Eine PEARL-Umgebung unter einem "Echtzeit"-UNIX

Krupp Atlas Elektronik GmbH
Hans-Dieter Worm
Sebaldsbrücker Heerstr. 235
2800 Bremen 44
☎ 0421/457-3153

1. Übersicht

Es werden Hintergrund, Konzept und Zukunftsperspektive einer Rechnerentwicklung und die Verbindung zweier Betriebssysteme beschrieben. Die Ausgangslage ist eine 16bit Prozeßrechnerfamilie EPR/MPR1300 mit dem Realzeitbetriebssystem MOS1300 und für Anwendungen die Sprache PEARL. Es wird eine 32bit Rechnerfamilie entwickelt, die die Produktpalette nach oben hin erweitern soll. Dabei stützt sich die Entwicklung auf marktübliche Standards, ohne die in unserem Haus übliche Aufwