lokale Variable und Prozedurparameter das Register `A6`. Ein Frame ist ein Bereich auf dem Stack, der die lokalen Variablen, die Prozedurparameter, die Rückkehradresse in die rufende Prozedur (*Procedure Return Address*) und den Pointer auf den Frame der rufenden Prozedur umfasst. Dieser Stack Frame wird zum Teil von der rufenden, zum Teil von der gerufenen Prozedur aufgebaut. Bedeutende Unterstützung beim Aufbau des Stack Frame bietet dabei der `LINK`-Befehl des Prozessors. Die einzelnen Schritte zum Aufbau eines Frames sind in Abbildung 3.4 auf Seite 15 dargestellt.

Die Legende ist wie folgt:

`(1)` Stack und Stack Frame vor der ersten Aktion

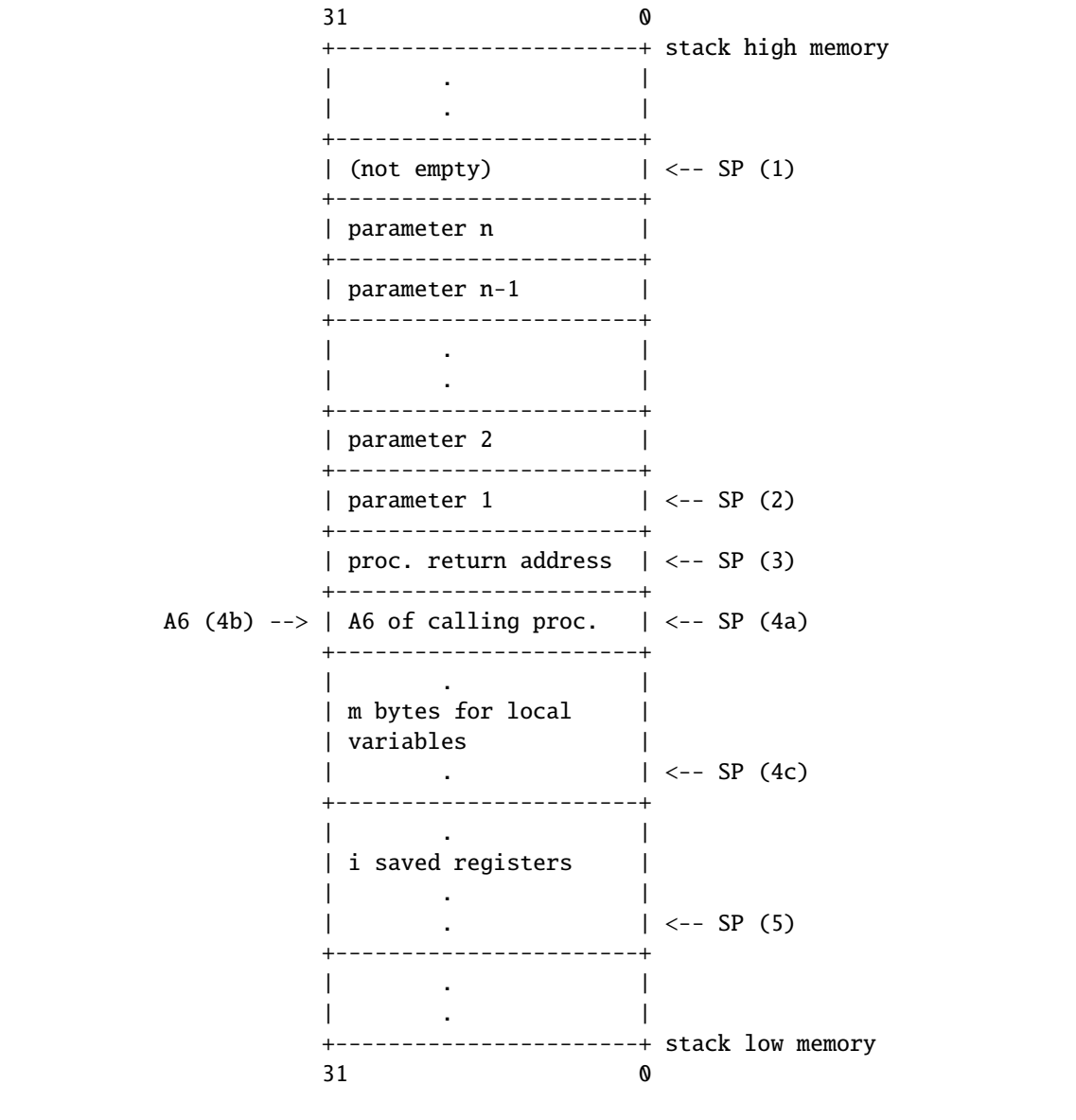

Abbildung 1: Aufbau eines *Procedure Stack Frame*

(2) nachdem n Prozedurparameter auf dem Stack abgelegt wurden

(3) nach dem Sprung (JSR) in die Unterroutine

(4) LINK A6,#<m>

  (4a) nach dem Link auf den Frame der rufenden Prozedur (pushed A6)

  (4b) nach dem Laden des Framepointers mit der Adresse des aktuellen Frames (SP -> A6)

  (4c) nach der Platzreservierung für die lokalen Variablen (SP + <m> -> SP)

(5) nachdem i Register gerettet wurden (MOVEM.L Di../Ai..,(SP))

3.5 Die Laufzeitbibliotheken

Der genaue Mechanismus der Generierung und des Aufrufs einer Laufzeitbibliothek des MOPS (siehe auch Abschnitt 1.5) sei hier erklärt.
Prinzipiell geht es darum, zwei zur Linkzeit unabhängige, ladbare Module (z. B. BIBLIOTHEK.SR und ANWENDUNG.SR) zu generieren, die dann zur Laufzeit doch voneinander abhängen, da die Anwendung Routinen der (Laufzeit-) Bibliothek aufruft.
Dies wird mit folgendem Mechanismus erreicht: Alle Routinen, die die Laufzeitbibliothek enthalten soll, werden in einer Datei zusammengefasst. Abbildung 3.5 auf Seite 16 zeigt die Struktur einer solchen Datei. Aus dieser Datei werden zwei Assemblerquelldateien, die Jump Table und die Entry Table, generiert.

```
*
* Entry points for pSOS interface
*
t_create_
t_ident_
t_start_
...
```

Abbildung 2: Datei für Jump und Entry Table

Die Jump Table (Abbildung 3.5 auf Seite 17) ist nichts anderes als eine Reihe aufeinanderfolgender Sprünge in die eigentlichen Routinen der Laufzeitbibliothek. Sie wird assembliert und mit den (Linkzeit-) Bibliotheken gelinkt, die die Routinen enthalten, die die Jump Table referenziert. Das Ergebnis des Links ist die fertige Laufzeitbibliothek (also BIBLIOTHEK.SR), die die Jump Table und die Bibliotheksroutinen enthält.
Die Entry Table (Abbildung 3.5 auf Seite 17) definiert die Routinen der Laufzeitbibliothek als Konstanten (EQU). Der Wert der Konstanten ist gleich der Adresse in der Jump Table an der der Sprung in die entsprechende Routine implementiert ist. Da jede Sprunganweisung (JMP xyz) 6 Bytes lang ist, unterscheidet sich jede Konstante von ihrer Vorgängerin um diese 6 Bytes.
Eine Anwendung (etwa die USRs), die Routinen der Laufzeitbibliothek benutzen möchte, wird mit dieser Entry Table gelinkt. Damit können alle Referenzen in die Laufzeitbibliothek aufgelöst werden. Im Extremfall muss keine weitere (Linkzeit-) Bibliothek dazugelinkt werden. Da die Entry Table die Routinen der Laufzeitbibliothek nur als Konstanten definiert (EQU), enthält die Anwendung selbst kein Byte Code der Bibliothek. Das Ergebnis dieses Links ist die fertige Anwendung (also ANWENDUNG.SR).
Die Verbindung zwischen Anwendung und Laufzeitbibliothek wird einzig und allein über die Adresse propjt[5] hergestellt. Die Laufzeitbibliothek, genauer gesagt die Jump Table, ist auf diese Adresse gelinkt,

[5]Der Name prop ist historisch bedingt (Oregon). Er wird weiterhin benutzt, weil die alten Tools zur Erzeugung von Jump und Entry Table, die propjt automatisch erzeugen, nach wie vor benutzt werden.

```
            SECTION 9
*
* Entry points for pSOS interface
*
            XREF    t_create_
            JMP.L   t_create_
            XREF    t_ident_
            JMP.L   t_ident_
            XREF    t_start_
            JMP.L   t_start_
            ...
```

Abbildung 3: Jump Table

```
*
* Entry points for pSOS interface
*
propjt      EQU     $1000
            XDEF    t_create_
t_create_   EQU     propjt
            XDEF    t_ident_
t_ident_    EQU     t_create_+6
            XDEF    t_start_
t_start_    EQU     t_ident_+6
            ...
```

Abbildung 4: Entry Table

sie beginnt an dieser Stelle. Die Entry Table definiert entsprechend die gleichnamigen Konstanten beginnend mit dieser Adresse und fortlaufend mit einem Offset von 6 Bytes.
Ruft eine Anwendung zum Beispiel die Routine `t_start_` auf, so sieht das in Assembler so aus, wenn `propjt`, wie oben, als `$1000` definiert ist:

```
JSR     $100C
```

Auf dieser Adresse befindet sich der dritte Sprung der Jump Table, da diese ab Adresse `$1000` gelinkt ist und jeder Sprung 6 Bytes lang ist. Das sieht so aus:

```
0000 100C   JMP     t_start_
```

Über diesen Sprung der Jump Table erreicht man endlich die eigentliche Routine `t_start_` in der Laufzeitbibliothek.
Die Tools zur Erzeugung von Jump und Entry Tables sind in Abschnitt 15.3 beschrieben.

3.6 Die Fähigkeit zum Re-Entry

In Multitasking-Systemen mit *pre-emptive Scheduling* ist die Eigenschaft von Prozeduren und Funktionen wiedereintrittsfähig oder *re-entrant* zu sein eine Grundvoraussetzung für die Funktionsfähigkeit des Systems.
Pre-emptive Scheduling heißt, dass eine laufende (`running`) Task an beliebiger Stelle von einer anderen Task unterbrochen werden kann. Die unterbrechende Task wird dann zur `running` Task.
Eine Routine ist re-entrant, wenn sie an beliebiger Stelle unterbrochen und *von neuem* aufgerufen werden kann, ohne dass bearbeitete Daten zerstört werden oder gar die Routine abstürtzt.
Ein Beispiel soll das verdeutlichen. Eine Task T1 hat die Funktion F aufgerufen. Diese wird mitten in der Bearbeitung der Daten D1 von der Task T2 unterbrochen. T2 ruft ebenfalls F auf, welche die Daten D2 bearbeitet und an T2 korrekt zurückliefert. Später, wenn T1 wieder aktiv ist, setzt F die Bearbeitung von D1 fort und liefert nach Beendigung die korrekten Daten D1 an T1 zurück. Dies ist möglich, wenn F re-entrant ist. Dargestellt ist die einfachste Verschachtelung. Auch kompliziertere, im Grunde beliebige, Verschachtelungen müssen möglich sein.
MOPS, das auf `pSOS+` basiert, besteht aus mehreren Tasks. Diese werden auf die beschriebene Art und Weise von `pSOS+` verwaltet. MOPS ist an beliebiger Stelle unterbrechbar.
Dies liegt allerdings nicht sofort auf der Hand, denn `pSOS+` arbeitet nicht im Time Slicing-Betrieb und kann somit nur ein Scheduling vornehmen, wenn ein Betriebssytemaufruf stattfindet. Trotzdem kann eine Task an beliebiger Stelle[6] unterbrochen werden. Jeder Interrupt unterbricht die gerade laufende Task sofort und führt in eine Interrupt Service Routine (ISR), die mit einem Systemaufruf `I_RETURN` beendet wird. Dieser Aufruf ermöglicht es `pSOS+` ein Scheduling durchzuführen und mit einer anderen als der unterbrochenen Task fortzufahren. Mindestens alle 10ms gibt es einen Timer Interrupt, der `pSOS+` über den Fortgang der Zeit informiert. MOPS ist also tatsächlich an beliebiger Stelle unterbrechbar.
Damit gilt für alle Routinen des MOPS, die von unterschiedlichen Stellen aus aufgerufen werden können, dass sie re-entrant sein müssen!
Das gilt zunächst einmal für alle Prozeduren und Funktionen der Laufzeitbibliotheken. Diese können von allen möglichen Stellen aus aufgerufen werden. Für die Teile der Pascal- und Systembibliothek garantieren deren Lieferanten. Die MOPS-Bibliothek ist entsprechend implementiert.
Ebenso müssen alle SSRs und USRs re-entarnt sein. Diese können von verschiedenen Dispatchern aus aufgerufen werden.
Und das gilt für die Dispatcher selbst, von denen es logisch zwar mehrere gibt, deren Code aber nur *einmal* im Speicher liegt. Alle anderen Task können zwar auch an beliebigen Stellen unterbrochen werden, können aber *nicht* von neuem aufgerufen werden. Sie setzen ihre Arbeit an der Stelle der Unterbrechung fort.
Besondere Vorkehrungen müssen getroffen werden bei speziellen Zugriffen auf globale Daten. So muss zum Beispiel die Bearbeitung von Listen oder Ringpuffern zu einem eindeutigen Ende geführt sein, bevor

[6] Ausgenommen sind solche Bereiche, in denen das pre-emptive Scheduling explizit (von der Anwendung) verboten wird.

eine andere Task eine neue Bearbeitung beginnt. Dies ist in MOPS an zwei Sellen notwendig, nämlich bei der Verwaltung der Sendepakete im Dualport-RAM des CP und bei der Verwaltung der Alarme von einem ECM in den Dualport-RAMs der ECs.
Die kritischen Bereiche werden mit Semaphoren geschützt. Eine Task muss vor dem Eintritt in den kritischen Bereich eine Semaphore anforden und am Ende des Bereichs wieder abgeben. Steht keine Semaphore bei der Anforderung zur Verfügung, wird die Task blockiert (`blocked`) und in eine Warteschlange (`Queue`) eingereiht, wo sie verbleibt, bis sie eine Semaphore zugeteilt bekommt.
Interrupt Service Routinen bedürfen keiner besonderen Beachtung was die Wiedereintrittsfähigkeit betrifft. ISRs können nur von ISRs höherer Priorität unterbrochen werden. Ist die ISR mit höherer Priorität beendet, fährt die unterbrochene ISR fort. ISRs laufen somit streng sequentiell und müssen deshalb nicht re-entrant sein.

4 Systemstart

ROOT ist die Basistask von MOPS. Sie wird beim Start des Gruppenmikro automatisch von `pSOS+` gestartet, fährt dann das ganze (MOPS-) System hoch, führt alle notwendigen Initialisierungen durch und geht dann in einen Ruhezustand (Endlosschleife), in der sie periodisch für ein Zeitintervall pausiert. Die Aktionen von ROOT im Einzelnen und in ihrer Reihenfolge sind im Folgenden beschrieben.

4.1 Der Header

Vor dem Starten der eigentlichen, in Pascal geschriebenen, Root-Task sind noch einige Dinge zu tun, die die Root-Task nicht übernehmen kann, bzw. deren Erledigung Voraussetzung sind für das fehlerfreie Laufen derselben. Dies übernimmt das in Assembler geschrieben Modul `ROOTHEAD`.
`ROOTHEAD` ist der Header der Root-Task. Dieses Modul wird von `pSOS+` aufgerufen, wenn es ROOT starten will.
Zunächst werden der Stack Pointer `SP` und der Frame Pointer `A6` terminiert. Damit ist unter anderem eine Kennung für den *Procedural Walkback* (siehe Abschnitt 13.5.3) möglich, an welcher Stelle das Walkback zu beenden ist.
Da ein Pascalprogramm für seine globalen Variablen einen entsprechend großen Speicherbereich und das Register `A5`, das darauf zeigt, erwartet, stellt `ROOTHEAD` dies zur Verfügung. Die Größe des Bereichs wird zur Linkzeit festgelegt (`gblmemstart`, `gblmemend`) und von `ROOTHEAD` allokiert (`rn_getseg`) und initialisiert (zu Null gesetzt).
Da die Pascal- und Mathematikbibliotheken ebenfalls globale Variablen haben, die sie (natürlich auch) über `A5` referieren, muss für diese zusätzlich ein Bereich von $\texttt{100}_{hex}$ Bytes pro Task reserviert werden. `ROOTHEAD` allokiert und initialisiert also insgesamt den Platz für die globalen Variablen der Task *plus* $\texttt{100}_{hex}$ Bytes für die Bibliotheken[7].
Der Zeiger auf diesen Bereich wird dem Register `A5` zugewiesen.
Der Bereich ist so aufgeteilt, dass in den ersten $\texttt{100}_{hex}$ ($\texttt{0}_{hex}$ bis $\texttt{FB}_{hex}$) die Variablen der Bibliotheken liegen und anschließend die der Task.
Eine Sonderstellung nimmt das Langwort auf der Adresse `A5` + $\texttt{FC}_{hex}$ ein. Es gehört nicht mehr zu den globalen Variablen der Laufzeitbibliothek. An dieser Stelle wird die Einsprungadresse für den Pascal-Teil der Task abgelegt. Das hat folgenden Grund. Der Start eines Pascalprogramms geschieht, indem nach `main_` (`JSR main_`) gesprungen wird. Die Prozedur `main_` ist ein Teil der Laufzeitbibliothek von Pascal. Sie nimmt einige pascal-spezifische Initialisierungen vor und springt dann nach `_MAIN__`. Dieses Label ist jetzt endlich die Startadresse der eigentlichen Applikation, des Teils der Task, der in Pascal geschrieben ist. Damit ergibt sich ein Problem. Die Prozedur `main_` befindet sich in der MOPSLIB. Da die MOPSLIB eine eigenständige Einheit ist, also unabhängig von einer Task gelinkt wird, muss zur Linkzeit von MOPSLIB die Adresse `_MAIN__` der Prozedur `main_` bekannt sein.

[7] Eine Schwierigkeit in Bezug auf globale Variablen der Bibliotheken ergibt sich im Zusammenhang mit XSRs, die ja im Kontext eines Dispatchers laufen aber getrennt generiert werden. Dazu mehr in Kapitel 15.5.

Jede Task, die gestartet wird ruft in ihrem Headerteil *genau dieses* `main_` in der MOPSLIB auf. `main_` müsste also für jede Task eine eigene Adresse `_MAIN__`, die Anfangsadresse der gerade startenden Task, kennen, was nicht geht.
Gelöst ist dieses Problem mit folgendem Trick. Jede Task hat ihren *eigenen* globalen Speicherbereich auf den `A5` zeigt. Jede Task legt in ihrem Headerteil (`ROOTHEAD` bzw. `TASKHEAD`) die Startadresse des Pascal-Teils, die ja bekannt ist, da Header und Pascal-Teil zusammengelinkt werden, auf der oben genannten Adresse `A5` + $\texttt{FC}_{hex}$ ab (zu `TASKHEAD` siehe auch Abschnitt on page 25).
Die Prozedur `main_` in MOPSLIB ist gegen die feste Adresse `_MAIN__` = `ROOTHEAD` + $\texttt{FA}_{hex}$ gelinkt. Dort stehen zwei Zeilen Assemblercode, die nichts anderes tun als die Startadresse der gerade zu startenden Task von `A5` + $\texttt{FC}_{hex}$ zu holen und dorthin zu springen. Das ist möglich, da jede Task ihr eigenes `A5` hat.
`main_` kehrt also für *jede* Task noch einmal nach `ROOTHEAD` zurück, von wo aus in den Pascal-Teil der Task gesprungen wird.
Für MOPSLIB ergibt sich daraus, dass zur Linkzeit nur die Startadresse von ROOT (genauer gesagt von `ROOTHEAD`) und der Offset zu den zwei Zeilen Assemblercode, die den eigentlichen Sprung in den Pascal-Teil der Task bewerkstelligen, bereits bekannt sind. Diese Adressen (`root_start` und `main_entry` = `root_start` + $\texttt{FA}_{hex}$) sind in `COM$CONSTANTS.PIN` definiert und werden von den entsprechenden Generierungsdateien benutzt.
In `ROOTHEAD.ASM` selbst finden sich ebenfalls einige Informationen zu seinen eben besprochenen Funktionen.

4.2 AC-Fail und Spurious Interrupt Handler initialisieren

AC-Fail und Spurious Interrupt Handler sind die beiden speziellen Errorhandler des MOPS. Eine ihrer Aufgaben ist das Zählen der entsprechenden Interrupts. ROOT setzt die zugehörigen Exceptionvektoren und initialisiert die Zähler.

4.3 Ethernet Controller testen

Zur Kommunikation mit anderen Knoten ist der CP unerlässlich. MOPS testet, ob ein Ethernet Controller vorhanden und ob es eine neue (FIO) oder eine alte Hardware (MP) ist. Alte und neue Hardware haben unterschiedliche Dualport-RAM-Adressen. Ist ein CP vorhanden, so allokiert MOPS den Bereich für die Sendepakete im Dualport-RAM des CP.

4.4 Restart Reason bestimmen

Der Neustart des MOPS kann verschiedene Gründe haben. So kann z. B. der Watchdog einen Reset ausgelöst haben, ein Benutzer kann die entsprechende Property aufgerufen haben und so weiter. MOPS bestimmt den Grund des Neustarts und trägt ihn zu Zwecken der Statistik in einen resetfesten Puffer ein. Nur bei einem Powerup wird dieser Puffer initialisiert.

4.5 Cache einschalten

Das Cache des 68020 ist bis zu diesem Zeitpunkt ausgeschaltet, um bei Fehlern in der Startphase bessere Debugmöglichkeiten zu haben. Hier wird es eingeschaltet.
Der 68000 hat kein Cache.

4.6 Lokale Datenbasis testen

MOPS testet, ob eine lokale Datenbasis vorhanden und in Ordnung ist. Dazu wird die Struktur der Datenbasis getestet. Ist diese nicht in Ordnung, wird ein Fehler ausgegeben, MOPS fährt aber trotzdem mit der Startphase fort.

4.7 Alarme initialisieren

Die Alarmpakete werden über ein `AlarmPointerArray` verwaltet. Dieses wird hier initialisiert.

4.8 CP-Interrupts scharfmachen

Hier geht es um die beiden Interrupts des CP, die dem MOPS geschickt werden, wenn ein neues Paket angekommen ist oder wenn das Senden eines Paketes beendet ist. Deren Vektoren werden gesetzt und die Interrupts scharfgemacht.

4.9 Semaphoren kreieren

Die beiden Semaphoren `SSND` und `SALR`, die den exclusiven Zugriff auf die Dualport-RAMs von CP und EC regeln (Routinen `Send_Packet` und `S_DevAlarm`) werden hier kreiert.

4.10 Messagequeues kreieren

Die MOPS-Tasks kommunizieren über den Messagequeue-Mechanismus von `pSOS+`. Da die Tasks die Messagequeues über eine Kennung (Ident) ansprechen, muss diese allgemein verfügbar sein. ROOT kreiert alle notwendigen Messagequeues (ihre Namen und Funktionen werden im Abschnitt 7 beschrieben), wobei ihre Idents im globalen Speicherbereich des MOPS (`LocalRAM`[8]) niedergelegt werden. Auf diesen Bereich können alle MOPS-Module zugreifen.

4.11 Partitions kreieren

Zur Zeit wird nur eine Partition kreiert. Sie wird für das Errorhandling benötigt, um Informationen über einen aufgetretenen Fehler vom Ort des Fehlers an EHDL zu melden. Die Größe einer Message reicht dafür (meistens) nicht aus.
Eine Partition ist nötig, weil nur von dieser, im Gegensatz zu Regions, auch von ISRs Puffer angefordert werden können.

4.12 Struktur der XCBs aufbauen

Die Verbindung zwischen MOPS und den Anwendungen wird über die XSR Control Blocks hergestellt. Diese enthalten unter anderem die Adressen der Init- und Run-Teile der XSRs, die MOPS nutzt, um die entsprechenden XSRs aufzurufen. Weiterhin sind enthalten die Nummer der XSR und des zugehörigen Gerätemodells, die maximal mögliche Anzahl der Daten im Antwortpaket, ein Zähler, der die Anzahl der Aufrufe der XSR zählt und ein Offset zum nächsten XCB. Die Struktur eines XCB ist in Abbildung 4.12 auf Seite 22 dargestellt.
Für jede XSR gibt es einen entsprechenden Kontrollblock. Da die XSRs der Repräsentationsebene anhand der Gerätemodellnummer und der XSR-Nummer aufgerufen werden, ist für eine einheitliche und schnelle Verwaltung der XCBs eine Kontrollblockstruktur namens `total_xcbs` implementiert. `total_xcbs` ist ein zweidimensionales Array von Kontrollblöcken, das eine Größe von `max_xcbs` * `max_xsr_classes` hat. Zur Zeit sind 20 Gerätemodelle und pro Gerätemodell 50 XSRs möglich. Diese Einschränkung ist sinnvoll, da pro Rahmen sowieso nur maximal neun SEs und damit zunächst neun Gerätemodelle möglich sind; weitere kommen durch die Computer (ECMS, EC) und eventuell durch Supergeräte hinzu. Sie ist aber auch sinnvoll, da bei 256 Gerätemodellen und 256 XSRs pro Gerätemodell ein Speicher von rund 1,5MB nötig wäre.
Durch diese Einschränkung bedingt ist es nicht möglich, direkt mit der Klasse (der Nummer des Gerätemodells) als ersten Index auf `total_xcbs` zuzugreifen. Die Klasse muss zunächst noch einmal mit `xcbindex`

[8] Das globale Memory heißt `LocalRAM`, weil es im lokalen RAM-Bereich des Gruppenmikros liegt, und weil mit *global* üblicherweise die A5-relativ adressierten globalen Variablen der Tasks gemeint sind.

```
XCB_Struct_Type = RECORD
    offset: uns_word;          { Offset to next XCB }
    xsr_class: uns_word;       { Eq-model number }
    xsr_num: uns_word;         { XSR number }
    xsr_ini_address: ADDRESS;  { Address of XSR init part }
    xsr_address: ADDRESS;      { Address of XSR run part }
    return_bc: uns_word;       { Maximum return bytecount }
    xsr_calls: uns_long;       { Number of calls to XSR }
    END;
```

Abbildung 5: Struktur eines XSR Control Blocks

indiziert werden. Die Nummern der XSRs sind beschränkt auf den Bereich 1 bis 50 und können direkt als zweiter Index benutzt werden. Ein Zugriff auf `total_xcbs` mit `xsr_class` und `xsr_nr` als Indizes sieht dann *vereinfacht* zum Beispiel so aus:

```
... total_xcbs[xcbindex[xsr_class], xsr_nr].xsr_address ...
```

Einige wenige SCBs liegen (physikalisch) in EPROM-Bausteinen. Der Rest der XCBs wird mit der Prozedur `add_usr` kreiert.
MOPS bietet die Möglichkeit neue oder geänderte XSRs ins RAM nachzuladen. Dazu wird an eine definierte Stelle im RAM ein Schlüsselwort und ein zusätzlicher SCB geladen. Wenn dieser Schlüssel existiert, werden von ROOT die zusätzlichen XCBs in `total_xcbs` kreiert. Wenn ein XCB in `total_xcbs` schon existiert, dann wird der existierende XCB mit dem neuen überschrieben und damit die neue XSR anstelle der alten als gültig implementiert.
Das Füllen der Struktur `total_xcbs` erfolgt durch ROOT bei jedem Start oder Neustart von MOPS. Mit dem Download von neuen XSRs und einem folgenden Neustart bietet MOPS so eine sehr komfortable Möglichkeit, neue oder geänderte Anwendungsprogramme in das System zu integrieren und auszutesten.
Da in einem GuP meist nur ein kleiner Bruchteil der Kontrollblöcke in `total_xcbs` besetzt ist, wird vor der beschriebenen Füllung das gesamte `total_xcbs` mit dem SCB der SSR *Not Implemented* besetzt. Der Aufruf einer nicht existenten USR oder SSR führt deswegen immer zum Aufruf dieser SSR.
Der genaue Mechanismus der Anmeldung der XSRs bzw. des Füllens von `total_xcbs` ist im folgenden Abschnitt erklärt.

4.13 XSRs anmelden

Das Anmelden der XSRs bei MOPS, mit anderen Worten, das Füllen der `total_xcbs`-Struktur, ist eine etwas trickreiche Sache, die deshalb etwas näher erläutert werden soll. Angemerkt sei schon, dass alle XSRs – bis auf die drei `UserInis` selbst und die »`not implemented`«-SSR – mit Hilfe der `UserIni`-Prozeduren angemeldet werden, es also insgesamt nur 4 *explizit* kodierte SCBs gibt.

1. Abbildung 1 zeigt eine Übersicht zur Verdeutlichung der Aufrufe der einzelnen `UserInis`.

   Dabei sind `ii` und `nn` Gerätemodell*nummern* und `II` und `NN` die entsprechenden Gerätemodell*namen*. Für jeden MOPS gibt es im Grunde 3 `UserInis`. Das zum Anmelden der SSRs, das zum Anmelden der EPROM-USRs und das zum Anmelden der RAM-USRs (siehe unten).

2. In der Prozedur `Compose_XCBs` im Modul `ROOT.PAS` werden die beiden SCBs für die SSRs `UserIni` (implementiert in `USERINI$01$04$14.PAS`) und `NIM` (»not implemented« SSR, implementiert in `SYSTEM$SRS.PAS`) definiert und nach `total_xcbs` kopiert.

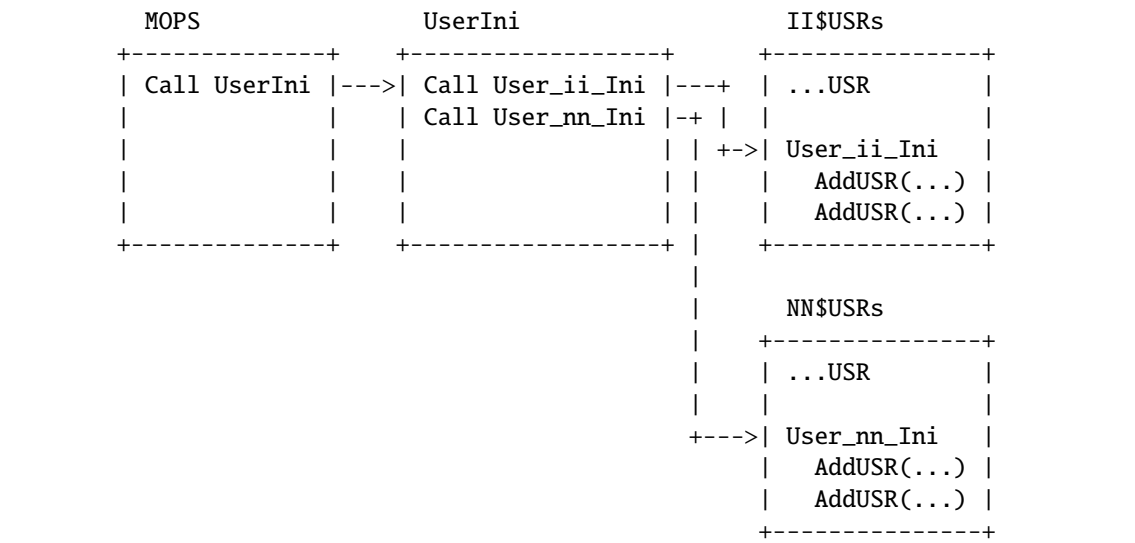

Abbildung 6: `UserIni`-Aufrufe

3. Im Modul `INITUSRS.ASM` ist als einziges der UCB für die USR `UserIni`(EPROM-Version der USRs) definiert. Dieser wird auf die Adresse `adr_ucbs` (n-fach definiert in `<cpuvers>$PAS$CONSTANTS.PIN`, `LINK$...$USRS_Vxx.COM`, `MAKE$USRS_Vxx.COM`, `GEN$USRS_Vxx.COM`!) gelinkt. Dabei geht `Compose_XCBs` davon aus, dass auf der Adresse tatsächlich ein SCB und kein Schrott steht, wenn

   ```
   - SCB.offset = xcb_offset,
   - 0 < SCB.xsr_class <= max_xsr_class_nr   und
   - 0 < SCB.xsr_num <= max_xcbs
   ```

   ist. Wird kein UCB gefunden, werden keine USRs angemeldet.

4. Im Modul `INITADDUSRS.ASM` ist als einziges der UCB für die USR `UserIni`(RAM-Version der USRs) definiert. Dieser wird auf die Adresse `adr_add_xcbs` (n-fach definiert in `<cpuvers>$PAS$CONSTANTS.PIN`, `LINK$...$USRS_Vxx.COM`, `MAKE$USRS_Vxx.COM`, `GEN$USRS_Vxx.COM`!) gelinkt. Weiterhin ist der `add_xcb_key` definiert, der vor den UCB gelinkt wird. Wird dieser Key von `Compose_XCBs` gefunden, geht es weiter wie unter Punkt 3 beschrieben. Ansonsten werden keine RAM-USRs angemeldet.

5. Nach den Punkten 2…4 stehen also die SCBs der SSRs `NIM`, `UserIni`(SSRs), `UserIni`(EPROM-USRs) und `UserIni`(RAM-USRs) in `total_xcbs`. Alle vier gehören zur XSR-Klasse 1 (`ssr_class`). Das ist das Gerätemodell des MOPS selbst (ECMS). Später im Startup von ROOT wird die Prozedur `start_init_ssrs` aufgerufen. Diese ruft den Initteil der `NIM`-SSR (z. Zt. keine Wirkung) und in einer Schleife alle weiteren in `total_xcbs` definierten SSRs auf. Dies sind allerdings bis jetzt nur die vier oben erwähnten SSRs. Da die SSRs in aufsteigender Reihenfolge (1…`max_xcbs`) aufgerufen werden, kommt als erstes in der Schleife die SSR `UserIni`(SSRs) dran (XSR-Klasse 1, XSR-Nummer 1.), die nichts anderes tut als das Anmelden der anderen SSRs, genauer gesagt das Komplettieren des `total_xcbs` mit den SCBs (siehe `UserIni` in `SYSTEM$SRS.PAS`). Nach der SSR `DBSLoad` (Klasse 1, Nummer 2) kommt als nächstes die SSR `UserIni`(EPROM-USRs) (Klasse 1, Nummer 3), die die EPROM-USRs anmeldet; also

das weitere Komplettieren des total_xcbs mit den UCBs. Danach kommt die SSR UserIni(RAM-USRs), die die RAM-USRs anmeldet.

Hiernach ist das total_xcbs komplett. Anschließend werden die Initteile der verbleibenden SSRs (5...max_xcbs) aufgerufen. Später im Startupprozess werden die Initteile der USRs im EPROM und anschließend die der USRs im RAM aufgerufen.

6. Zu beachten ist, dass die jeweiligen UserInis nur die gerätemodellspezifischen Aufrufe USER_<GM-Name>_INI enthalten. Für jedes zu linkende Gerätemodell (GM) gibt es in UserIni einen solchen Aufruf. Die einzelnen Prozeduren USER_<GM-Name>_INI sind in den jeweiligen Dateien <GM-Name>$USRS.PAS definiert. Diese rufen nun endgültig für jede anzumeldende USR die Prozedur AddUSR auf. AddUSR trägt nun endlich die Daten der USR in das total_xcbs an entsprechender Stelle ein (siehe Abbildung 1 auf Seite 23). UserIni selbst ist in einer eigenen Datei (USERINI$<GuP-Name>.PAS bzw. USERINI$<GM-Namen>.PAS) definiert. Diese wird normalerweise in den Link-Commandfiles automatisch generiert, übersetzt und zu den USRs dazugelinkt.

4.14 XSRs initialisieren

Eine XSR besteht aus zwei Teilen, einer Initialisierung-Prozedur und einer Laufzeit-Prozedur. Die Laufzeit-Prozedur ist der Teil, der von der VMS-Ebene als *Property* über die Userface-Schnittstelle aufgerufen oder mit einem Interrupt oder einer periodischen Triggerung konnektiert werden kann. Der Initialisierungsteil ist die Routine, die automatisch beim Hochfahren von MOPS abläuft. Damit ist dem Anwender die Möglichkeit gegeben, anwendungsbezogene Reset- oder Vorbereitungsoperationen auszuführen (MOPS selbst macht intensiven Gebrauch von dieser Möglichkeit, indem es periodische und interruptkonnektierte Aufrufe von SSRs in den Initialisierungsprozeduren selbst aufsetzt).
ROOT startet für alle *sinnvollen* XSRs, deren XCBs in total_xcbs stehen, ihren Initialisierungsteil. Die Adresse der Initialisierungsprozedur ist Bestandteil des Kontrollblocks. Sinnvoll heißt hier, alle XSRs außer der SSR *Not Implemented*, mit deren XCB ja alle nicht benutzten Blöcke von total_xcbs vorbesetzt sind. Da ROOT für alle XSRs die Initialisierungs-Prozedur aufruft, muss diese auch für alle XSRs als Code und als Eintrag in den zugehörigen XCBs existieren. Für XSRs, die von ihrer Anwendung her keinen Initialisierungs-Code benötigen, ist dies eine leere BEGIN..END-Prozedur.

4.15 Errorhandler starten

Der Errorhandler hat unter den MOPS-Tasks eine besondere Stellung, da alle Fehlermeldungen und die darauf folgenden Aktionen über ihn abgewickelt werden. Deshalb wird er als erste Task hochgefahren und mit der höchsten Priorität aller MOPS-Tasks versehen. Eine ausführliche Beschreibung des Errorhandlers folgt unter Abschnitt 13.

4.16 Tasks kreieren und starten

ROOT ist die Basis-Task (*parent task*) von MOPS, die von pSOS+ automatisch kreiert und gestartet wird. Sie kreiert und startet alle anderen Tasks (*offspring tasks*) des MOPS.
Das Kreieren einer Task wird durch den pSOS+ System Service t_create bewerkstelligt. Dieser macht die neue Task pSOS+ bekannt. Dabei wird der Name und die Priorität der Task festgelegt, ob sie mit oder ohne FPU läuft und aus dem Memory-Pool ein Segment für die Stacks allokiert.
Beim t_create einer Task wird diesem von pSOS+ ein Ident zugeteilt. Da sich Tasks über diesen Ident gegenseitig referieren, werden die Idents im globalen MOPS-Datenbereich abgelegt.
Gestartet wird die Task durch den pSOS+ System Service t_start. Dabei wird die Startadresse der Task bekanntgegeben und deren Mode festgelegt. Alle Tasks des MOPS sind preemptable, nehmen nicht am Timeslicing-Betrieb teil, haben keine ASRs und laufen im Supervisor-Mode. Zuletzt wird die Task den ready-Zustand gesetzt und nimmt dann, ihrer Priorität entsprechend, am Scheduling teil.

Beim Start der Task wird zunächst der Taskheader (TASKHEAD.ASM) durchlaufen. Er hat die gleichen Funktionen wie ROOTHEAD mit Ausnahme des Sprungs in den Pascal-Teil der Task. Dieser ist nur in ROOTHEAD.ASM implementiert (zu ROOTHEAD siehe Abschnitt 4.1).
Die Funktionalität der Tasks und ihre Stellung in MOPS werden in späteren Abschnitten beschrieben.

4.17 EC-Interrupts scharfmachen

Hier geht es um die drei Interrupts, die eine SE (EC) dem MOPS schicken kann. Dies sind der Alarm-Interrupt, der ECInfo-Interrupt und der Event-Interrupt. Für jede existierende SE werden deren Vektoren gesetzt und die Interrupts scharfgemacht.
Die Interrupts dürfen erst nach dem Start der Tasks scharfgemacht werden, da die entsprechenden Interrupt Service Routinen (pSOS+-) Events an die beteiligten Tasks schicken, wozu sie deren TIDs benötigen.

4.18 Main Loop

Die Schleife des Hauptprogramms (Main Loop) der Task ROOT hat nur eine Aufgabe. Sie triggert alle 10 Sekunden den Watchdog. Dazwischen pausiert sie für diese Zeit.

5 Ethernet-Kommunikation

Die höhere Ebene (VMS) und die mittlere Ebene (GuPs) des Kontrollsystems sind durch Ethernet verbunden. Die Kommunikation für die GuPs wird von einem eigenen Prozessor (Communication Processor, CP) auf einem separaten Board (Prozessorkarte der SEs mit einem Ethernet-spezifischen Piggy) abgewickelt. Auf der Ethernetkarte liegt ein Dualport-RAM, in dem die Empfangs- und die Sendepakete abgelegt werden. Vom GuP (MOPS) wird über den VME-Bus auf das Dualport-RAM zugegriffen. Das Dualport-RAM ist der Arbeitsspeicher, das heißt, die Pakete werden nicht in das lokale RAM des GuP kopiert. MOPS und die Anwendungen unter MOPS lesen aus den Empfangspaketen und schreiben in die Sendepakete im Dualport-RAM.
Die Speicherverwaltung des Dualport-RAMs ist zweigeteilt. Der CP verwaltet die Empfangspakete, MOPS verwaltet die Sendepakete. Verwalten heißt hier das Anfordern und Freigeben von Speicherplatz. Die Zweiteilung der Verwaltung ist eine Folge der asynchronen Zusammenarbeit des GuP mit dem CP. Dieser kopiert die empfangenen Pakete aus seinem lokalen RAM in den Empfangspaketeteil des Dualport-RAMs. Es ist naheliegend, Systemaufrufe von pSOS+ für die notwendige Speicherverwaltung zu nutzen. So fordert der CP über pSOS+ den notwendigen Speicherplatz für die Empfangspakete im Dualport-RAM an. Da pSOS+ kein Multiprozessorsystem (sondern ein Multitaskingsystem) ist, kann derselbe Speicherbereich nicht von zwei pSOS+-Systemen (CP-pSOS+ und MOPS-pSOS+) gleichzeitig verwaltet werden[9], das heißt, das CP-pSOS+ muss den Speicherplatz selbst wieder freigeben, wenn er nicht mehr benötigt wird. Analog ist die Situation für die Sendepakete. Durch Auswertung des Empfangspakets legt MOPS die Größe des zugehörigen Sendepakets fest. Den Speicherplatz für dieses Paket im Sendeteil des Dualport-RAMs fordert MOPS dann über *sein* pSOS+ an. Wenn das Sendepaket auf das Netz transferiert wurde, kann dessen Speicherplatz freigegeben werden. Dies muss aus den oben geschilderten Gründen von MOPS veranlasst werden.
Die Information über die Pakete (Anzahl, Adressen, Länge, Stand der Bearbeitung etc), die zwischen dem CP und MOPS ausgetauscht werden müssen, sind in einem Verwaltungsdatenbereich im Dualport-RAM abgelegt. Die Strukturen dieser Daten und der Algorithmus, mit dem von beiden Seiten auf diesen Daten gearbeitet wird, sind im folgenden beschrieben.

5.1 Die Verwaltungsstruktur der Empfangspakete

Die Verwaltungsstruktur der Empfangspakete (rcv_struct) ist als Ringpuffer organisiert. Ein Element des Ringpuffers enthält als einziges einen Zeiger auf ein Empfangspaket. (rcv_struct[i]). Die beiden

[9] Mit pSOS+m sollte das mittlerweile möglich sein

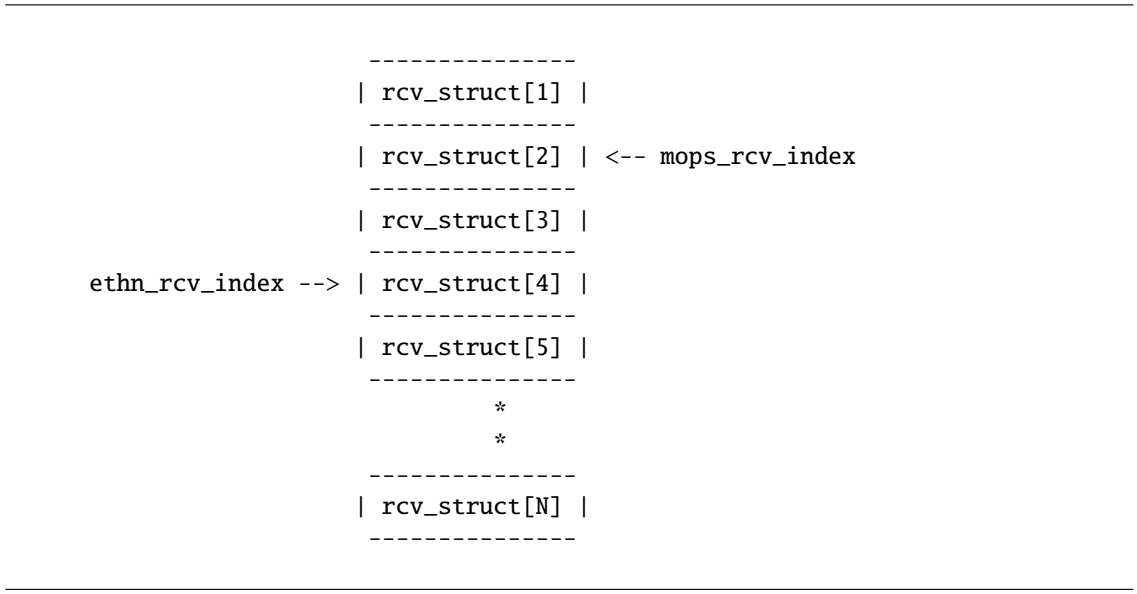

Abbildung 7: Die Verwaltungsstruktur der Empfangspakete

Indizes in Abbildung 5.1 auf Seite 26 haben folgende Bedeutung:

Der `mops_rcv_index` wird vom MOPS verwaltet. Er kennzeichnet das Empfangspaket, das von MOPS bearbeitet wird oder als nächsets bearbeitet werden soll.

Der `ethn_rcv_index` wird vom CP verwaltet. Er kennzeichnet den Zeiger, der auf das nächste, noch nicht empfangene Empfangspaket zeigen wird.

5.2 Die Verwaltungsstruktur der Sendepakete

Die Verwaltungsstruktur der Sendepakete (`snd_struct`) ist ebenfalls als Ringpuffer organisiert. Ein Element des Ringpuffers enthält einen Zeiger auf ein Sendepaket (`snd_pack`), die Größe des Sendepakets in Bytes (`snd_bc`) und einen Zeiger auf das zugehörige Empfangspaket (`rcv_pack`). Die drei Indizes in der Abbildung 5.2 haben folgende Bedeutung:

Der `mops_snd_index` wird vom MOPS verwaltet. Er kennzeichnet das Element, in das die Informationen über das nächste, noch nicht bereitgestellte Sendepaket eingetragen werden.

Der `ethn_snd_index` wird vom CP verwaltet. Er kennzeichnet das Element, das die Informationen über das nächste zu sendende Sendepaket enthält.

Der `snd_free_index` wird vom MOPS verwaltet. Er kennzeichnet das Element, das die Informationen über das nächste bereits gesendete Sendepaket enthält.

5.3 Multipakete

Die maximale Größe des Datenbereiches eines Ethernetpaketes beträgt 1450 Bytes. Dies reicht für die Kommunikation im Kontrollsystem nicht aus. So müssen zum Beispiel die SIS-Dipole mit zweimal 900 Stützpunkten versorgt werden. Dies ergibt bei einer 4 Bytes großen REAL-Zahl 7200 Bytes.
Aus diesem Grund wurden die Multipakete eingeführt. Damit sind Datengrößen bis zu 8700 Bytes möglich. Die notwendige Zerlegung in Teilpakete, die über Ethernet transportierbar sind, beim Sender und die Wiederzusammensetzung beim Empfänger sind für den Benutzer transparent. Auf der VME-Ebene ist dafür

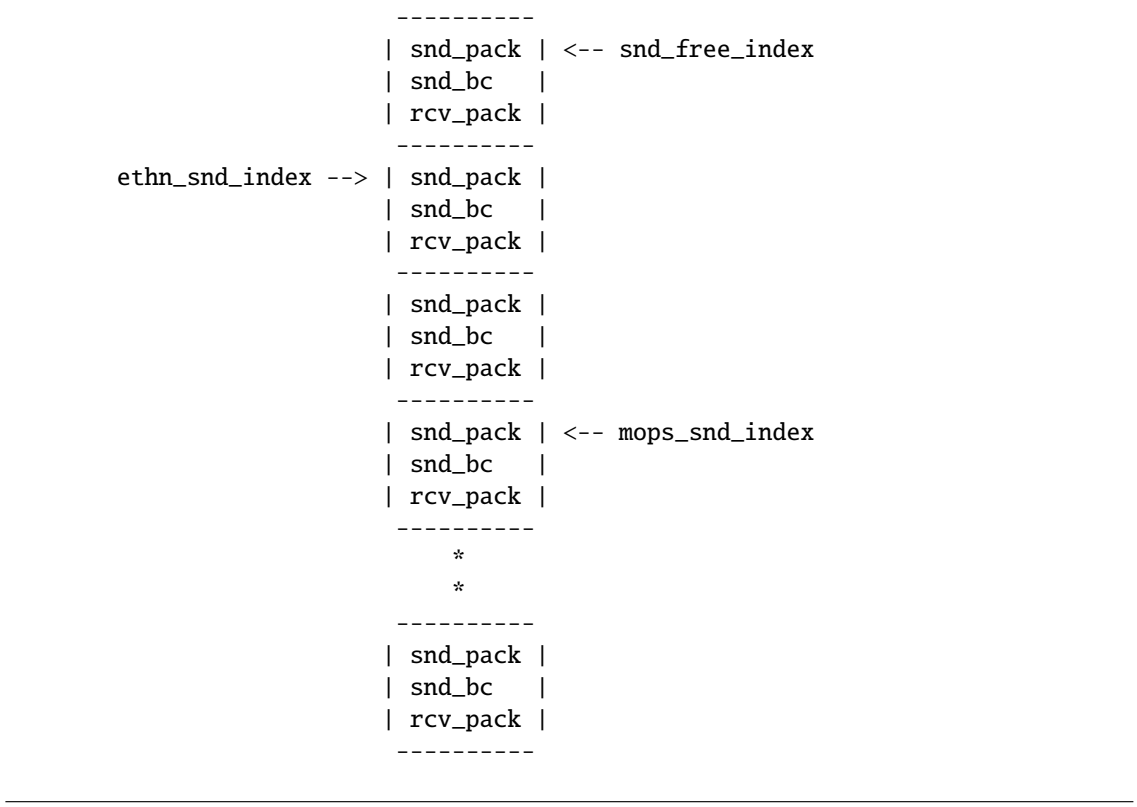

Abbildung 8: Die Verwaltungsstruktur der Sendepakete

alleine der Ethernet Controller verantwortlich. Für MOPS ist es damit möglich, Pakete mit *einem* Ethernet Header und einem Datenbereich bis zu 8700 Bytes zu handhaben.

5.4 Initialisierung der Ethernet-Kommunikation

Da bei der Kommunikationsverwaltung MOPS und der Communication Processor (CP) aktiv beteiligt sind, muss die Initialisierung der Verwaltungsstrukturen synchronisiert zweistufig abgewickelt werden. ROOT setzt sein Ready-Flag (`mops_rcv_ready`) für den CP auf `not ready`. Dann initialisiert es die Verwaltungsstruktur der Sendepakete (siehe Abbildung 5.2) und löst einen Reset für den CP aus, falls der Reset nicht vom CP selbst initiiert wurde. Der CP setzt sein Ready-Flag (`ethn_rcv_ready`) ebenfalls auf `not ready` und wartet darauf, dass MOPS seine Initialisierungsphase beendet (`mops_rcv_ready = ready`). Dann setzt der CP bei seiner Initialisierung die Zeiger auf die Verwaltungsstruktur der Empfangspaket entsprechend zurück und baut seine Knotenliste auf. Ist dies geschehen, setzt CP (`ethn_rcv_ready = ready`). Darauf hat MOPS gewartet und die Initialisierungsphase ist damit beendet.

5.5 Die Ethernetkommunikation

Die Verwaltungsalgorithmen setzen verschiedene Indizes für die Elemente der beiden Ringpuffer. Die Ungleichheit von Indizes veranlassen CP oder MOPS zu entsprechenden Aktionen. Die Gleichheit von Indizes bedeutet einen Ruhezustand (es sind keine Pakete empfangen worden, es müssen keine Pakete weggeschickt werden, es muss kein Speicherplatz freigegeben werden etc).
Im Ausgangszustand nach der Initialisierungsphase stehen alle fünf Indizes auf ihrem Anfangswert 1. Empfängt CP ein Paket, wird für dieses der entsprechende Speicher im Dualport-RAM allokiert, der Zeiger auf den Speicher in `rcv_struct[1]` abgelegt, `ethn_rcv_index` um eins erhöht und ein `recv`-Interrupt an MOPS geschickt.
Empfängt MOPS in der Task `RECV` einen `recv`-Interrupt, zeigt `mops_rcv_index` auf den Zeiger im Empfangsringpuffer, der auf das nächste, zu bearbeitende Paket zeigt. Dieser Zeiger wird mit weiteren Informationen in eine Message gepackt und an die Messagequeue `QRCV` geschickt. Die Dispatcher warten an dieser Queue auf Aufträge. MOPS bearbeitet solange Pakete, bis der `mops_rcv_index` wieder gleich dem `ethn_rcv_index` ist. Damit ist wieder der Ruhezustand eingekehrt, es sind keine weiteren Pakete mehr zu bearbeiten.
Mit dem Weiterschalten des `mops_rcv_index` ist das bearbeitete Element des Empfangsringpuffers bereits wieder frei für einen neuen Eintrag durch den CP. Das heißt allerdings nicht, dass das entsprechende Empfangspaket schon bearbeitet wäre. Das Paket und der Zeiger darauf wird jedoch von einem Dispatcher weiter verwaltet.
Empfängt ein Dispatcher eine Message aus der Messagequeue, legt er den darin befindlichen Zeiger auf das zu bearbeitende Empfangspaket in einer Variablen ab. Außerdem allokiert er im Dualport-RAM des CP Speicherplatz für das entsprechende Sendepaket. Auch der Zeiger auf das Sendepaket wird zwischengespeichert.
Erst wenn das Empfangspaket und das zugehörige Sendepaket vollständig bearbeitet sind, werden die Zeiger auf die beiden Pakete und die Größe des Sendepakets in das Element des Senderingpuffers geschrieben, auf das der `mops_snd_index` zeigt. Anschließend wird dieser um eins erhöht und ein `ethn`-Interrupt an den CP geschickt.
Der CP verschickt alle Sendepakete, die zwischen `mops_snd_index` und `ethn_snd_index` liegen und deallokiert die zugehörigen Empfangspakete. Dabei wird der `ethn_snd_index` solange erhöht, bis der `mops_snd_index` wieder erreicht ist. Der Ruhezustand ist hergestellt und CP schickt einen `xend`-Interrupt an MOPS.
MOPS kann nun alle Sendepakete, die zwischen `snd_free_index` und `ethn_snd_index` liegen, deallokieren. Dabei wird der `snd_free_index` solange erhöht, bis der `ethn_snd_index` wieder erreicht ist.
Das Zwischenspeichern der Zeiger auf das Empfangs- und das Sendepaket sowie die Übergabe des Zeigers auf das Empfangspaket an CP über den Senderingpuffer ist notwendig, weil MOPS ein Multitaskingsystem

ist. Damit ist es möglich, dass das Empfangspaket n eher vollständig bearbeitet ist als das Empfangspaket n-1. Somit muss auch das Sendepaket n *vor* dem Sendepaket n-1 verschickt werden.
Aus diesem Grund ist es zunächst unmöglich den Zeiger auf das Empfangspaket bis zur endgültigen Bearbeitung des Paketes im Empfangsringpuffer zu halten, da die Ringstruktur des Puffers nur eine streng sequenzielle Bearbeitung der Elemente zulässt. Die Bearbeitung eines Elementes n *vor* der Bearbeitung des Elementes n-1 ist *nicht* möglich.
Aus dem gleichen Grund ist es *nicht* möglich, den Zeiger auf das Sendepaket *vor* der vollständigen Bearbeitung beider Pakete im Senderingpuffer abzulegen.
Erst nach der vollständigen Bearbeitung beider Pakete (des Empfangs- und des zugehörigen Sendepakets) werden die Zeiger auf die Pakete in das nächste Element des Senderingpuffers durch den Dispatcher eingetragen. Der CP kann mit diesem Zeiger den Speicher des Empfangspakets deallokieren und MOPS den Speicher des Sendepakets, ohne dass „ältere“ Empfangs- und Sendepakete berührt würden und ohne dass die Verwaltung der Ringpuffer durcheinander käme. Vorraussetzung ist, dass diese Zuweisung von keinem anderen Dispatcher unterbrochen werden kann. Dies ist mit Hilfe der boolschen Semaphore `SSND` (in der Prozedur `send_packet`) realisiert und sichergestellt.

6 Die Dispatcher DInn

Die Dispatcher sind die Tasks, die für die Verwaltung, sowohl der synchronen als auch der asynchronen Aufträge (siehe Kapitel 7 bzw. 8) an die XSRs verantwortlich sind. Sie sind die Hauptprogramme, die die als Prozeduren implementierten XSRs aufrufen. Die Dispatcher sind in MOPS n-fach (üblicherweise 7 mal) implementiert, um die Anwendungen in einem Multitasking-Modus ablaufen lassen zu können. Geht zum Beispiel ein Dispatcher in den Zustand `blocked`, weil die aufgerufene XSR auf eine Resource oder ein externes Ereignis wartet, kann ein anderer Dispatcher derweil *seine* XSR aufrufen. Damit können im günstigsten Fall soviele XSRs quasi gleichzeitig laufen, wie Dispatcher vorhanden sind.
Die Dispatcher warten alle in einer Taskqueue an QRCV auf Messages. Wird eine Message nach QRCV geschrieben, dann erhält der erste Dispatcher in der Queue dieselbe, wird von `pSOS+` aus dem `blocked`-Zustand in den `ready`-Zustand gesetzt und wird `running`, wenn keine höherprioritäre Task `ready` ist. Der laufende DInn[10] entnimmt der Message die Adresse des Empfangspakets und beginnt mit dessen Bearbeitung.
Die Struktur eines Paketes ist in Abbildung 6 dargestellt. Diese ist gültig für Empfangs- und Sendepakete, mit der Einschränkung, dass Sendepakete keine Parameter enthalten können. Zu beachten ist ferner, dass alle Pakete mit einem sogenannten Endoffset terminiert sind. Dieser Endoffset ist im Prinzip der Offset der `XSR n+1` mit dem Wert Null.
DInn prüft anhand der Kennung in der Message (`pack_source`), woher das Empfangspaket kommt. Mit Hilfe dieser Kennung wird entschieden, ob das Paket konvertiert werden muss. Das ist notwendig, wenn das Paket über Ethernet von der Alpha kam, da die Repräsentation der Daten im Speicher auf VMS- und Motorola-Ebene unterschiedlich sind. Kam das Paket über Ethernet von der T85, ist keine Konvertierung notwendig, da diese die gleiche Datenrepräsentation wie Motorola hat. Hat das Paket die Kennung `int_connect_id` oder `mops_id`, dann ist die Quelle MOPS selbst, und es muss ebenfalls nicht konvertiert werden.
Alle Pakete müssen im Sinne von Ethernet Data Packets sein. Dies ist im Transport Header des Pakets gekennzeichnet.

```
Netman.PacketIdent = data_pc
Netman.PacketIdent = data_pc + sc_m68k
```

Der Subcode `sc_m68k` unterscheidet VMS-Pakete von T85-Paketen. Ist das Paket mit `error_pc` gekennzeichnet, dann ist dies die fehlerhafte Folge eines Ablaufs, die im Abschnitt 8 beschrieben wird.
Der nächste Schritt ist der Aufbau des Antwortpakets (des Sendepakets). Da in einem Paket die Aufrufe mehrerer XSRs enthalten sein können, läuft die folgende Aktion in einer Schleife ab.

[10] „nn“ steht für die Nummer des Dispatchers, derzeit „01“ bis „07“

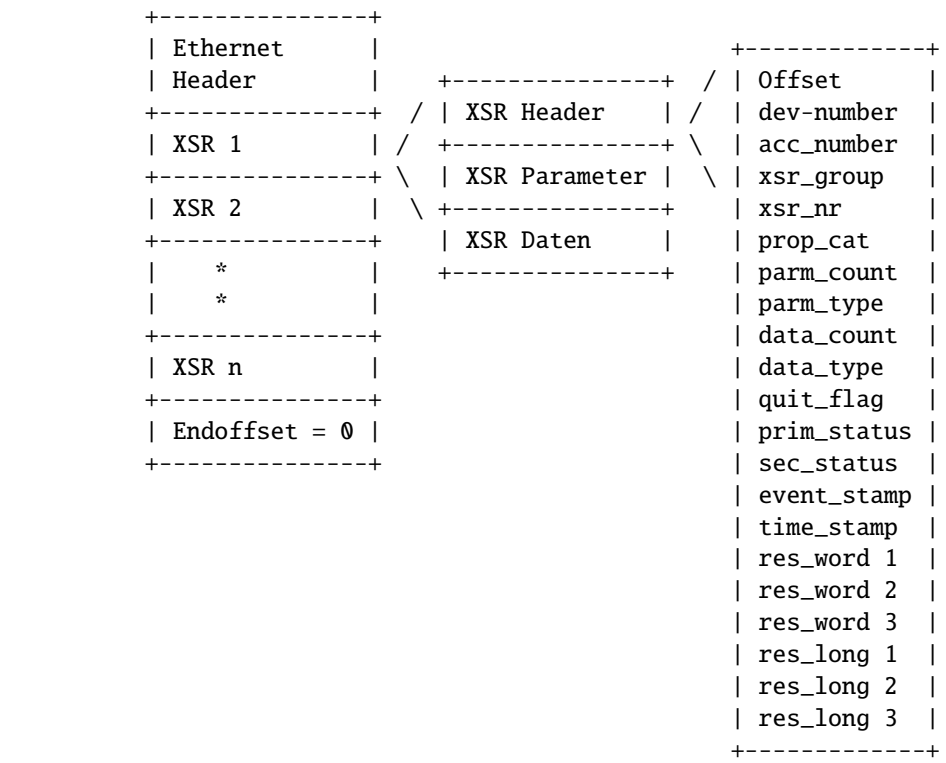

Abbildung 9: Ein Empfangs- bzw. Sendepaket

Für alle aufzurufenden XSRs entnimmt DInn der Kontrollblockstruktur `total_xcbs` die maximal zu erwartenden Bytecounts der Sendedaten, addiert diese und fügt für jede XSR die Länge eines XSR-Headers und einmal die Länge für den Ethernet-Header hinzu.
Ein Sendepaket dieser Länge wird von DInn aus dem Memory-Pool über pSOS+ allokiert, und der Ethernet-Header (unter Vertauschung der Quell- und Zieldaten) als auch der XSR-Header aus dem Empfangspaket in das Sendepaket übertragen.
Dann ruft DInn, über die in `total_xcbs` gespeicherte Adresse, die XSR auf. Beim Aufruf werden der XSR vorher berechnete Parameter übergeben: die Adressen der XSR-Header im Empfangs- und Sendepaket, die Adresse der Parameter im Empfangspaket und die Adresse der Daten (bei einer Lesefunktion liegen sie im Sendepaket, bei einer Schreibfunktion im Empfangspaket).
Nach Beendigung der XSR geht die Kontrolle an DInn zurück. Sind mehrere Aufträge im Empfangspaket spezifiziert, so ruft DInn die entsprechenden XSRs in einer Schleife nach obigem Muster auf. Alle Aufträge in einem Paket werden also von *einem* Dispatcher *streng sequentiell*, beginnend beim ersten Auftrag im Paket (`XSR 1`) und endend beim letzten Auftrag (`XSR n`), bearbeitet.
Nach der Terminierung der letzten XSR organisiert DInn die Übergabe des Sendepakets an den CP. Er trägt die Adresse des Sendepakets und die Adresse des zugehörigen Empfangspakets (falls vorhanden, es gibt unter Abschnitt 8 beschriebene Ausnahmen) in die Verwaltungsstruktur der Sendepakete ein, inkrementiert den `mops_snd_index` und löst einen Interrupt an den CP aus. Der CP transferiert das Sendepaket über Ethernet zu dem adressierten VMS-Knoten und gibt den Speicherplatz des Empfangspakets frei.
Führte die Prüfung von `pack_source` zu einer undefinierten Kennung, wird das Paket nicht weiter ausgewertet. Der Speicherplatz des Empfangspakets muss aber freigegeben werden. Deshalb trägt DInn die Adresse des undefinierten Empfangspakets in die Verwaltungsstruktur der Sendepakete ein und veranlasst den CP zur entsprechenden Bearbeitung. In diesem Fall wird die Verwaltungsstruktur der Sendepakete nur zur Freigabe des Speicherbereichs des Empfangspakets benutzt, da es kein Sendepaket gibt.
Damit ist für DInn die Bearbeitung dieses Paketes abgeschlossen. DInn wird über einen pSOS+ Service Call in die Queue der Dispatcher eingereiht, die an der Messagequeue QRCV auf neue Empfangspakete warten.

7 Synchroner Datenfluss

Mit dem Begriff *synchron* wird hier der einfache, klassische Ablauf einer Aktion im Kontrollsystem bezeichnet: Ein Anwenderprogramm auf VMS veranlasst über die Kontrollsystemschnittstelle Userface einen Gerätezugriff. Dieses generiert ein Ethernetpaket und schickt es an das entsprechende VME-System. Das Paket wird vom CP empfangen und an MOPS übergeben. MOPS bereitet das Antwortpaket vor und ruft die im Paket vorgegebene XSR auf. Die XSR verrichtet ihre Arbeit und schreibt die Ergebnisse in das Antwortpaket. Dieses ist damit komplett. MOPS übergibt das fertige Antwortpaket an den CP und dieser schickt es an die entsprechende Alpha zurück. Über die Kontrollsystemschnittstelle auf VMS erhält das Anwenderprogramm dann das Ergebnis. Die Komponenten von MOPS, die an diesem Ablauf beteiligt sind, werden im folgenden beschrieben.

7.1 Die Task RECV und die Messagequeue QRCV

Nach der Synchronisation mit dem CP, die in der Initialisierungsphase des Systems beim Hochfahren von MOPS erfolgt, wartet die Task RECV auf das Event `recv_event`. Hat der CP ein Paket empfangen und in die Verwaltungsstruktur für die Empfangspakete eingetragen, schickt er einen Interrupt an MOPS. Die Interrupt Service Routine `recv_isr` bearbeitet diesen Interrupt und schickt das `recv_event` an RECV. Bei Empfang dieses Events baut RECV für jedes Empfangspaket, das durch die Ungleichheit zwischen den beiden Indizes `ethn_rcv_index` und `mops_rcv_index` in der Verwaltungsstruktur der Empfangspakete definiert ist, eine Message auf, die als wesentlichen Inhalt die Quelle und die Adresse des Empfangspakets sowie das Ziel des Sendepakets hat. Diese Message schickt RECV an die Messagequeue QRCV[11].

[11] Die Namen aller Messagequeues in MOPS beginnen mit einem Q

QRCV ist der zentrale Eingang für alle Aufträge an MOPS. Die Verwalter der Aufträge, die im Abschnitt 6 beschriebenen Dispatcher DInn, warten an dieser Messagequeue auf Messages.
RECV inkrementiert für jedes bearbeitete Paket den Index `mops_rcv_index` und geht nach der Bearbeitung des letzten Pakets (bei Gleichheit der beiden genannten Indizes) wieder in seinen Wartezustand bis zum Empfang eines neuen `recv-event`.
Zur Verwaltung der Empfangspakete siehe Abschnitt 5.5 und Abbildung 5.1.

7.2 Die Task XEND

Die Notwendigkeit der Task XEND ist eine Folge der Verlagerung der Ethernet-Kommunikation auf den separaten Prozessor CP. Dieser kopiert das fertige Sendepaket in sein lokales RAM und transferiert es von da über seinen Ethernet-Baustein auf das Netz. Nach dem Kopieren in das lokale RAM kann der Speicherplatz für das Sendepaket freigegeben werden. Der Speicherplatz für die Sendepakete wird aber, wie im Abschnitt 5 beschrieben, von MOPS verwaltet. Der CP informiert MOPS, dass Sendepakete freigegeben werden können, indem er den `ethn_snd_index` in der Verwaltungsstruktur der Sendepakete inkrementiert und einen Interrupt auslöst, der die ISR `xend_isr` triggert. Die `xend_isr` schickt das `xmit_end_event` an XEND, das in einem `pSOS+` Service auf dieses Event wartet. Bei Empfang des Events vergleicht XEND die beiden Indizes `ethn_snd_index` und `snd_free_index` in der Verwaltungstruktur der Sendepakete und gibt den Speicherplatz für Sendepakete frei, die durch die Ungleichheit der beiden Indizes definiert werden. Bei jeder Freigabe inkrementiert XEND den `snd_free_index` und geht bei Gleichheit der beiden genannten Indizes wieder in seinen Wartezustand bis zum Empfang eines neuen `xmit_end_event`.
Zur Verwaltung der Sendepakete siehe Abschnitt 5.5 und Abbildung 5.2.

8 Asynchroner Datenfluss

Der *asynchrone* Datenfluss ist eine Erweiterung des einfachen Auftrag/Antwort-Schemas des synchronen Datenflusses. Asynchron heißt, dass eine Aktion in MOPS an ein äußeres Ereignis geknüpft ist. Nach jedem aufgetretenen Ereignis führt MOPS selbständig die Aktion aus, ohne dass dafür eigens ein Auftrag der VMS-Ebene nötig ist. Das Ereignis kann zum Beispiel der Ablauf eines Zeitintervalls oder die Nachricht über den Empfang eines bestimmten Events durch eine SE sein.
Diese Art des Datenflusses muss allerdings von einem Prozess auf der VMS-Ebene in Auftrag gegeben werden, damit klar ist, an welches Ereignis die Aktion geknüpft werden soll und wer der Empfänger des Ergebnisses der ausgeführten Aktion, also der gewonnenen Daten, ist. Das Herstellen einer Verbindung, mit anderen Worten, das Öffnen eines Kanals zwischen einer Datenquelle auf VME-Ebene und einer Datensenke auf der VMS-Ebene, sowie das Verknüpfen der Datenquelle mit einem bestimmten äußeren Ereignis wird *Konnektieren* genannt. Eine *Konnektierung* ist damit eine bestehende Verbindung zwischen Datenquelle und -Senke inklusive der Verknüpfung der Datenquelle mit einem äußeren Ereignis.
Mit dieser Methode ist es möglich, mit nur einem Auftrag beliebig viele Antworten zu erhalten.
Die asynchrone Aktion ist MOPS-intern gesehen nichts anderes als das Aufrufen einer XSR mit anschließendem Abschicken des von der XSR mit Daten gefüllten Antwortpakets an den konnektierten Prozess auf VMS.
MOPS unterscheidet zwei Arten von Konnektierungen. Zeit- oder periodische Konnektierungen sind an zeitliche Ereignisse geknüpft. Das Eintreten einer bestimmten Uhrzeit oder der Ablauf eines Zeitintervalls sind ein solches Ereignis und führen zum Aufruf der konnektierten XSR.
Interruptkonnektierungen sind Verknüpfungen an Ereignisse, die ihre Ursache auf einer SE haben und die mittels Interrupts an MOPS weitergeleitet werden. Die Änderung der Gerätekonfiguration auf einer SE oder deren Kaltstart sind ein solches Ereignis und führen zum Aufruf der konnektierten XSR.
Für die Behandlung periodischer Konnektierungen ist die Task PERI und die Messagequeue QPER zuständig, für die Behandlung von Interruptkonnektierungen die Task RUPT und die Messagequeue QRUP.

Ein wesentlicher Teil der Arbeit ist in die zwei SSRs Connect (S_Conn) und Disconnect (S_Dcon) verlagert, die hier auch beschrieben werden (siehe Abschnitte 11.2 ab Seite 38 und 11.3 ab Seite 42).

8.1 Die Task PERI und die Messagequeue QPER

Konnektierungen werden in Queues verwaltet. Diese Queues sind als einfach verzeigerte, lineare Listen organisiert, wie sie z. B. in [14] beschrieben sind. Die periodische Queue kann maximal max_peris Elemente enthalten. Das Memory für die Elemente wird in der Initialisierungsphase der Task PERI allokiert. Es findet also keine dynamische Speicherverwaltung im strengen Sinne statt.
Ein Queue-Element in der PERI-Queue hat die Recordstruktur, wie sie in Abbildung 8.1 dargestellt ist.

```
queue_t = RECORD
              delta:      uns_long;
              tot_delta:  uns_long;
              conn_count: word;
              ident:      uns_word;
              process:    PACKED ARRAY[1..6] OF CHAR;
              node:       uns_word;
              mess_add:   pack_pointer;
              next:       queue_p_t;
          END;
```

Abbildung 10: Struktur eines Elementes der Periodical Queue

Ein Element der PERI-Queue hat zwei Zeitparameter: delta und tot_delta.
delta definiert die Anzahl von Ticks, nach denen der in diesem Queue-Element definierte Auftrag ausgeführt wird[12] (ein Tick hat die Dauer von 10 bzw. 20 Millisekunden). Die Deltas werden additiv behandelt, d. h. ein Queue-Element steht dann zur Bearbeitung an, wenn die Summe aller Deltas der vor ihm in der Queue stehenden Elemente abgelaufen ist. delta ist (beim Eintrag in die Queue) eine variable Größe, die von den weiter vorn stehenden Deltas abhängt.
tot_delta (steht für totales Delta) hat pro Auftrag einen festen Wert. Es definiert die Wiederkehr (Periode) des Auftrags.
conn_count gibt an, wieviele mal der definierte Auftrag auszuführen ist.
ident ist der Packet Ident aus dem Ethernet-Header des Auftragspakets.
process ist der Name des VMS-Prozesses, der die Konnektierung aufgesetzt hat.
node ist die Nummer des VMS-Knotens, von dem der Konnektierauftrag kam.
mess_add enthält die Adresse des *kopierten* Auftragspakets.
next ist der Pointer zum nächsten Element in der Queue.
In seiner Initialisierungsphase baut PERI die Queue auf, in die die Konnektierungen später eingetragen werden. Diese Queue hat zwei Anker hat, die im globalen Datenbereich von MOPS liegen, free_an, der der Kopf aller freien, noch nicht benutzten Queue-Elemente ist, und used_an, der der Kopf aller belegten Queue-Elemente ist.

[12] Die von Anwenderprogramm über Userface bis zum MOPS durchgängige Einheit *Ticks* für diese Element hat sich spätestens zum Zeitpunkt der Einführung einer zweiten Hardware (FIC8236) als schlecht gewählt herausgestellt. Diese Hardware generiert im Gegensatz zur alten alle 10ms einen Tick. Anwender wissen aber nicht (und sollten es auch nicht wissen müssen), ob sie mit einem MOPS auf einer alten oder auf einer neuen Hardware sprechen. Für sie gilt immer 1 Tick gleich 20ms. Aber auch das sollten sie eigentlich nicht wissen müssen. Ein Anwender sollte einfach sagen können: „Periodischen Auftrag alle soundsoviel *Sekunden* ausführen." Die Umrechnung von Zeiten in Ticks sollte einzig und allein Sache des MOPS sein! Im Moment rechnet MOPS Ticks (20ms) in Ticks (10ms) um. Schön, gell?

Nach der Initialisierung wartet PERI auf eines von zwei möglichen Events. Auf `conn_event`, das von der SSR `S_Conn` kommt, oder auf `timer_event`, das die Timer-ISR `tim_isr` abschickt.
Bei Empfang von `conn_event` liest PERI seine Messagequeue QPER, die eine oder mehrere Konnektierungs- oder Diskonnektierungs-Messages enthält.
Bei einer Konnektierung trägt PERI die relevanten Daten aus der Message in ein Queue-Element ein. Das gefüllte Queue-Element wird aus der `free`-Queue in die `used`-Queue umgehängt. Hierbei wird durch Aufaddieren der Deltas der Queue-Elemente die Queue-Position für das neue Element bestimmt, das Queue-Element dort eingehängt und das `delta` des folgenden Elementes entsprechend vermindert. Da das Einordnen eines Queue-Elements nur einmal pro Aufruf des darin definierten Auftrags stattfindet, das Update der Queue aber bei jedem Tick, ist es aus Effektivitätsgründen sinnvoll, die Zeitbelastungen in der beschriebenen Weise zu verteilen. Das Einfügen und Umhängen von Queue-Elementen ist wie in [14] beschrieben implementiert.
Bei einer Diskonnektierungs-Message, also der Terminierung eines (hier periodischen) Auftrags, werden entsprechend dem Disconnect-Descriptor `dcon_desc`

- der periodische Auftrag eines (VMS-)Prozesses eines (VMS-)Knotens mit einer Kennung `ident` oder
- alle periodischen Aufträge eines (VMS-)Prozesses eines (VMS-)Knotens oder
- alle periodischen Aufträge eines (VMS-)Knotens

terminiert, d. h., die zugehörigen Queue-Elemente werden aus der `used`-Queue in die `free`-Queue umgehängt und der Speicherplatz für die Empfangspakete wird freigegeben.
Die Laufzeitverwaltung der periodischen Queue wird von der Timer-ISR mit übernommen. Diese ISR läuft bei jedem Tick an und dekrementiert das `delta` des ersten Queue-Elements um einen Tick. Hat `delta` den Wert Null erreicht, schickt die Timer-ISR das Event `timer_event` an PERI, um sie darüber zu informieren, dass ein periodischer Auftrag zur Bearbeitung ansteht.
Wie schon oben erwähnt, beruht der Algorithmus der Laufzeitverwaltung der Queue auf einer Addition der Zeit-Deltas der Queue-Elemente. Dies hat den großen Vorteil, dass die Timer-ISR nur mit dem ersten Element in der Queue arbeiten (das `delta` dekrementieren) muss. Damit ist die Timer-ISR, die alle 10ms bzw. 20ms läuft, am geringsten belastet und völlig unabhängig von der Anzahl der Elemente in der Queue.
Wurde PERI durch den Empfang eines `timer_event` aktiviert, ist ein periodischer Auftrag reif zur Abarbeitung. Die Timer-ISR hat das `delta` des ersten Queue-Elements bis auf 0 dekrementiert und daraufhin das `timer_event` an PERI geschickt. PERI baut ein Message-Paket an die Dispatcher mit Daten aus dem ersten Queue-Elements auf und schickt dieses an die Messagequeue QRCV. Ab hier wird dieser Auftrag dann wie ein direkter Auftrag eines VMS-Knotens von einem Dispatcher bearbeitet. Der `conn_count` im Queue-Element wird um 1 dekrementiert und das Queue-Element dann wieder zeitrichtig in die Queue einsortiert. Ist `conn_count` jetzt gleich null, dann war dies der letzte periodische Auftrag, den dieses Queue-Element auslösen sollte. Das Queue-Element kann aber wegen der Mechanismen der Speicherverwaltung für das Empfangspaket jetzt noch nicht gelöscht werden. Die Dispatcher, die bei der Rückkehr aus der aufgerufenen XSR den Speicher des Empfangspakets freigeben, tun dies nicht für periodische Aufträge, weil in diesem Fall das Empfangspaket für den nächsten periodischen Aufruf erhalten bleiben muss. PERI selbst kann das Empfangspaket auch nicht freigeben, weil beim jetzt erst folgenden letzten periodischen Aufruf die aufgerufene XSR noch mit den Daten und Parametern des Empfangspakets arbeitet. Nur der die XSR aufrufende Dispatcher weiß, wann die XSR beendet ist und das Empfangspaket freigegeben werden kann. Deswegen markiert PERI die Message an die Dispatcher, was den Dispatcher, der die Message dann bearbeitet, veranlasst, nach Ablauf der aufgerufenen XSR eine Diskonnektierungs-Message für diesen periodischen Auftrag an PERI zu schicken. Diese Message veranlasst bei PERI die oben beschriebene Diskonnektierungsaktionen (Löschen/Umhängen des Queue-Elements und Freigabe des Speicherplatzes für das Empfangspaket).
Konnektierungen, Diskonnektierungen und das Starten von periodischen Aufträgen können in einer Schleife ablaufen. Wenn bei einer Aktivierung von PERI mehrere Messages in seiner Messagequeue QPER stehen, werden die damit verbundenen Konnektierungs- und Diskonnektierungsaktionen solange nacheinan-

der ausgeführt, bis QPER leer ist. Wenn in der `used`-Queue nicht nur das erste Element ein `delta` von null hat, sondern auch noch das/die folgende(n), heißt das, dass die darin enthaltenen periodischen Aufträge gleichzeitig anstehen. PERI gibt sie nacheinander als Aufträge an die Dispatcher.
Nach der Beendigung der Auftragsstarts oder der (Dis-) Konnektierungen geht PERI wieder in den Event-Wartezustand.
Siehe auch die Beschreibungen der SSRs Connect auf Seite 38 und Disconnect auf Seite 42.

8.2 Die Task RUPT und die Messagequeue QRUP

Auch die Interruptqueue ist als eine, entsprechend [14], einfach verzeigerte, lineare Liste organisiert. Die Interruptqueue kann maximal `max_rupts` Elemente enthalten. Das Memory für die Elemente wird in der Initialisierungsphase der Task RUPT allokiert. Es findet also keine dynamische Speicherverwaltung im strengen Sinne statt.
Ein Element in der RUPT-Queue hat die Recordstruktur, wie sie in Abbildung 8.2 dargestellt ist.

```
ru_qu_t = RECORD
              se_int_nr: uns_word;
              int_desc:  uns_word;
              ident:     uns_word;
              process:   PACKED ARRAY[1..6] OF CHAR;
              node:      uns_word;
              mess_add:  pack_pointer;
              next:      ru_qu_p_t;
          END;
```

Abbildung 11: Struktur eines Elementes der Interrupt Queue

`se_int_nr` enthält (jeweils in einem Byte) die Nummer der SE, an die der Auftrag geknüpft ist (1 ... 9 für *eine* SE, 0 für alle SEs) und die SE-bezogene Nummer des Interrupts (0 für einen Alarm, 1 für eine ECM-Information, 2 für ein Event des Timingbusses).
`int_desc` ist die nähere Spezifizierung des Interrupts (Alarm: keine Spezifizierung; ECM-Info: 0 für Kaltstart, 1 für Konfigurationsänderung, 2 für Lebenszeichen, 6 für IPS-Interrupt; Event: das Eventbuswort).
Die restlichen Elemente im Record haben die gleiche Bedeutung wie die der PERI-Queue.
RUPT ist in seiner Funktionalität und seinen Abläufen der Task PERI so ähnlich, dass hier im wesentlichen nur auf die Unterschiede zwischen beiden eingegangen werden soll. Der Kontrollfluss ist identisch mit dem von PERI. RUPT wartet auf Events ((Dis-) Konnektierungs-Event `conn_event` oder Connect Interrupt Event `cir_event`), führt die entsprechenden Aktionen aus und geht dann wieder in den Event-Wartezustand.
Konnektierungen und Diskonnektierungen laufen nach den gleichen Mechanismen wie bei PERI ab. Die Inhalte von Konnektierungen sind andere: Interrupts anstelle von Zeit. Diese Inhalte sind detailliert im Rahmen der Recordstrukturen der RUPT-Queue und der Konnekt-Message beschrieben.
Das Aktualisieren eines konnektierten Auftrags, d. h., der Aufbau und das Senden einer Message an die Dispatcher, wird durch den Connekt Interrupt Event `cir_event` ausgelöst. Anders als bei PERI, wo die Quelle des Start-Events nur die Timer-ISR sein konnte, gibt es bei den Interruptkonnektierungen viele Quellen. Implementiert sind jeweils 3 ISRs mit fester Bedeutung (Alarm, ECInfo und Timingbusevent) für die 9 möglichen SEs. Vom Anwender können auf einer anderen Hardware (Ersatz für die SE) zusätzliche Interrupts integriert werden, vorausgesetzt, sie sind systemkonform implementiert.
Das Ordnungsprinzip der RUPT-Queue ist die Nummer des konnektierten Interrupts, was bei Suchaktionen in der Queue einen kleinen Effektivitätsvorteil gegenüber einer ungeordneten Queue bietet.

8.3 Überwachung von Konnektierungen

Eine Konnektierung ist eine unidirektionale Verbindung. Alle Pakete laufen von MOPS zu Prozessen auf der VMS-Ebene. Damit ist MOPS zunächst nicht in der Lage, zu entscheiden, ob der Knoten oder der Prozess auf dem Knoten, an den es das Paket schicken möchte, noch existiert. Das Paket wird auf jeden Fall geschickt.
Trotzdem muss es eine Überwachung von Konnektierungen geben, um zu gewährleisten, dass Konnektierungen, deren Senken nicht mehr vorhanden sind, aufgehoben werden. In MOPS ist diese Überwachung implementiert, wobei zwei Möglichkeiten unterschieden werden.

1. Der Zielknoten existiert nicht mehr. CP führt eine Liste aller Knoten am Netz, die ständig auf dem neuesten Stand gehalten wird. Ist der Zielknoten nicht mehr in der Liste, ändert CP den Packet Code im Transport Header des Paketes von `data_pc` auf `error_pc + node_not_exist` und schickt das Paket an MOPS zurück. MOPS hebt daraufhin alle Konnektierungen an diesen Knoten auf.

2. Der Zielprozess auf dem Zielknoten existiert nicht mehr. Auf dem Zielknoten führt Netman eine Liste aller (Kontrollsystem-) Prozesse, die auf diesem Knoten laufen. Auch diese wird ständig auf dem neuesten Stand gehalten. Ist der Zielprozess nicht mehr in der Liste, ändert Netman den Packet Code im Transport Header des empfangenen Paketes von `data_pc` auf `error_pc` und schickt das Paket an MOPS zurück. MOPS hebt daraufhin alle Konnektierungen an diesen Prozess auf diesem Knoten auf.

Bisher ungelöst sind die Probleme, die entstehen, wenn MOPS oder ECM einen Reset machen. Nach einem Reset des MOPS sind alle Konnektierungen aufgehoben, ohne dass der Benutzer einerseits und, bei Eventkonnektierungen, der ECM andererseits davon erfährt. Nach einem Reset des ECM bestehen Eventkonnektierungen auf der MOPS-Seite nach wie vor, auf der ECM-Seite nicht mehr.
In allen Fällen ist die Konnektierung unterbrochen. Sie wird weder wieder hergestellt, noch wird der konnektierte VMS-Prozess über die Unterbrechung informiert. Er wartet vergeblich auf weitere Pakete.
Es ist denkbar, bestimmte Informationen über Konnektierungen reset-fest im Speicher zu halten. Damit wäre es nach einem Reset zumindest möglich, VMS-Prozesse über unterbrochene Konnektierungen zu informieren. Dies ist allerdings nicht mehr möglich, wenn am VME-Rahmen der Strom ausgefallen war.
Für neue Versionen von MOPS und ECM ist dies ein wichtiger Punkt, der gelöst und implementiert werden muss.

9 Alarmsystem

```
% <text>
%    a) Lib-Funktionen (user interface): siehe unter Benutzerbibliothek
%    b) ALRM-Task, TIMR-Task und QALR-Messagequeue.
%       Prioritätsüberlegungen, ...
%    c) Handling der Alarm-Interrupts (Status-, Error- und Configänderung;
%       User-Alarme von der SE in INTSER und RUPT.
%    d) SSRs S_Alarm; W_Shutup, R_Shutup für MOPS und ECM
%    e) geraeteunabhaengige / geraetespezifische Alarme
%    f) MOPS schickt nicht jeden Alarm als einzelnes Ethernetpaket auf
%       das Netz.
%       Mehrere Alarme werden zu einem Paket geschnürt und auf das Netz
%       geschickt. MOPS sammelt Alarme und verschickt sie erst,
%       \begin{itemize}
%       \item wenn ein Ethernetpaket gefüllt ist oder
%       \item wenn eine Zeit von 20 Sekunden seit dem letzten Verschicken eines
%           Paketes vergangen ist.
%       \end{itemize}
```

```
%    g) Anderer Offset als bei Auftrags-Paketen beachten. Es gibt KEINEN
%       Endoffset!\index{Offset>- eines Alarmpakets}
```

Die Schnittstelle zur Alarm-Bibliothek des MOPS ist in `ALARM$SUPPORT.PIN` definiert und in [7] beschrieben.
Weitere Informationen zu Alarmen und deren Verarbeitung finden sich in [4] und [2].

10 Lokales Terminal-I/O

Zum Zweck der Diagnose vor Ort wurde die Task SHOW kreiert. Sie ermöglicht es, wichtige Informationen über Zustände des MOPS und seiner Komponenten mittels einer einfachen Menübedienung zu erhalten. Zur Zeit kann auf folgende Informationen zugegriffen werden:

- Menu: Dies ist das Menü zur Auswahl der hier aufgeführten Punkte.
- General Informations: In dieser Übersicht erhält man einen groben Überblick über das System. Er zeigt die Anzahl der unterstützten Gerätemodelle, die Anzahl periodischere und interrupt-konnektierter Aufträge, die Knotennummer und die Info, ob Cache, Watchdog und Alarme enabled sind. Weiterhin gibt es eine Statistik über die Anzahl der empfangenen und gesendeten Pakete, die Anzahl der ausgeführten periodischen und interrupt-konnektierten Aufträge, sowie die Anzahl der gepufferten, der verschickten und der verlorenen Alarme. Dabei werden alle Zähler bei einem Reset zurückgesetzt.
- Device Configuration: Alle Geräte, die auf diesem VME-Rahmen bekannt sind, werden mit ihrer physikalischen und logischen Geräteadresse, der Nummer der SE an der sie angeschlossen sind und ihrem Zustand (online, offline, ...) angezeigt.
- EC Configuration: Alle SEs und der Ethernet Controller (CP) werden mit ihrem Zustand (online, ...), ihrem Type (MP1050, FIO8130), der Startadresse des Dualport-RAMs, des implementierten Gerätemodells und ihrem aktuellen Lebenszeichenzähler angezeigt.
- Periodical Queue: Die (maximal ersten 10) Elemente der Queue für periodisch konnektierte Aufträge werden mit Prozessname und Knotennummer des konnektierten VMS-Prozesses, Gerätemodell- und XSR-Nummer der konnektierten XSR sowie einer genauen Beschreibung der Konnektierung angezeigt.
- Interrupt Queue: Die (maximal ersten 10) Elemente der Queue für interrupt-konnektierte Aufträge werden mit Prozessname und Knotennummer des konnektierten VMS-Prozesses, Gerätemodell- und XSR-Nummer der konnektierten XSR sowie einer genauen Beschreibung der Konnektierung angezeigt.
- Implemented XSRs: Hier werden zunächst alle im Rahmen implementierte Gerätemodelle angezeigt. Wird ein Gerätemodell ausgewählt, so erhält man eine Anzeige aller XSRs des ausgewählten Gerätemodells mit den Startadressen der Ini- und Runteile, dem maximalen Returnbytecount und der Anzahl der Aufrufe der XSR seit dem letzten Reset. Es ist möglich durch diese Anzeige zu scrollen.
- Error Buffer: Dieser Punkt ist zur Zeit noch nicht realisiert. Geplant ist, die letzten n Fehlermeldungen des MOPS mit Datum und Uhrzeit resetsicher zwischenzuspeichern und sie mit diesem Menüpunkt anzuzeigen.
- Restart Stastistics: Es gibt insgesamt 15 Gründe für einen Restart des MOPS. Davon sind 8 MOPS und 7 CP zugeordnet. Dieser Punkt zeigt für jeden Grund die Anzahl der Restarts seit dem letzten Einschalten und den Zeitpunkt des letzten Restarts.
- Hardware Exceptions: Hier werden die Anzahl der AC-Fails und der Spurious Interrupts seit dem letzten Einschalten des Rahmens angezeigt.

- Cache enable/disable: Anzeige, ob der Cache der CPU ein- oder ausgeschaltet ist. Dieser Menüpunkt erlaubt es auch, den Cache ein- oder auszuschalten, was nur lokal möglich ist.
- Global Types: Dieser Punkt ist im Grunde überflüssig geworden. Er zeigt die Grössen und die Start- und Endadressen der meisten globalen Datentypen des MOPS. Da diese mittlerweile in einem Record untergebracht sind, kann es zu keinen Überlappungen mehr kommen.
- Extended Output Area: Alle anderen Tasks außer SHOW haben nur die untersten fünf Zeilen des Bildschirms für ihre Ausgaben zur Verfügung. Um mehrzeilige Ausgaben, wie zum Beispiel ein »Procedure Walkback«, komplett darstellen zu können, kann die Scroll Region mit diesem Menüpunkt auf 18 Zeilen vergrößert werden.

Der Bildschirm wird ständig aufgefrischt, sodass Änderungen sofort angezeigt werden. Bedient wird mit einfachen Tastendrücken ohne die Returntaste. Spezielle Eingaben müssen allerdings mit Return abgeschlossen werden.
SHOW nutzt den größten Teil des Bildschirms für seine Ausgaben. Alle anderen Tasks haben normalerweise nur die untersten fünf Zeilen des Bildschirms für ihre Ausgaben zur Verfügung. Die Scroll Region ist auf diese Zeilen beschränkt. Nach jeder Ausgabe eines Strings positioniert SHOW den Cursor in die untere, linke Ecke des Bildschirms. Ausgaben von anderen Tasks landen damit automatisch in der von SHOW definierten Scroll Region, ohne dass diese den Cursor erneut positionieren müssten. Ein einfaches `WRITELN` genügt. Durch die eingeschränkte Scroll Region wird der SHOW-Teil des Bildschirms nicht beeinflusst.
MOPS benötigt für dieses I/O einen speziellen Terminaltreiber, der in Abschnitt 3.2 auf Seite 12 näher beschrieben wird.

11 System Service Routinen

Auch die Computer auf der VME-Ebene (GuP und SE) sind Geräte im Sinne des Kontrollsystems. Das Gerätemodell des GuP ist `ECMS`, das der SE ist `EC`. System Service Routinen repräsentieren die Gerätemodelle dieser Computer. Weitere SSRs stellen Dienstleistungen für MOPS zur Verfügung.
Die SSRs sind in [6] und [8] genau beschrieben. Hier wird nur auf die speziellen SSRs zur Behandlung von konnektierten Aufträgen eingegangen.

11.1 Die SSRs DBSLoad, EEPROMLoad und Download

Die SSR DBSLoad (`s_dbsl`) dient zum Upload der lokalen Datenbasis auf die Leitrechnerebene (VMS).
Die SSR EEPROMLoad (`s_eeprom_load`) dient zum Download der lokalen Datenbasis auf die Kontrollrechnerebene (VME). Dort wird sie in der Watchdog-Karte gespeichert.
Die SSR Download (`w_dload`) dient zum Download der VME-Software.
Wichtig bei diesen drei SSRs ist, dass deren XSR-Nummern (`dbsl_nr`, `eeprom_load_nr`, `dload_nr`) in verschiedenen Programmen auf der Leitrechnerebene fest „verdrahtet“ sind (REDABAS, AUTCON, ...). Die Nummern dürfen also nur dann geändert werden, wenn auch die entsprechenden Operatingprogramme angepasst werden.

11.2 Die SSR Connect

`S_Conn` interpretiert das Empfangspaket und baut damit eine Message auf, die dann an die Messagequeues QPER (→ PERI) oder QRUP (→ RUPT) geschickt wird. Die Struktur einer solchen Message ist im folgenden dargestellt (wegen der Vielfältigkeit der Informationen, die für die verschiedenen Fälle darin enthalten sein müssen, ist dies eine etwas komplexere Variant-Record-Struktur). Da PERI oder RUPT mit dieser Message dann ein Element ihrer Queue füllen, entspricht ein Teil der Message-Struktur der Struktur eines Queue-Elements. Die Struktur zeigt Abbildung 11.2 auf Seite 39.
Die Beschreibung der Elemente dieses Records:

```
conn_msg_type = PACKED RECORD
    CASE action: action_type OF
        connect:
           (target_id: uns_byte;
            pack_adr: pack_pointer;
            CASE conn_id_type OF
                peri_id, time_id:
                   (count: uns_word;
                    act_delta: uns_long;
                    tot_delta: uns_long);
                alarm_id, ecinfo_id, event_id:
                   (CASE BOOLEAN OF
                        TRUE:
                           (con_se_nr: uns_byte;
                            con_int_nr: uns_byte;
                            con_int_desc: uns_word);
                        FALSE:
                           (con_se_int_nr: uns_word;
                            con_acc: uns_byte;
                            con_evt: uns_byte)));
        disconnect:
           (dcon_desc: disconn_desc_type;
            packet_id: uns_word;
            proc_name: PACKED ARRAY [1..6] OF CHAR;
            node_id: uns_word);
        isr_msg:
           (dummy: uns_byte;
            CASE INTEGER OF
                  1: (isr_se_nr: uns_byte;
                      isr_int_nr: uns_byte;
                      isr_int_desc: uns_word);
                  2: (isr_se_int_nr: uns_word;
                      isr_int_desc_acc: uns_byte;
                      isr_int_desc_evt: uns_byte);
                  3: (isr_dummy: uns_word;
                      isr_int_dummy: uns_byte;
                      isr_int_desc_set: ecinfo_int_set_type))
    END;
```

Abbildung 12: Struktur der Connect Message

action `connect` für einen Konnektierungsauftrag

target_id eine Kennung, ob die Anwortpakete des Konnektierungsauftrags ein *lokales* Ziel haben oder über Ethernet an eine Alpha gehen (bisher ist nur `ethernet_id` implementiert)

pack_adr die Adresse des kopierten und modifizierten Empfangspakets

conn_id_type mit `peri_id` kennzeichnet es einen periodischen Auftrag, bei dem die Wiederkehr (Periode) in Ticks vorgegeben ist, mit `time_id` kennzeichnet es einen periodischen Auftrag, der zu der mit Stunde/Minute/Sekunde definierten Zeit ablaufen soll und dessen Wiederkehr 24 Stunden ist

count Anzahl der periodischen Aufrufe

act_delta Zeitintervall (in Ticks) bis zum erstmaligen Aufruf

tot_delta Wiederkehr (Periode) in Ticks

conn_id_type `alarm_id` steht für die Verknüpfung mit einem Alarm, `ecinfo_id` steht für die Anbindung an einen Kaltstart einer SE, die Änderung einer von einer SE kontrollierten Gerätekonfiguration, ein Lebenszeichen einer SE oder einen IPS-Interrupt, und `event_id` für die Anbindung an ein Muster auf dem Eventbus.

con_se_nr die Nummer der den Interrupt auslösenden SE

con_int_nr die Nummer des Interrupts von der konnektierten SE

con_int_desc eine fallabhängige weitergehende Beschreibung eines SE-Interrupts

con_se_int_nr ist die Zusammenfassung von `con_se_nr` und `con_int_nr` in einem 16-Bit-Wort

action `dis_connect` für die Terminierung eines Konnektierungsauftrags

dcon_desc `packet` für die Terminierung eines durch den `packet_id`, den `process_id` und den `node_id` gekennzeichneten Auftrags, `process` für die Terminierung aller Aufträge, die durch `proc_name` und `node_id` gekennzeichnet sind, und `node` für die Terminierung aller Aufträge eines VMS-Knotens, definiert durch `node_id`.

packet_id der aus dem Auftragspaket kopierte Ethernet packet_id.

proc_name der Name des VMS-Prozesses, der die Konnektierung aufgesetzt hat.

node_id Die Nummer des VMS-Knotens, von der der Auftrag aufgesetzt wurde.

action `isr_msg` für eine Message, die von einer SE-Interrupt-Service-Routine (`se_isr`) an RUPT geschickt wird.

isr_se_nr die Nummer der SE, die den Interrupt ausgelöst hat

isr_int_nr die Nummer des Interrupts von der auslösenden SE

isr_int_desc eine fallabhängige weitergehende Beschreibung des ausgelösten SE-Interrupts

isr_se_int_nr ist die Zusammenfassung von `isr_se_nr` und `isr_int_nr` in einem 16-Bit-Wort

Alle Konnektierungen, die mit der Zeit zu tun haben, gekennzeichent durch `peri_id` oder `time_id`, werden PERI zugeordnet, d. h., der Konnekt-Task-Ident `conn_tid` wird mit `peri_tid` und der Konnekt-Queue-Ident `conn_qid` mit `peri_qid` besetzt. Alle Konnektierungen, die mit Interrupts oder Events verbunden sind (`alarm_id`, `ecinfo_id`, `event_id`), werden RUPT durch `rupt_tid` und `rupt_qid` zugeordnet.
`S_Conn` baut ein neues Empfangspaket auf, indem es sein eigenes Empfangspaket im Dualport-RAM unter Auslassung des sie selbst aufrufenden XSR-Teils kopiert. Bei Aufträgen an die SSR `S_Conn` gibt es *zwei* XSR-Header. Der äußere ist der Header für `S_Conn` selbst, die Daten für `S_Conn` bestehen aus dem Auftragspaket für die zu konnektierende XSR, also aus deren Header (dem inneren), deren Parametern und deren Daten. Das kopierte Paket bleibt im lokalen RAM so lange erhalten wie die Konnektierung besteht.

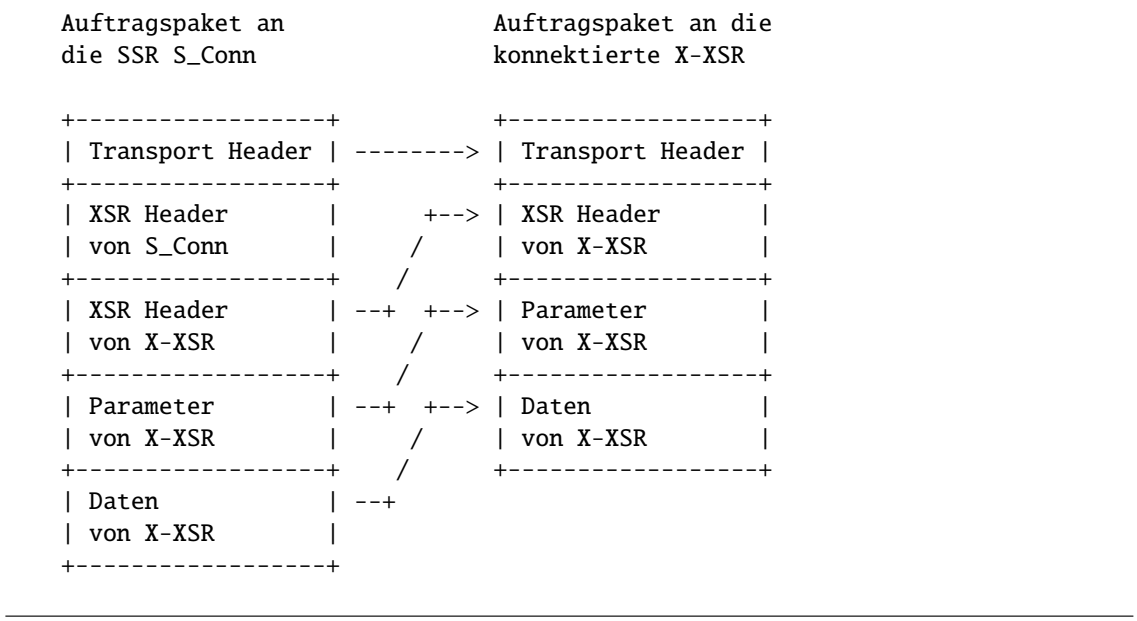

Abbildung 13: Erstellen des konnektierten Auftragspaketes an X-XSR

In Abbildung 13 ist der Kopiervorgang nochmals verdeutlicht, wobei `X-XSR` die zu konnektierende XSR darstellt.
Synchrone uns asynchrone Pakete unterscheiden sich in der Paketkennung `PacketIdent` des Ethernet Transport Headers. Synchrone Kennungen liegen im Bereich $-127 \ldots -1$, asynchrone Kennungen im Bereich $1 \ldots 127$. Die Kennung 0 ist nicht zulässig. Diese Kennung ist notwendig, um auf der VMS-Ebene die Antworten den entsprechenden Aufträgen zuzuordnen.
Der Auftrag an `S_Conn`, eine Konnektierung herzustellen, ist ein synchroner Auftrag. Er wird mit einer Quittung beantwortet und hat deshalb eine negative Paketkennung.
Auch die asynchronen Antworten müssen dem entsprechenden Auftrag zugeordnent werden können. Deshalb wurde definiert, dass alle asynchronen Antworten auf die Konnektierung mit der Kennung $-n$, mit der Kennung n versehen werden. Aus diesem Grund wird beim Erstellen des konnektierten Auftragspaketes der *Absolutwert* des `PacketIdent` des Transport Headers kopiert.
Vor der Übergabe der weiteren Bearbeitung an PERI oder RUPT bereitet `S_Conn` einige, für die Art der Konnektierung spezifische Parameter auf und trägt diese in das Message-Paket ein:

Für die Message an PERI:

Alle Zeitangaben bei konnektierten Aufträgen müssen in eine Anzahl von Ticks umgewandelt werden. Ist der `connect_id` gleich `peri_id`, dann wird das `delta` schon von oben (VMS/Userface) als Parameter vorgegeben. Dieses `delta` wird in `tot_delta` eingetragen. In `act_delta` wird eine 1 geschrieben, was den fast sofortigen (Verzögerung = 1 Tick) Ablauf des aufgesetzten Auftrags bewirkt.

Ist der `connect_id` gleich `time_id`, dann wird die vorgegebene absolute Zeit (Stunden, Minuten und Sekunden) in ein Zeit-Delta (Anzahl von Ticks) zur aktuellen Zeit umgerechnet. Das Zeit-Delta wird in `act_delta` eingetragen, `tot_delta` wird mit der Anzahl von Ticks für 24 Stunden besetzt.

Für die Message an RUPT:

Hier ist der `connect_id` mit `alarm_id`, `ecinfo_id` oder `event_id` besetzt. Die entsprechenden Informationen sind als Parameter im Auftragspaket enthalten und werden in die oben beschriebenen

Message-Strukturen eingetragen. Für die Konnektierung an ein Event (`event_id`) wird dabei durch die Auswertung des SE-Dualport-RAMs überprüft, ob die SE, die den event-getriggerten Interrupt auslösen soll, überhaupt existiert und diesen Auftrag ausführen kann. Wenn keine spezifische SE angegeben ist, wird die erstbeste SE genommen, die die Event-Konnektierung bearbeiten kann. Der zutreffenden SE wird der Konnektierungsauftrag über ihr Dualport-RAM übergeben.

Die so aufbereitete Message wird an QPER oder QRUP geschickt und das Event `conn_event` entsprechend an PERI oder RUPT, um diese auf die Message aufmerksam zu machen.
Zum Schluss antwortete `S_Conn` mit einer Quittung, die angibt, ob die Konnektierung hergestellt wurde oder ob ein Fehler auftrat.

11.3 Die SSR Disconnect

Die Abläufe bei der Diskonnektierung, d. h. bei der Terminierung eines konnektierten Auftrags, sind in dem Abschnitt 8.1 beschrieben. Periodische Aufträge werden von MOPS nach dem Ende der vorgegebenen Anzahl von Abläufen (`conn_count`) automatisch terminiert. Periodische Aufträge mit einem unendlichen `conn_count` und interruptkonnektierte Aufträge müssen explizit von der VMS-Knoten-Ebene über die Standard-Schnittstelle Userface diskonnektiert werden. Das von Userface aufgebaute Paket löst über die Standardabläufe in MOPS den Aufruf der SSR `S_Dcon` aus. Diese überträgt aus dem Empfangspaket die Diskonnektierungsparameter in eine Message und schickt diese an PERI (über QPER) oder an RUPT (über QRUP).

12 Interrupts

Die Organisation und der Ablauf von Interrupts in Systemen mit Motorola-680x0-Prozessoren soll hier als bekannt vorausgesetzt werden. Zu beschreiben ist die Anbindung oder Integration der Interrupts in pSOS+ und MOPS.
Das Verhältnis Interrupts ↔ pSOS+ ist einfach: es existiert im Prinzip nicht. Interrupts laufen an pSOS+ völlig vorbei. Sie unterbrechen die gerade laufenden pSOS+-Task, und es muss nur Sorge dafür getragen werden, dass nach dem Ende der Interruptbehandlung keine Register oder Stacks verändert wurden.
Gleichwohl können von einer ISR eine beschränkte Anzahl von pSOS+ System Services aufgerufen werden. Von dieser Möglichkeit wird in MOPS Gebrauch gemacht, indem zum Beispiel pSOS+-Events von ISRs an MOPS-Tasks geschickt werden.
Normalerweise führt ein pSOS+ Service Call zum Scheduling. Dies ist bei ISRs nicht erwünscht. Die ISR muss in kürzester Zeit beendet sein. pSOS+ kehrt bei Service Calls, die aus einer ISR heraus gemacht wurden, zur ISR zurück. Allerdings kann es gewünscht sein, dass nach Beendigung der ISR ein Scheduling stattfindet, um eine (hochprioritäre) Task zu starten, um z. B. ein Event zu bearbeiten, das ihr von der ISR geschickt wurde. Dies ermöglicht der System Service Call `I_RETURN`. Er ersetzt das `RTE` am Ende der ISR und bewirkt, dass pSOS+ ein Scheduling ausführt. Wo es möglich ist, sollten die ISRs immer `I_RETURN` anstatt `RTE` benutzen!
Die ISRs, die MOPS-Funktionen ausführen, sind in den entsprechenden Abschnitten schon teilweise beschrieben:

RECV-ISR : Wird vom CP getriggert und schickt das `recv_event` an die Task RECV.

XEND-ISR : Wird vom CP getriggert und schickt das `xend_event` an die Task XEND.

SE-ISRs : Davon gibt es für jede mögliche SE drei Assemblerrahmen, entsprechend den drei verschiedenen Interrupts, die eine SE generieren kann, insgesamt also 27. Der Pascalteil, den es nur einmal gibt, baut unter Auswertung des globalen MOPS-Datenbereiches und des SE-Dualport-RAMs eine Message auf und schickt diese und das `cir_event` an die Task RUPT.

BUS-ISR : Die *eigene* Bus/Adresserror-ISR. Setzt ein Flag in dem globalen Datenbereich von MOPS. Wird benutzt, um die Existenz von SEs (= die Existenz von zugehörigen Dualport-RAM-Adressen) festzustellen.

TIMER-ISR : Eine doppelte Funktion hat die Timer-ISR. Sie ruft bei jedem Durchlauf den pSOS+ System Service `tm_tick` (Announce Tick) auf, der von der internen Zeitverwaltung von pSOS+ verarbeitet wird. Gleichzeitig übernimmt sie noch die Zeitverwaltung der PERI-Queue: Das Dekrementieren des `delta` und das `delta`-abhängige Senden des `timer_event` an PERI. Die Timer-ISR ist in Assembler geschrieben und zusammen mit der pSOS+/pROBE+-Systemanpassung implementiert.

I/O-ISRs : Die Ein- und Ausgabe am Terminal ist ebenfalls interruptgesteuert. Die zugehörigen ISRs sind in Assembler geschrieben und zusammen mit der pSOS+/pROBE+-Systemanpassung implementiert.

Alle ISRs bis auf die TIMER- und I/O-ISRs sind in Pascal geschrieben. Da es in Pascal keine dem Assemblerbefehl RTE entsprechende Instruktion gibt, sind die Pascal-ISRs in einen Assemblerrahmen eingebettet. Der Rahmen besteht aus den Teilen:

- Rettung der Register.
- ISR-abhängig: Durchführung einiger vorbereitender Operationen. Übertragung von ISR-spezifischen Paramatern.
- Aufruf der Pascal-ISR.
- Terminierung mit dem pSOS+ Service Call `I_RETURN`, der neben der RTE-Instruktion noch ein pSOS+ Task Scheduling auslöst.

Anstelle der eigentlichen Pascal-ISR werden die entsprechenden Vektoren mit den Adressen der Assemblerrahmen besetzt.

13 Fehlerbehandlung

13.1 Fehlerquellen

In MOPS gibt es vier unterschiedliche Ebenen auf denen Fehler auftreten können.

Ebene 1 ist die hardware- oder prozessornahe Ebene mit Fehlern, die über den Exception-Mechanismus des 680x0-Prozessors abgehandelt werden. Solche Fehler sind zum Beispiel

- Bus/Adressfehler: der Zugriff auf eine falsche oder ungültige Adresse;
- Privilegverletzung: der Versuch im Usermode eine Instruktion auszuführen, die nur im Supervisormode erlaubt ist;
- Division durch Null: der Versuch mit einer `DIVx`-Instruktion durch Null zu Teilen;
- Spurious Interrupt: das Auftreten eines falschen, nicht erwarteten Interrupts;
- etc.

Ebene 2 ist die systemnahe Ebene mit Fehlern, die das Betriebssystem pSOS+ oder die anderen Systemkomponenten pROBE+ oder pREPC+ generieren. Damit sind *nicht* die Fehler gemeint, die bei einem Service Call zurückgeliefert werden können. Diese werden in der Ebene 4 abgehandelt. Ausschließlich systeminterne Fehler gehören zur Ebene 2.

Diese Ebene ist bisher nicht implementiert.

Ebene 3 ist die compilernahe Ebene mit Fehlern, die der Compiler generiert. Dies sind die Runtime-Checks des Pascal-Compilers wie zum Beispiel

- zu großer/kleiner Index eines Arrays;
- zu großer/kleiner Wert für einen Datentyp;
- Stack overflow;
- etc.

Ebene 4 ist die Ebene mit Fehlern der Betriebszustände des MOPS, die das ordnungsgemäße Funktionieren des Gesamtsystems behindern oder unmöglich machen. Zum Beispiel:

- Messages werden an eine Messagequeue geschickt und dort nicht mehr oder nicht schnell genug abgeholt. Wenn die als Maximum definierte Anzahl von Messages an der Messagequeue hängt, kann keine Message mehr eingequeued werden.
- Im Memory-Pool ist kein Speicherplatz mehr für Empfangs- oder Sendepakete ver-fügbar, weil benutzter Speicherplatz nicht mehr oder nicht schnell genug freigegeben wird.
- Ein Empfangspaket enthält falsche Strukturen.
- In einer Connect-Queue gibt es kein freies Element mehr.
- etc.

13.2 Fehlerschweren

Fehler jeder Ebene können verschiedene Schweren (Severities) haben. Die fehlergenerierenden Module oder der Errorhandler selbst entscheiden, von welcher Schwere ein aufgetretener Fehler ist. Für die gesamte VME-Ebene sind zwei verschiedene Schweren definiert, die auch von MOPS benutzt werden.

minor_error ist ein minder schwerer Fehler. Der Errorhandler versucht, die Ursache des Fehlers zu klären und, wenn möglich, zu beheben.

fatal_error ist ein schwerer Fehler. Der Errorhandler hat keine Chance mehr diesen Fehler zu beheben. Als einzig mögliche Aktion bleibt ein Neustart des Gruppenmikros. So ist zum Beispiel fehlendes Memory zum allokieren eines Sendepaketes ein schwerer Fehler.

Hier gibt es allerdings eine Ausnahme! Tritt ein Fehler in einer XSR auf, so muss nicht der gesamte MOPS erneut gestartet werden, sondern nur der zugehörige Dispatcher DInn. In Abschnitt 13.5.3 wird darauf näher eingegangen.

13.3 Fehlertypen

Für die gesamte VME-Ebene sind systemweite Fehlertypen, sogenannte Flavours, definiert. Diese Typen unterscheiden im wesentlichen die Herkunft des Fehlers. Jeder Typ beschreibt den Fehler mit eigenen Parametern und führt zu einer speziellen Behandlung durch den Errorhandler.
Die von MOPS benutzen Fehlertypen sind hier aufgeführt:

err_hwexc ist der Fehlertyp der Ebene 1. Er kennzeichnet Fehler, die durch Exceptions generiert wurden. Solche Fehler werden von der systemweiten Prozedur `hw_error` behandelt. Sie schickt eine Message mit der Fehlerbeschreibung via QERR an den Errorhandler EHDL und suspendiert die gerade laufende Task, damit deren Stackframe auf jeden Fall bis zur Auswerung durch EHDL erhalten bleibt.

err_sys ist der Fehlertyp der Ebene 2. Er kennzeichnet Fehler, der Systemsoftware `pSOS+`, `pROBE+` und `pREPC+`. Alle Fehler zur Laufzeit, die nicht über den Return Code einer aufgerufenen Systemfunktion abgehandelt werden können (Ebene 4) werden über den `K_FATAL`-Ausgang von `pSOS+` behandelt.

Fehler diese Typs werden bis dato nicht behandelt.

err_pas ist der Fehlertyp der Ebene 3. Er kennzeichnet Laufzeitfehler (Runtime Errors) des Compilers. Der Organon-Compiler ruft bei aufgetretenen Fehlern eine Fehlerprozedur auf und übergibt dieser einige Parameter, die den Fehler beschreiben. Das sind zum Beispiel die Zeilennummer in der Quelldatei, in der der Fehler auftrat, oder der Index, der zu einem Überlauf führte. Diese Routinen (wie z. B. `error_` oder `range_`), die in der Pascalbibliothek enthalten sind, wurden zum Teil durch GSI-eigene Routinen ersetzt, um sie an das Errorhandling der Systemsoftware anzupassen. Die zentrale Instanz ist hier die Prozedur `pas_error`. Sie schickt, wie `hw_error`, eine Message mit der Fehlerbeschreibung via QERR an den Errorhandler EHDL und suspendiert die gerade laufende, fehlerhafte Task.

err_mops ist der Fehlertyp der Ebene 4. Die Tasks des MOPS, die XSRs als auch die Module der MOPS-Library rufen im Fehlerfall die Prozedur `error_io` mit den Parametern Modulname (`module`), Fehlernummer (`err_num`), einem dem Fehler entsprechenden Parameter (`param`) und der Fehlerschwere (`severity`) auf. Die Prozedur baut eine Message mit diesen Parametern auf, kennzeichnet sie als Fehler des Typs `err_mops` und schickt sie über die Messagequeue QERR an den Errorhandler EHDL.

13.4 Definitions- und Implementationsmodule

Die Grundlagen der Fehlerbehandlung sind für die gesamte VME-Ebene die gleichen. Dies hat verschiedene Gründe. Einerseits haben der 680x0-Prozessor selbst als auch der Pascal-Compiler inhärente Methoden der Fehlerbehandlung, von denen nicht oder nur in Grenzen abgewichen werden kann. Andererseits vereinfacht eine einheitliche Fehlerbehandlung die Struktur und Pflege der Software.
Aus diesen Gründen gibt es VME-weite Definitions- und Implementationsmodule, die für Ethernetcontroller, Gruppenmicro und SE gemeinsam gelten und systemeigene Module, die nur für ein System zuständig sind.
Die VME-weiten und die MOPS-eigenen Module seien hier aufgeführt. Die systemweiten Module befinden sich alle auf der Directory `XLIB$ERROR`[13].

ERROR_DEF.PIN Dieses Modul enthält die Definitionen der Fehlerschweren (Severities), der Prozessor-Exceptions, der Fehlertypen (Error Flavours), der Struktur der Message für den Errorhandler (Error Handler Message) sowie die Definitionen der Strukturen der Fehlerdeskriptoren.

ERROR.ASM Dieses Implementationsmodul enthält alle für die Fehlerbehandlung notwendigen Assemblerprozeduren. Hauptsächlich sind dies die Assemblerrahmen für die Behandlung der Exceptions des 680x0-Prozessors. Die Adressen dieser Prozeduren sind die entsprechenden Vektoren in der Vektortabelle des Prozessors.

EXCEPTIONHANDLER.PAS Dieses Implementationsmodul enthält die Pascal-Prozedur `hw_error`, die von den Assembler-Prozeduren in `ERROR.ASM` aufgerufen wird. Diese Prozedur ist verantwortlich für das Zusammenbauen einer dem Fehler entsprechenden Message und für das Senden dieser Message über die Messagequeue QERR und die Partition PERR an den Errorhandler. Das Modul setzt voraus, dass die Messagequeue QERR und die Partition PERR existiert.

PASCALERROR.PAS Dieses Implementationsmodul enthält die Routinen, die die vom Pascalcompiler generierten Laufzeitfehler behandeln. Sie ersetzen die originalen Organon-Routinen.

Die MOPS-eigenen Module befinden sich entweder auf `MOPS68` oder auf `MOPSLIB68`.

ERROR.PIN Dieses Modul enthält die Definitionen aller möglichen MOPS-Fehler. Außerdem deklariert es die Prozeduren `error_io` und `MOPSErrMsg`—.

[13] Die ursprünglichen C-Quellen der Module, die durch GSI-eigene ersetzt wurden, befinden sich auf der Directory `NODEC$ROOT:-[ORGANON.DISTPC.LIBSRC]`

ERROR.PAS Dieses Modul enthält die Implementationen der Prozeduren `error_io` und `MOPSErrMsg`. Es ist Teil der MOPS-Bibliothek.

`error_io` ist die zentrale Prozedur zum Melden eines Fehlers der Ebene 4 (siehe Abschnitt on page 43). Sie wird von allen MOPS-Modulen benutzt, um eine Fehlermeldung via QERR an den Errorhandler EHDL zu schicken.

`MOPSErrMsg` interpretiert die Fehlermeldungen und gibt einen entsprechenden String auf dem lokalen Bildschirm aus.

EHDL.PAS Dieses Modul ist die Implementationen des Errorhandlers. Er wird in Abschnitt 13.5.2 näher beschrieben.

13.5 Errorhandler

In MOPS gibt es im Grunde zwei Errorhandler, die je nach Fehler bzw. Zustand des MOPS in Aktion treten können. Es sind dies der System Debugger `pROBE+` und der eigentliche Errorhandler EHDL, eine Task des MOPS.

13.5.1 pROBE+

`pROBE+`, der System Debugger ist für alle Fehler, die nicht vom Errorhandler EHDL behandelt werden, zuständig. Während der Startphase des MOPS, zu der Zeit, wo es noch keinen EHDL gibt, ist er für die Behandlung *aller* auftretenden Fehler verantwortlich. `pROBE+` unterscheidet nicht zwischen `minor_error` und `fatal_error`. Alle Fehler führen zu einer Ausgabe auf dem lokalen Bildschirm und zur Unterbrechung von MOPS, da `pROBE+` die Kontrolle behält und auf eine Eingabe des Benutzers wartet.

13.5.2 EHDL

EHDL, der Errorhandler, eine eigenständige Task des MOPS, ist die zentrale Instanz, die während der Laufzeit des MOPS für alle „üblichen“ Fehler der Ebenen 1 und 2 (siehe oben), sowie für alle Fehler der Ebenen 3 und 4 verantwortlich ist. EHDL unterscheidet die verschiedenen Fehlerschweren und -Typen und agiert entsprechend. Nicht alle Fehler der Ebene 1 und 2 werden von EHDL behandelt. Die meisten dieser Art führen nach wie vor in `pROBE+`. Für einige wenige, wie zum Beispiel AC-Fail- und Spurious Interrupts gibt es spezielle Behandlungsprozeduren.

13.5.3 Funktionen von EHDL

Nach dem Kreieren und Starten des EHDL erledigt dieser zunächst einige Initialisierungsaufgaben. Diese bestehen haupsächlich aus dem Setzen der Exceptionvektoren der Exceptions, die von MOPS behandelt werden sollen.
Nach der Initialisierungsphase begibt sich EHDL in die Main Loop, eine Endlosschleife, in der er an der Messagequeue QERR auf Fehlermeldungen von anderen Modulen wartet.
Erreicht eine Fehlermeldung EHDL, so sind seine weiteren Aktionen determiniert vom Typ und von der Schwere des Fehlers und teilweise auch vom Ort, an dem der Fehler auftrat.
Die verschiedenen Aktionen sind hier nach Fehlertyp und Fehlerschwere geordnet.

err_hwexc: Zur Zeit sind alle Fehler dieses Typs Fehler der Schwere `fatal_error`. Der Errorhandler gibt eine Fehlermeldung aus und bewirkt einen *Procedural Walkback*. Dies ist eine Ausgabe der *Procedure Return Addresses*, angefangen vom Ort des Fehlers bis hinauf zur äußersten Prozedurebene.

Obwohl üblicherweise Fehler der Schwere `fatal_error` zu einem Neustart des MOPS führen, gibt es hier eine Ausnahme. Trat ein Fehler in einer XSR auf, so muss nicht der gesamte MOPS erneut gestartet werden. Der Errorhandler löscht nur den zugehörigen, bereits suspendierten Dispatcher DInn, unter dessen Regie die fehlerbehaftete XSR läuft, und übernimmt die weitere Bearbeitung der

XSR, indem er dem Sendepaket eine entsprechende Fehlermeldung im XSR-Status an den Aufrufer[14] mitgibt. Weitere, noch unbearbeitete Aufträge, die sich noch im Auftragspaket befinden, können weder bearbeitet noch mit einer Fehlermeldung beantwortet werden, sie gehen verloren.

Abschließend setzt der Errorhandler einen neuen Dispatcher auf, damit die ursprüngliche Anzahl von Dispatchern erhalten bleibt. Diese Aktion beeinträchtigt den MOPS als laufendes Gesamtsystem nicht und ist für den Benutzer, mit obiger Ausnahme, transparent.

err_sys: Systemfehler von `pSOS+` und den anderen Systemmodulen werden noch nicht von MOPS abgefangen Sie sollten in `pROBE` landen und dort behandelt werden.

Aus diesem Grund dürften Fehler des Typs `err_sys` den Errorhandler *nicht* erreichen. Kommt es trotzdem dazu, wird einfach eine allgemeine Fehlermeldung ausgegeben und MOPS neu gestartet.

err_pas: Folgende, vom Pascalcompiler generierten Laufzeitfehler werden vom EHDL behandelt:

- Array index out of range
- Attempt to dereference a NIL-pointer
- Case value unaccounted for
- Library error
- I/O error

EHDL gibt den Typ des Fehlers und, wo es möglich ist, die den Fehler beschreibenden Parameter aus.

Alle Pascalfehler haben die Schwere `fatal_error` und führen üblicherweise zu einem Restart des GuP. Jedoch gibt es auch hier die Ausnahme, wenn der Fehler in einer XSR auftrat. Diese ist unter **err_hwexc** beschrieben.

err_mops: Tritt ein Fehler dieses Typs auf, gibt EHDL zunächst die entsprechende Fehlermeldung aus.

Bei einem `minor_error` sind damit die Aktionen von EHDL beendet. MOPS setzt seinen Normalbetrieb ohne weitere Fehlerbehandlung fort.

Bei einem `fatal_error` wird ein Neustart des MOPS mit dem Grund (Restart Reason) `gup_error` durchgeführt.

Weder `minor_error` noch `fatal_error` bewirken eine dem Fehler entsprechende Meldung an den Auftraggeber. Bei einem `minor_error` ist dafür das fehlermeldende Modul selbst verantwortlich. Bei einem `fatal_error` ist das in den meisten Fällen sowieso nicht mehr möglich. Der Auftraggeber erhält „nur“ einen Timeout-Fehler vom Userface, da kein Antwortpaket mehr geschickt wurde.

otherwise: Andere Fehlertypen als die oben genannten dürfen den Errorhandler nicht erreichen. Geschieht dies trotzdem, wird eine Meldung und die falsche Typnummer ausgegeben.

13.5.4 Der Weg einer Exception

Hier soll anhand eines Beispiels der Weg einer Exception von ihrem Auftreten bis zum Beheben aufgezeigt werden. Gewählt wurde die »Divide by Zero«-Exception, die von einer `DIVx`-Instruktion des Prozessors generiert wird, wenn der Versuch gemacht wird durch Null zu teilen.
Zunächst muss der entsprechende Vektor in der Vektortabelle des Prozessors mit der entsprechenden Behandlungsroutine (`zero_div_vector` mit `p_zerdiv`) gesetzt werden. Das geschieht in der Initialisierungsphase der Errorhandlers EHDL (in `MOPS68:EHDL.PAS`).
Tritt nun eine Zero Divide Exception auf, wird der entsprechende Vektor aus der Vektortabelle geholt und die Routine `p_zerdiv` angesprungen (in `XLIB$DIR:ERROR.ASM`). Diese legt zunächst den entsprechenden Vectoroffset auf den Stack, der von `exc_asm` für die Behandlung der Exceptions des 68000 gebraucht wird,

[14] Gemeint ist der Auftraggeber. Also etwa ein Operatingprogramm oder NODAL.

und ruft anschließend den allgemeinen Handler exc_asm auf. Dieser ist für die weitere Behandlung aller Exceptions verantwortlich.
Der Handler exc_asm (in XLIB$DIR:ERROR.ASM) rettet die gesamte Umgebung der Exception. Dies sind die Register D0 bis D7 und A0 bis A6, die fehlerhafte Adresse bei einem Bus- oder Adressfehler, der Vectoroffset, der Program Counter an dem die Exception passierte und das zugehörige Statusregister.
Da die Prozessoren 68000 und 68020 verschiedene Exception Frames generieren, muss die Behandlung prozessorspezifisch geschehen. Aus diesem Grund wird vor dem Aufruf des allgemeinen Handlers der Vectoroffset auf den Stack gepushed. Im Exception Frame des 68000 gibt es darüber keine Information.
Diese Informationen werden von exc_asm so auf den Stack gelegt, dass die aufgerufene Prozedur hw_error sie als Parameter weiterverarbeiten kann.
Die Prozedur hw_error (in XLIB$DIR:EXCEPTIONHANDLER.PAS) baut nun die Message an den Errorhandler EHDL zusammen. Der Taskname wird im Taskregister 0 der fehlerhaften Task erwartet. Da eine Message maximal 16 Bytes lang sein kann, ist ein Errordescriptor notwendig, der weitere Informationen über den aufgetretenen Fehler enthält. Ein Pointer in der Message selbst zeigt auf diesen Descriptor. Der Descriptor ist ein Puffer, der von hw_error von der Partition PERR angefordert wird und der vom jeweiligen Errorhandler wieder freigegeben werden muss. Die fertige Message wird anschließend über die Messagequeue QERR an den EHDL geschickt. Als letzte Aktion wird die fehlerhafte Task vorläufig suspendiert, das heißt sie kann nicht mehr weiterlaufen. ISRs können nicht suspendiert werden. EHDL entscheidet über das weitere Vorgehen.
hw_error setzt voraus, dass PERR und QERR existieren. Gehen die Systemaufrufe pt_ident, pt_getbuf, q_ident oder q_send schief, wird der gesamte MOPS angehalten und auf einen Reset des Watchdog gewartet.
Der gesamte bisher beschriebene Ablauf, beginnend mit der Exception, fand noch im Kontext der fehlerhaften Task statt. Aus diesem Grund ist es möglich, leicht den Tasknamen zu erhalten und die Task zu suspendieren.
Nun wird EHDL (in MOPS68:EHDL.PAS) die running Task. Sie empfängt die Message und entscheidet anhand dieser Message über das weitere Vorgehen. Bei einer Hardware Exception wird zunächst die entsprechende Fehlermeldung und ein Procedural Walkback ausgegeben. Anschließend wird getestet, ob der Fehler in einer XSR auftrat. War das nicht der Fall, wird MOPS neu gestartet (restart).
Die Prozedur XSRError (in MOPS68:EHDL.PAS) entscheidet, ob der Fehler innerhalb einer XSR passierte. Die Antwort lautet ja, wenn die fehlerhafte Task ein Dispatcher war und wenn im Kontext *dieses* Dispatchers gerade eine XSR lief. In diesem Fall schickt die Prozedur XSRErrorHandling eine entsprechende Fehlermeldung an den Auftraggeber, löscht den fehlerhaften und bereits suspendierten Dispatcher und setzt einen neuen auf.
Das Beispiel ist hier nur in sehr verkürzter Form mit den wichtigsten Teilen dargestellt. Sie sollten sich aber ohne Probleme in den entsprechenden Quelldateien wiederfinden lassen.
Die Objekte der Dateien ERROR.ASM, EXCEPTIONHANDLER.PAS und PASCALERROR.PAS – alle aus der Directory XLIB$ERROR – sind Teil der MOPS-Library. Näheres dazu findet sich im Abschnitt on page 55.

13.6 Eigenständige Errorhandler

Neben dem MOPS-eigenen Errorhandler EHDL und pROBE+ gibt es noch zwei weitere Handler, die für ganz spezielle Fehler der Hardware ausgelegt sind.

13.6.1 Der AC-Fail Handler

Der AC-Fail Handler – der eigentlich ein DC-Fail Handler ist (siehe HW-Beschreibung, nicht in diesem Dokument) – ist für die Behandlung des AC-Fail-Interrupts verantwortlich. Er hat zwei Aufgaben. Zunächst werden die Interrupts in einem reset-festen Zähler gezählt. Anschließend wird ein STOP-Befehl ausgeführt, um MOPS in einen sicheren Zustand zu bringen, während die Power in die Knie geht. Aus diesem Zustand kommt der Prozessor nur heraus durch einen Interrupt oder durch einen Reset, zum Beispiel beim Powerup.

Der AC-Fail Handler wurde als eigenständiger Handler realisiert, da er erstens sehr schnell reagieren muss – alle Aktionen müssen beendet sein bevor die Power einen kritischen Wert unterschreitet – und zweitens kein „normales“ Errorhandling mehr sinnvoll ist, da die Power mit ziemlicher Sicherheit sowieso gleich weggeht.

13.6.2 Der Spurious Interrupt Handler

Der Spurious Interrupt Handler wurde notwendig, da der 68155 Interrupt Handler (ein IC) auf dem Gruppenmikro nicht fehlerfrei arbeitet. Dies führt bei hohen Interruptraten dazu, dass der IC sporadisch falsche Interrupts (Spurious Interrupts) generiert. Diese Exception wurde nur von `pROBE` behandelt und MOPS musste mit dem Watchdog neu gestartet werden, was jedesmal zu Ausfallzeiten (offline) von im Schnitt 15 bis 20 Sekunden führte.
Der Spurious Interrupt Handler fängt nun diese Exception ab und tut nichts anderes als die falschen Interrupts in einem reset-festen Zähler mitzuzählen.

13.7 Watchdog

Eine Rückfallposition für alle ernsten Fehler, die zur Laufzeit auftreten, ist der *Watchdog*. Ein Watchdog ist eine Elektronik, die innerhalb eines definierten Zeitintervalls (MOPS: timeout = 25s) getriggert werden muss, um das Ablaufen des Zeitintervalls zu verhindern. Das Triggern führt zum erneuten Aufziehen der Uhr des Watchdog, sodass das Intervall von neuem beginnt. Wird der Watchdog innerhalb dieses Intervalls nicht in der beschriebenen Weise bedient (wieder aufgezogen), löst er einen VME-Bus-Systemreset aus, der zum Neustart von MOPS führt. So führen zum Beispiel Fehler, deren Behandlung in `pROBE+` enden, nach spätestens 25s zu einem Neustart des MOPS. Das System wird damit immer wieder in einen laufenden Zustand gebracht[15]. Die Triggerung des Watchdogs ist in der Task ROOT[16] implementiert.

14 Systemparameter und Tuning

Der Betriebssystemkern `pSOS+`, auf den MOPS aufbaut, hat als Multitaskingsystem einige einstellbare Parameter, die das Verhalten des Gesamtsystems erheblich beeinflussen. Der wesentliche ist die (Software-) Priorität, die den einzelnen Tasks zugeordnet wird. Für das Verhalten bei Überlastungen und Fehlern spielen die Größen der Stacks und die Kapazitäten der Messagequeues eine wichtige Rolle.
Direkt als MOPS-Parameter ist die Anzahl der Dispatcher (DInn) einstellbar.
Die Hardware-Priorität der Interrupts hat ebenfalls einen Einfluss auf das Systemverhalten, obwohl sie, wegen der sehr kurzen Laufzeiten der ISRs, nicht so relevant ist wie die Software-Priorität der Tasks.

14.1 Die Rolle der Priorität

Scheduling ist die Zuteilung der Ressource CPU an eine der um sie konkurrierenden Tasks. Eine Task begehrt dann die CPU, wenn sie im `ready`-Zustand ist. Sind mehrere Tasks `ready`, erhält die Task mit der höchsten Priorität die CPU. Tasks gleicher Priorität im `ready`-Zustand werden in eine Queue eingereiht. Beim Scheduling bekommt die Task am Kopf der Queue die CPU.
Ein Scheduling findet bei jedem System Service Call statt, also immer dann, wenn `pSOS+` die Programmkontrolle hat. Da eine Task nicht notwendigerweise Service Calls benutzen muss, sondern z. B. langandauernde Arithmetik betreiben kann, gibt es einen *besonderen* System Service, der das Scheduling triggert:

[15] Das ist natürlich nur für seltene, sporadisch auftretende Fehler sinnvoll. Ein periodisch sich neustartendes Systems ist praktisch von geringem Wert.

[16] Vielleicht ist es sinnvoll, die Bedienung des Watchdogs (wieder) als periodischen SSR-Auftrag zu implementieren. Da bei der Abwicklung periodischer Aufträge einige wesentliche Komponenten von MOPS beteiligt sind, ist die ordungsgemäße Bedienung des Watchdogs ein gutes Indiz für das Funktionieren des Systems. Verklemmungen in der Auftragsabwicklung, die durch ein mangelhaftes Errorhandling verursacht wurden, könnten dadurch erkannt und beseitigt werden. Allerdings muss dann darüber nachgedacht werden, wie es sichergestellt werden kann, dass innerhalb des Watchdog Timeout ein Dispatcher zur Verfügung steht, der die entsprechende SSR zum Triggern des Watchdog aufruft. In der Vergangenheit gab es mit sehr langen „busy“-Zeiten der Dispatcher schon Probleme!

I_RETURN, der in Abschnitt on page 42 schon erwähnte Ausgang aus einer ISR. Da die Timer-ISR in MOPS mit der Frequenz von 50 bzw. 100 Hertz getriggert wird, ist mindestens alle 20 bzw. 10 Millisekunden ein Scheduling sichergestellt.
Damit arbeiten die – zur Zeit 7 – Dispatcher des MOPS in einer Art *Time Slicing*-Betrieb. Alle Dispatcher haben die gleiche Priorität. Sind mehrere Dispatcher ready und einer running und wird der running Dispatcher von der Timer-ISR unterbrochen, so wird durch das I_RETURN der Timer-ISR der running Dispatcher hinten in die ready-Queue eingehängt und der nächste Dispatcher, der ready ist und nun am Kopf der Queue steht wird zum running Dispatcher (siehe dazu auch Abschnitt on the current page).
Der Software-Priorität übergeordnet ist die Hardware-Priorität der Interrupts. MOPS-Tasks laufen mit der Hardware-Priorität 0, werden also von allen Interrupts unterbrochen. ISRs selbst können nur von einem Interrupt höherer Priorität unterbrochen werden.

14.1.1 Die Hardware-Priorität der Interrupts

Da die Zeit eine herausragende Rolle in einem Realtime-System hat, läuft die Timer-ISR (tim_isr), die die pSOS+-Systemzeit versorgt, mit der höchsten Priorität aller ISRs.
Die recv_isr und die xend_isr triggern die MOPS-Tasks RECV und XEND, die zusammen mit dem CP die Verwaltung der Empfangs- und Sendepakete durchführen. Diese beiden ISRs laufen mit der nächst niedrigeren Priorität.
Auf gleicher, nächst niedrigerer Priorität liegen die Interrupts von der SE und die Interrupts für die Terminalein- und Ausgabe.
Die genauen Interruptlevel sollen hier nicht angegeben werden, da sie sich je nach Hardware (MP1001, FIC8230) unterscheiden. Selbst die Reihenfolge der Prioritäten stimmt nicht in jedem Fall überein. So hat der Abort-Interrupt einmal eine höhere Priorität als der Timer-Interrupt und einmal eine niedrigere. Die exakten Level und Zuordnungen können den entsprechenden Manuals der Hardwarehersteller entnommen werden (siehe [9] und [1]).

14.1.2 Die Software-Priorität der Tasks

Die höchste Priorität hat die ROOT-Task. Sie ist von pSOS+ beim Startup vorgegeben und wird nicht verändert.
Es folgt der Errorhandler EHDL, weil es sinnvoll ist, dass eine Fehlerbehandlung so schnell wie möglich durchgeführt wird.
Das Laufzeitsystem von MOPS besteht aus den Kommunikationstasks RECV und XEND, den Konnektierungsverwaltern PERI und RUPT, den Tasks zur Verwaltung von Alarmen ALRM und TIMR und den Anwendertasks, den Dispatchern DInn. Die Prioritäten unter diesen Tasks wurden nach dem Prinzip „*je näher zur Anwendung desto kleiner*“ zugeteilt. Das heißt, RECV und XEND haben – beide gleich – die höchste Priorität, dann folgen – wiederum beide gleich – PERI und RUPT, danach ALRM, TIMR und zum Schluss die Dispatcher DInn. Diese Reihenfolge ist notwendig, damit langandauernde Abläufe von Anwendungsprogrammen die Gesamtfunktionalität des Systems nicht beeinträchtigen oder gar unmöglich machen.
Es folgt die Task SHOW, die niedrige Priorität hat, da lokales Terminal-I/O für die eigentlichen Aufgaben des MOPS zweitrangig ist.
Die pSOS+-eigene Nulltask IDLE, die immer dann läuft, wenn keine andere Task die CPU begehrt, hat erwartungsgemäß die niedrigste Priorität, die ebenfalls von pSOS+ vorgegeben ist.
Die Prioritäten der Tasks sind in Tabelle on the facing page zusammengefaßt.

14.2 Die Bedeutung der Anzahl der Dispatcher

Wie im Abschnitt on page 29 beschrieben, gibt es in MOPS mehrere Dispatcher, um Anwenderprogramme quasi-parallel ablaufen lassen zu können. Aufgrund des Schedulingmechanismus von pSOS+ gibt es einen

Task	Priorität
ROOT	255
EHDL	220
RECV	105
XEND	105
PERI	103
RUPT	103
ALRM	102
TIMR	101
DINN	100
SHOW	90
IDLE	0

Tabelle 1: Die Prioritäten der Tasks

Konflikt zwischen der Existenz mehrerer Dispatcher und der Forderung, dass die Anwendungen (XSRs) genau in der Reihenfolge ablaufen, wie sie vom CP empfangen und an MOPS übergeben werden.
Eine `running` Task kann jederzeit durch eine ISR unterbrochen werden. Der Systemaufruf `I_RETURN` am Ende der ISR führt zum Scheduling durch `pSOS+`, was dazu führen kann, dass ein *anderer* als der unterbrochene Dispatcher zum `running` Dispatcher wird (siehe oben). Dieses Überholfenster kann nicht vollständig geschlossen werden[17]. Es gibt drei Lösungen dieses Problems:

1. Jeder Auftrag kann durch eine Quittung bestätigt werden. Darf ein Auftrag B auf keinen Fall einen Auftrag A überholen, so darf B erst abgeschickt werden, wenn die Quittung für A eingetroffen ist.

2. Ein Dispatcher ist immer zuständig für ein Auftrags*paket*. Ein Auftragspaket kann mehrere Aufträge enthalten. Diese Aufträge werden vom Dispatcher streng sequenziell bearbeitet. Steht Auftrag A an erster Stelle und Auftag B an zweiter Stelle im Paket, wird auf jeden Fall Auftrag A *vor* Auftrag B ausgeführt. Allerdings wird Auftrag B auf jeden Fall[18] und unabhängig vom Ergebnis des Auftrags A ausgeführt. In diesem Sinn sind die Aufträge eines Paketes voneinander unabhängig.

3. Für zeitkritischere Anwendungen oder für asynchrone Aufträge, die eine strenge Sequentialität der Auftragsabarbeitung erfordern, darf es nur einen Dispatcher in MOPS geben. Ein Beispiel dafür ist das unter MOPS implementierte Grafiksystem, das wegen aufeinanderbezogener Schreib- und Löschaktionen diese Sequentialität braucht. In diesem Zusammenhang ist nur ein Dispatcher kein Verlust an Leistungsfähigkeit, da die geforderte Sequentialität die Quasi-Parallelität des MOPS (durch mehrere Dispatcher) sowieso ausschließt.

   Falls dies notwendig wird, kann die Anzahl der Dispatcher, die beim Hochfahren von MOPS kreiert und gestartet werden, als Zahl in der anwendungsbezogenen, lokalen Datenbasis im GuP abgelegt werden.

14.3 Die Größe der Stacks

Durch unterschiedliche Stackgrößen kann die Leistungsfähigkeit des Gesamtsystems nicht beeinflusst werden, ein ausreichend großer Stack für *worst case*-Situationen ist aber die unbedingte Voraussetzung für die Funkionsfähigkeit des Systems überhaupt. Bei der Entwicklung von MOPS wurden die Stacks durch grobe Abschätzungen vermeintlich großzügig dimensioniert. Eine ganze Reihe von dubiosen Fehlern, die bei

[17] In älteren Versionen von MOPS wurde dies mit Hilfe der expliziten Erhöhung der Priorität des `running` Dispatchers versucht. Einzelheiten dazu sind in der 1. Auflage dieses Dokumentes unter der gleichen Überschrift zu finden.

[18] Es sei denn, Auftrag A führt zu einem fatalen Fehler und zum anschließenden Löschen des Dispatchers. Siehe dazu die Behandlung des Fehlertyps `err_hwexc` in Abschnitt on page 46.

den Systemtests auftraten, konnten (nach oft mühevollen und langwierigen Irrwegen) schlicht auf einen zu klein dimensionierten Stack, durch den andere, fremde Stacks überschrieben wurden, zurückgeführt werden. Durch eine Vergrößerung des entsprechenden Stacks war der jeweilige Fehler verschwunden. Welche Größe notwendig ist, ist nur empirisch zu bestimmen. Auf dem Hintergrund dieser Erfahrungen sollte auf jeden Fall beim Auftreten von schwer zu diagnostizierbaren Fehlern eine solche Stackverletzung in Betracht gezogen werden.

14.4 Die Kapazitäten der Messagequeues

In der `pSOS+`-Konfigurationstabelle ist die maximale Anzahl von Messages festgelegt, die sich insgesamt im System in Messagequeues befinden können. Beim Kreieren von Messagequeues wird ebenfalls ein Maximum von Messages festgelegt, die in diese Queue eingereiht werden können. Die beiden Größen sind in der Weise einander angepasst, dass das Maximum für eine Messagequeue um einiges kleiner als das Gesamtmaximum. Dies verhindert, dass ein Überlauf an einer Queue die prinzipielle Funktionalität der anderen beeinträchtigt.
Sind allerdings zwei Queues an einer solchen Fehlfunktion beteiligt, so kann es leicht geschehen, dass das Gesamtmaximum an eingereihten Messages erreicht wird. Damit käme MOPS eigentlich zum Stillstand.
Um in dieser Situation wenigstens noch ein vernünftiges Errorhandling zu gewährleisten, ist die Errormessagequeue QERR mit sogenannten *Private Buffers* ausgestattet. Diese werden *ausschließlich* von QERR genutzt. Sie kommen nicht aus dem Systempool und können nicht von Messages, die für andere Queues bestimmt sind, belegt werden.

15 Code Management und Generierung

15.1 Dateiorganisation

Die gesamte VME-Systemsoftware, also MOPS, CP, ECM, MIL-Treiber und die Benutzerbibliothek, ist unter der Root Directory `SIS$ROOT:[SYSVME]` untergebracht. Die einzelnen Subdirectories, die in Abbildung on the next page im Überblick dargestellt sind, sollen hier kurz beschrieben werden.

CMREF: Hier liegen die Reference Copies der CMS-Bibliothek.

CMS: Das ist die Root Directory der CMS-Bibliothek.

COM: Oft genutzte Command Files, die nicht jedesmal, wenn sie gebraucht werden, aus der CMS-Bibliothek geholt werden sollen, liegen auf dieser Directory. Aber Vorsicht! Viele Command Files sind ebenfalls versionsabhängig, das heißt, sie gelten nur zur Generierung bestimmter Versionen eines Systemmoduls. Sicherheitshalber holt man das passende Command File ebenfalls aus der CMS-Bibliothek.

DOC: Alle Dokumente, Beschreibungen, Fehlerreports usw. liegen auf dieser Directory. Command Files, die zur Generierung der Incode Documentation dienen, liegen nicht hier, sondern auf der Directory des entsprechenden Moduls.

ECM: Dieses Directory enthält alle Module des ECM.

ETH: Dieses Directory enthält alle Module des CP.

GEN: Zur Generierung älterer Versionen ist es nötig, eigene Directories zu haben, da die Quellen nicht auf die normalen Directories gefetched werden können, wenn dort bereits in der Weiterentwicklung befindliche Module liegen. Aus diesem Grund existiert unter dieser Directory ein weiterer Baum, der nochmals die Subdirectories ECM, ETH, LIB, MIL, MOPS und MOPSLIB enthält.

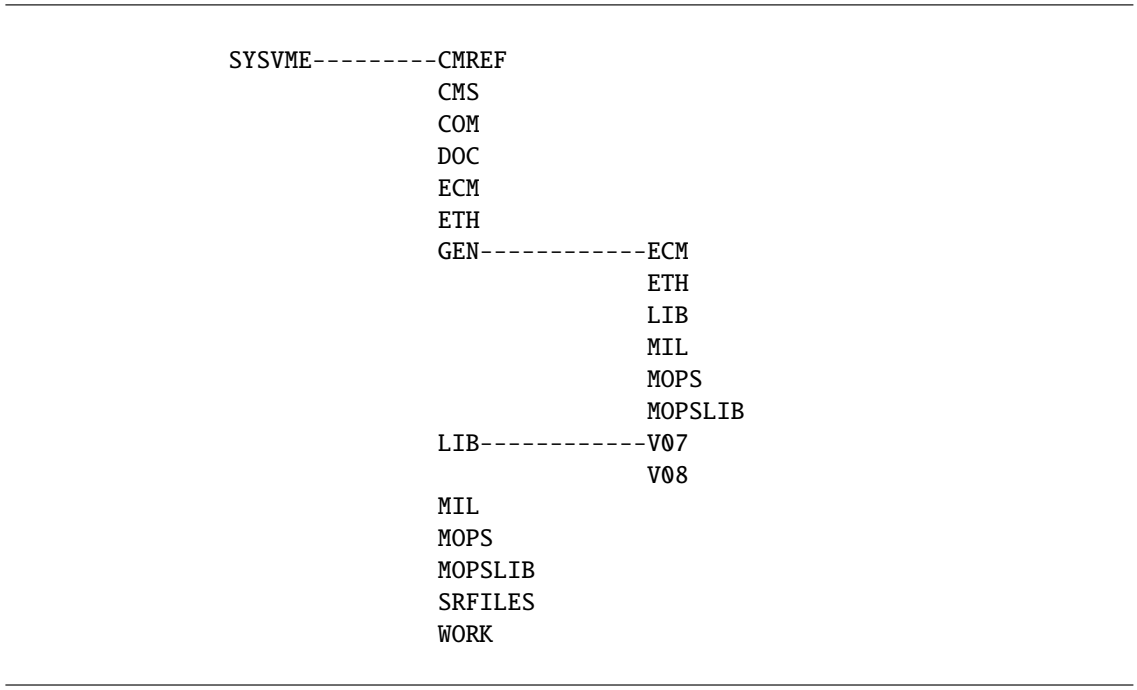

Abbildung 14: Dateiorganisation der VME-Systemsoftware

LIB: Alle Quellen und Objekte, die modulübergreifend genutzt werden, liegen unter dieser Directory. Da LIB selbst nur zu Zeiten der Weiterentwicklung eines Systemmoduls Dateien enthält und ansonsten leer ist, und da Benutzer (Gerätesoftwareentwickler) ständig Zugriff auf die systemversionsabhängige Bibliothek haben müssen, gibt es die Subdirectories V07, V08 usw. V07 ist die Bibliothek für die Systemversion 7. Jedes Systemmodul mit einer Versionsnummer 07.xx passt zu Gerätesoftware, die mit dieser Version der Bibliothek generiert wurde.

Fragen der Versionskennung, der Kompatibilität und der aufwärtskompatiblen Änderungen in der Bibliothek sind in [3] genau behandelt.

MIL: Hier liegen die verschiedenen MIL-Treiber und deren Definitionsmodule.

MOPS: Dieses Directory enthält alle Tasks des MOPS, die System Service Routinen für MOPS und ECM, sowie alle, ausschließlich MOPS betreffenden Definitionsmodule.

MOPSLIB: Dieses Directory enthält alle Module, die die MOPS Runtime Library bilden.

SRFILES: Hier liegen die fertigen Systemmodule MOPS, CP und ECM im Motorola-Hexformat.

WORK: Dies ist eine Arbeitsdirectory für alle möglichen Zwecke.

Dargestellt und beschrieben sind hier nur die notwendigen und wichtigen Directories. Einige von diesen können weitere Subdirectories für bestimmte temporäre oder andere Zwecke haben.
Abbildung 14 zeigt, dass einige Directories doppelt vorhanden sind (`[.MOPS]`, `[.GEN.MOPS]`). Die Bibliothek (`[.LIB]`) hat versionsabhängige Subdirectories (z. B. `[.LIB.V07]`). Je nach Aufgabe liegen gleiche Dateien einmal auf dem einen, einmal auf einem anderen Directory. So wird zum Beispiel die Datei `GENDATATYPE.PIN`, das sowohl von MOPS-Modulen als auch von Gerätesoftware benötigt wird, bei der Weiterentwicklung des MOPS auf `[.LIB]` erwartet, bei der Generierung einer älteren Version des MOPS auf `[.GEN.LIB]` und bei der Generierung einer Gerätesoftware für Systemversion 7 auf `[.LIB.V07]`. Aus

diesem Grund wurden logische Namen definiert, die in den includierenden Modulen benutzt werden und die je nach Aufgabe unterschiedlich definiert werden. Diese sind in Tabelle on the current page zusammengefasst.

Log. Name	SYSVME	SYSVMEGEN / SYS68Vss
CMREF68	SIS$ROOT:[SYSVME.CMREF]	SIS$ROOT:[SYSVME.CMREF]
CMS68	SIS$ROOT:[SYSVME.CMS]	SIS$ROOT:[SYSVME.CMS]
COM68	SIS$ROOT:[SYSVME.COM]	SIS$ROOT:[SYSVME.COM]
DOC68	SIS$ROOT:[SYSVME.DOC]	SIS$ROOT:[SYSVME.DOC]
ECM68	SIS$ROOT:[SYSVME.ECM]	SIS$ROOT:[SYSVME.GEN.ECM]
ETH68	SIS$ROOT:[SYSVME.ETH]	SIS$ROOT:[SYSVME.GEN.ETH]
GEN68	SIS$ROOT:[SYSVME.GEN]	SIS$ROOT:[SYSVME.GEN]
LIB68	SIS$ROOT:[SYSVME.LIB]	SIS$ROOT:[SYSVME.GEN.LIB] / SIS$ROOT:[SYSVME.LIB.Vss]
MIL68	SIS$ROOT:[SYSVME.MIL]	SIS$ROOT:[SYSVME.GEN.MIL]
MOPS68	SIS$ROOT:[SYSVME.MOPS]	SIS$ROOT:[SYSVME.GEN.MOPS]
MOPSLIB68	SIS$ROOT:[SYSVME.MOPSLIB]	SIS$ROOT:[SYSVME.GEN.MOPSLIB]
SYS68	SIS$ROOT:[SYSVME.SRFILES]	SIS$ROOT:[SYSVME.SRFILES]
WORK68	SIS$ROOT:[SYSVME.WORK]	SIS$ROOT:[SYSVME.WORK]

Tabelle 2: Logische Namen der Directories der VME-Systemsoftware

Die logischen Namen werden mit Hilfe der bekannten Kommandos `SYSVME` für die Weiterentwicklung von MOPS, `SYSVMEGEN` für die Generierung einer fertigen Version des MOPS, oder mit `SYS68Vss` für die Generierung von Gerätesoftware, die kompatibel mit der Systemversion ss ist, definiert. `SYS68Vss` stellt nicht alle logischen Namen zur Verfügung, um zu vermeiden, dass Gerätesoftware Dateien von falschen Directories benutzt.
Einige Definitionsmodule enthalten Festlegungen, die hardwareabhängig sind. Deshalb gibt es für jede Hardware jeweils eigene Definitionsmodule, die sich im Dateinamen durch ein vorangestelltes `F$` bzw. `M$` unterscheiden. Für diese Module gibt es ebenfalls logische Namen, die von den oben genannten Kommandos definiert werden. Diese Kommandos definieren auch das globale Symbol `cpuvers`, das die Hardwarekennung (`F|` bzw. `M`) enthält. Die logischen Namen für hardwareabhängige Definitionsmodule sind in Tabelle on this page zusammengefasst.

Logischer Name	Definitionsmodul
TASKS	MOPS68:<cpuvers>$TASKS.PIN
MOPSASMCONSTANTS	MOPS68:<cpuvers>ASMCONSTANTS.AIN
MOPSPASCONSTANTS	MOPS68:<cpuvers>PASCONSTANTS.PIN

Tabelle 3: Logische Namen für hardwareabhängige Definitionsmodule

Vorsicht ist geboten bei MOPSADRCONFIG. Dieses Modul definiert die Adressen der beiden Konfigurationstabellen des MOPS für *ganz spezielle* Gerätesoftware *noch einmal*. Es wurde einzig aus dem Grund kreiert, um zu vermeiden, dass Gerätesoftware (ursprünglich T85$USRS) einen Wust von Definitionsmodulen includieren muss, nur um an diese beiden Adressen zu kommen.

15.2 CMS-Bibliothek

Alle Quelldateien von MOPS, CP, ECM, den MIL-Treibern und den zugehörigen Bibliotheken sind in einer CMS-Bibliothek zusammengefasst. Dies erlaubt eine konsistente Versionsverwaltung und bietet die

Möglichkeit jederzeit kompatible Versionen der Systemmodule zu generieren. Die Struktur der CMS-Bibliothek mit ihren Elementen Gruppen und Klassen spiegelt die Module der Systemsoftware weitgehend wider.
Die CMS-Bibliothek ist in [3] ausführlich behandelt. Dort wird Bezug genommen auf eine ältere Bibliothek, die auf dem Entwicklungssystem von Oregon beruht. Für das Entwicklungssystem Organon der Firma CAD-UL ist eine eigene CMS-Bibliothek eingerichtet worden, die aber in ihrer Struktur der alten genau entspricht.

15.3 Systemgenerierung

MOPS kann betrachtet werden als ein System, dass sich aus mehreren, relativ unabhängigen Teilsystemen zusammensetzt. Diese Teilsysteme können zwar nicht unabhängig, aber doch getrennt voneinander generiert werden. Das Ergebnis der Generierung der Teilssyteme sind Dateien im S-Record-Format, die zusammengenommen (in eine Datei kopiert) das Gesamtsystem MOPS bilden.
Grob betrachtet gibt es zunächst drei Teilsysteme:

SYSTEM: Mit SYSTEM ist hier die grundlegende Systemumgebung gemeint, auf der MOPS basiert. Das ist zunächst die Hardwareanpassung, die den Bootcode und die Treiber, u. a. für die Terminalein- und -Ausgabe, enthält. Es folgt der Realtime Multitasking Kernel pSOS+ mit seinen Komponenten pROBE+ und pREPC+. Außerdem enthalten sind die für eine Runtime Library notwendigen Bibliotheken. Diese bestehen aus der Anpassung für die pSOS+ Service Calls, der Anpassung für die pREPC+ Service Calls, der Pascal- und der Mathematikbibliothek sowie der entsprechenden Sprungtabelle für die Runtime Library. Weiterhin enthalten sind die VME-systemweit einheitlichen Fehlerbehandlungsroutinen.

MOPSLIB: Dieses Teilsystem bildet hauptsächlich die zweite Runtime Library im Gesamtsystem MOPS. Sie stellt kontrollsystemnahe Dienste zur Verfügung, die von MOPS selbst als auch von SSRs und USRs benutzt werden. Auch diese Runtime Library ist über eine entsprechende Sprungtabelle zu erreichen, die Teil der Bibliothek ist.

MOPS: Hier sind alle Tasks des MOPS, die System Service Routinen für MOPS, dessen Gerätemodell ECMS und für das Gerätemodell EC vereint. Außerdem enthalten ist das Modul Roothead, das für die Initialisierung und den Urstart des Hochsprachenteils des Systems (Task ROOT) verantwortlich ist.

Im Gegensatz zu den beiden anderen Teilsystemen, die als Ganzes generiert (gelinkt) werden, zerfällt das Teilsystem MOPS nochmals in einzelne Module. Getrennt voneinander generiert werden die einzelnen Tasks, die SSRs und Roothead.

```
% <text>
% * Memory Layout
%       - alter/neuer GuP
%       - welche Adressen muessen fest definiert sein (ROOT Start,
%         Sprungtabellen, SSR-Start, USR-Start)
%       - Wo sind die Adressen definiert (COM-Files, PIN-Files, ...)
% * COM-Files
%       - @MAKE... ohne, @GENERATE... mit CMS-Lib
%       - Files zum Assemblieren von pSOS+,... und zum Generieren
%         der Sprungtabellen
%       - Reihenfolge System, MOPSlib, MOPS
%       - Verbindung zwischen diesen 3 Teilen durch Sprungtabellen
%       - neue Extensions (*.LS = Mapfile, ...)
% * Compiler
%       - Globals A5-relativ
```

```
%       - F$ -> /cpu=68020, M$ -> /cpu=68000
% * Linker
%       - linken fuer A5-relativ und 100H Platz fuer Pascal-Lib
% * Taskbuilder
%       - Reihenfolge der Tasks (wg. Startadresse: 'symfile *.bd')
%       - xlib$error:error.asm usw. => Mit MOPSLIB linken =>
%         MOPSLIB.BD beim Taskbuilden fuer EHDL
% * BTOS
```

15.4 Cross Library

```
VMECROSS-------COMMON
               ERROR
               PASCAL
               PSOS
```

Abbildung 15: Dateiorganisation der Cross Library

```
"XLIB$DIR"    = "SIS$ROOT:[VMECROSS]"
"XLIB$COMMON" = "SIS$ROOT:[VMECROSS.COMMON]"
"XLIB$ERROR"  = "SIS$ROOT:[VMECROSS.ERROR]"
"XLIB$PASCAL" = "SIS$ROOT:[VMECROSS.PASCAL]"
"XLIB$PSOS"   = "SIS$ROOT:[VMECROSS.PSOS]"
```

Tabelle 4: Logische Namen der Directories der Cross Library

15.5 Globale Variable der Pascal- und Mathematikbibliotheken

Eine besondere Schwierigkeit in Bezug auf globale Variablen der Bibliotheken ergibt sich im Zusammenhang mit USRs, die solche Variablen benutzen, zum Beispiel, wenn sie auf den Bildschirm schreiben. USRs laufen im Kontext eines Dispatchers. Sie sind nichts anderes als Prozeduren, die vom Dispatcher aufgerufen werden. Damit benutzen sie die gleichen globalen Variablen der Bibliothek wie der Dispatcher selbst.
Globale Variablen der Pascal- und Mathematikbibliothek liegen in den ersten 100_{Hex} relativ zu Register `A5`. Die Offsets der einzelnen Variablen werden zur Linkzeit des Moduls festgelegt. Dann heißt z. B. `output_` auf `0x0000003c`, dass die Variable `output_` auf $3c_{Hex}$ relativ zum aktuellen `A5` (Assembler: `3c(A5)`) zu finden ist.
Da aber USRs völlig getrennt von den Dispatchern generiert werden, ist zunächst nicht garantiert, dass die gleichen Variablen auf der gleichen Adresse liegen, also die gleichen Offsets haben. Damit könnten sich USRs und Dispatcher gegenseitig Variablen Überschreiben. Um dies zu vermeiden wurde das Modul `XLIB$GBLVAR` kreiert.
Gleiche Offsets der globalen Variablen von USRs und Dispatchern werden nun wie folgt garantiert:

1. Bei der Generierung von `<CPU-Typ>$SYSTEM` werden zur Linkzeit die Offsets aller globalen Variablen der Pascal- und Mathematikbibliothek festgelegt. Die Generierung beinhaltet die Erstellung der

system-spezifischen Laufzeitbibliothek (siehe Kapitel on page 9 und on page 16). Dabei werden alle Module der Pascal- und Mathematikbibliothek referiert und damit die globalen Variablen, die von diesen benutzt werden.

2. Beim Linken der Dispatcher werden die Referenzen auf die globalen Variablen aufgelöst, indem das Modul <CPU-Typ>$SYSTEM.BD als Symbolfile mit dazugelinkt wird.

3. Ein Modul XLIB$GBLVAR wird generiert. Dazu wird beim Linken in der richtigen Reihenfolge auf Module in der Pascal- und Mathematikbibliothek zugegriffen, so dass die globalen Variablen genauso gelinkt werden wie bei der Generierung des Moduls <CPU-Typ>$SYSTEM. Siehe dazu auch die Incode-Dokumentation in XLIB$GBLVAR.COM auf der Directory XLIB$DIR.

4. Beim Linken der USRs werden die Referenzen auf die globalen Variablen aufgelöst, indem das Modul XLIB$GBLVAR.BD als Symbolfile mit dazugelinkt wird.

Damit haben die globalen Variablen von USRs und Dispatchern die gleichen Offsets.
Möglich wäre auch, direkt das Modul <CPU-Typ>$SYSTEM.BD zu den USRs zu linken. Damit besteht aber die Gefahr, dass die USRs systemabhängig werden, wenn sie weitere Symbole aus dieser Datei linken, mit denen sie nichts zu tun haben sollten und die sich mit einer neueren Systemversion in ihrem Wert ändern.

16 Das Entwicklungssystem Organon

16.1 Dateiorganisation

```
ORGANON--------DISTHP---------LOCK
                              WORK
               DISTHPTCP------LOCK
               DISTPC---------LIB
                              LIBSRC
                              LOCK
                              ORIGINAL
               DISTPSOS
               DOC------------BUGS
                              EXAMPLES
               PASPSOS
               TMP
```

Abbildung 16: Organon Dateiorganisation

16.2 Pascal Compiler

```
% <text>
% Falls license exceeded: purge mwc$tmp
```

16.3 Hochsprachendebugger XDB

```
% <text>
% Nur was zu beachten ist wenn MOPS debugged wird.
```

```
"ORGA$DIR"    = "NODEC$ROOT:[ORGANON]"
"ORGA$PASCAL" = "NODEC$ROOT:[ORGANON.DISTPC]"
"ORGA$SRC"    = "NODEC$ROOT:[ORGANON.DISTPC.LIBSRC]"
"ORGA$LIB"    = "NODEC$ROOT:[ORGANON.DISTPC.LIB]"
"ORGA$XDB"    = "NODEC$ROOT:[ORGANON.DISTPSOS]"
"ORGA$XDBHP"  = "NODEC$ROOT:[ORGANON.DISTHPTCP]"
"ORGA$BUGS"   = "NODEC$ROOT:[ORGANON.DOC.BUGS]"

"ORGANON$PC68"      = "NODEC$ROOT:[ORGANON.DISTPC.]"
"ORGANON$PC68$LOCK" = "NODEC$ROOT:[ORGANON.DISTPC.LOCK]"
```

Tabelle 5: Logische Namen der Directories des Organon Entwicklungssystems

A Glossar

Alarm: Ein Alarm ist die Meldung über eine Zustandsänderung einer Komponente des Kontrollsystems. MOPS verarbeitet (und produziert eigene) Alarme auf der VME-Ebene.

Anwenderprogramm: Ein Programm, das mit Hilfe der Kontrollsystemschnittstellen und -Services (sowohl auf VMS als auch auf GuP und SE) beschleunigerbezogene Operationen ausführt.

Communication Processor: Der Ethernetcontroller inklusive der zugehörigen Software, der für MOPS die Netzwerkkommunikation abwickelt.

CP: siehe Communication Processor.

DInn: Einer der (z. Zt.) 7 Dispatcher in MOPS. Der erste heißt DI01, der letzte DI07.

Diskonnektierung: Die Auflösung einer Konnektierung.

Dualport-RAM: Ein RAM, das physikalisch auf einer I/O-Prozessorkarte (CP oder SE) liegt, und auf das von CP oder SE lokal und vom GuP global (über den VME-Bus) zugegriffen werden kann.

EC: Equipment Controller. Ein Synonym für SE und gleichzeitig die Bezeichnung des Gerätemodells für die SEs.

ECM: Equipment Control Monitor, die Systemsoftware, die auf den SEs läuft.

ECMS: Equipment Control Monitor System, die Bezeichnung des Gerätemodells für die Gruppenmikros.

EHDL: Die Errorhandlertask in MOPS.

Ethernet: Ein bekanntes und weitverbreitetes Local Area Network.

Event: Zwei Bedeutungen: 1.) Ein Event, mit dem sich Tasks gegenseitig informieren oder synchronisieren können. 2.) Ein Muster auf dem Timingbus des Kontrollsystems.

Exchange: In älteren `pSOS`-Versionen die Bezeichnung der Messagequeues.

GuP: Gruppenmikroprozessor, die Hardware, auf der das MOPS läuft.

Ident: Eine Kennung.

IDLE: Die Nulltask in MOPS.

ISR: Interrupt Service Routine.

Konnektieren: Das Herstellen einer asynchronen Verbindung zwischen einer Datenquelle auf VME-Ebene und einer Datensenke auf VMS-Ebene und das Verknüpfen der Datenquelle mit einem bestimmten äußeren Ereignis.

Konnektierter Auftrag: Eine an ein äußeres Ereignis geknüpfte Aktion des MOPS, die zu einer asynchronen Antwort an den konnektierten Prozess führt.

Konnektierung: Die asynchrone Verbindung zwischen Datenquelle und -Senke inklusive der Verknüpfung der Datenquelle mit einem äußeren Ereignis.

Mailbox: Synonym für Messagequeue.

Message: Ein Datenblock, der von einer Task über eine Messagequeue an eine andere Task geschickt wird.

Messagequeue: Eine Konstruktion, über die Daten (Messages) zwischen verschiedenen Tasks ausgetauscht werden können.

MOPS: Das Micro Operating System, ist die in diesem Handbuch beschriebene Systemsoftware auf dem GuP.

Multipaket: Ein Ethernetpaket dessen Datenbereich von 1450 auf 8700 Bytes vergrößert wurde. Die notwendige Zerlegung des Multipakets in via Ethernet transportable (Teil-) Pakete und die Zusammensetzung der (Teil-) Pakete zu einem Multipaket wird vom Ethernet Controller auf VME-Ebene und von Netman bzw. NetAccess auf VMS-Ebene vorgenommen.

Organon: Organon ist das Entwicklungssystems der Firma CAD-UL für die VME-Ebene. Es enthält neben Compiler, Assembler, Linker und weiteren Werkzeugen auch den Hochsprachendebugger XDB.

PERI: Die Task, die die periodischen Aufträge verwaltet.

pROBE+: Der zu pSOS+ gehörende Debugger/Monitor.

pREPC+: Eine pSOS+- und gleichzeitig C-nahe Laufzeitbibliothek.

Process: Synonym für Task.

pSOS+: Ein Realtimebetriebssystemkern. pSOS ist die Bezeichnung des alten, pSOS+ des neuen Systems. pSOS+ entspricht (weitgehend) dem POSIX-Standard.

QALR: Messagequeue an die Alarme geschickt werden und die von ALRM ausgelesen wird.

QERR: Messagequeue an die Fehlermeldungen geschickt werden und die von EHDL ausgelesen wird.

QPER: Messagequeue an die periodische Konnektierungen und Diskonnektierungen geschickt werden und die von PERI ausgelesen wird.

QRCV: Messagequeue an die RECV, PERI und RUPT Messages schicken und von der die Dispatcher (DInn) Messages empfangen.

QRUP: Messagequeue an die Interruptkonnektierungen und Diskonnektierungen geschickt werden und die von RUPT ausgelesen wird.

Queue: 1. Die Kurzbezeichnung für die MOPS-eigene periodische bzw. Interruptqueue. 2. Ein weiteres Synonym für Messagequeue.

RECV: Die Task zur Verwaltung der Empfangspakete.

ROOT: Die Basistask in MOPS.

RUPT: Die Task, die die interruptkonnektierten Aufträge verwaltet.

SCB: SSR Control Block, eine Kontrollblockstruktur für SSRs.

Scheduling: Die Zuteilung der Ressource CPU an darum konkurrierende Tasks.

SE: Steuereinheit. Die I/O-Prozessorkarten MP1050— und FIO8130 in den VME-Systemen. Englisch auch EC (Equipment Controller) genannt.

SHOW: Die Displaytask.

Signal: In älteren pSOS-Versionen die Bezeichnung der Events.

SSR: System Service Routinen sind MOPS-nahe Anwendungsprogramme (Prozeduren), die die Gerätemodelle ECMS und EC repräsentieren und die Dienstleistungen für MOPS zur Verfügung stellen.

System Service: Mit System Service werden in diesem Dokument zwei verschiedene Dinge beschrieben. Zum einen sind die System Services gemeint, die vom Betriebssystemkern pSOS+ zur Verfügung gestellt werden, zum anderen die System Services, die MOPS als Dienstleistung in Form von SSRs (siehe dort) oder als Routinen der MOPS-Library zur Verfügung stellt.

Task: Eine ablauffähige Einheit mit einer virtuellen, isolierten Umgebung, die durch den pSOS+-Kern vorgegeben wird. In dieser Umgebung konkurrieren die Tasks um die verfügbaren Ressourcen (CPU, Speicherplatz, I/O-Devices etc).

TIMR: Die Timertask, die die ALRM-Task triggert, damit diese wartende Alarme verschickt.

total_xcbs: Die Kontrollblockstruktur in MOPS, die alle SCBs und UCBs enthält.

UCB: USR Control Block, eine Kontrollblockstruktur für USRs.

USR: User Service Routine, ein Anwendungsprogramm (Prozedur), die unter MOPS läuft.

Userface: Eine Softwareschnittstelle zum Kontrollsystem für Anwenderprogramme auf der VMS-Ebene.

VME: Ein bekannter und weitverbreiteter Bus-Standard.

XCB: Der zusammenfassende Begriff für SCB und UCB.

XDB: Der High Level Language Cross Debugger ist Teil des Entwicklungssystems Organon der Firma CAD-UL.

XEND: Die Task zur Verwaltung der Sendepakete.

XSR: Der zusammenfassende Begriff für SSR und USR.

Literatur

[1] CES. Fast Intelligent Controller FIC8230 User's Manual. , Creative Electronic Systems, Petit-Lancy, Geneva, Switzerland, 1987.

[2] Ludwig Hechler. The Central Alarm Processor and its User Program Interface. Accelerator Controls Documentation U-CAP-01, Gesellschaft für Schwerionenforschung, Darmstadt, Dezember 1991. (Source: capaccess.tex).

[3] Ludwig Hechler. Organisation und Verwaltung der Software der VME-Ebene. Accelerator Controls Documentation O-SIS-09, Gesellschaft für Schwerionenforschung, Darmstadt, 1992. (Source: vmesworganisation.tex).

[4] Ludwig Hechler. Zur Realisierung eines Alarmprozessors. Accelerator Controls Documentation G-CAP-01, Gesellschaft für Schwerionenforschung, Darmstadt, Juli 1992. (Source: capdef.tex).

[5] Ludwig Hechler et al. MOPS. Das Micro Operating System. Technisches Handbuch. Accelerator Controls Documentation B-MOPS-01 1. Auflage, Gesellschaft für Schwerionenforschung, Darmstadt, August 1990. (Source: mops.tex).

[6] Ludwig Hechler et al. ECMS. The Equipment Control Micro System. Equipment Model of GuP. Accelerator Controls Documentation B-MOPS-03, Gesellschaft für Schwerionenforschung, Darmstadt, Februar 1991. (Source: ecms$docu.com).

[7] Ludwig Hechler et al. The USR Support Library. Accelerator Controls Documentation B-MOPS-04, Gesellschaft für Schwerionenforschung, Darmstadt, Mai 1992. (Source: usr$support$docu.com).

[8] Peter Kainberger et al. EC. The Equipment Control Monitor. Equipment Model of EC. Accelerator Controls Documentation B-ECM-01, Gesellschaft für Schwerionenforschung, Darmstadt, Mai 1991. (Source: ec$docu.com).

[9] Dipl. Phys. Martin Kämmerer. MPVME1001 CPU Benutzerhandbuch. Revision 1.0, Systemforschung, Bonn, Sep 1987.

[10] SCG. pSOS - 68K Real-Time, Multi-processing Operating System Kernel User's Manual. Version 4.1, Software Components Group, Inc., Santa Clara, California, May 1987.

[11] SCG. pREPC+/68K User's Manual. Version 1.2, Software Components Group, Inc., San Jose, California, Oct 1991.

[12] SCG. pROBE+/68K System Debugger User's Manual. Version 1.2, Software Components Group, Inc., San Jose, California, Aug 1991.

[13] SCG. pSOS+/68K User's Manual. Version 1.2, Software Components Group, Inc., San Jose, California, Aug 1991.

[14] N. Wirth. *Algorithmen und Datenstrukturen*. Teubner, Stuttgart, 1979.

Index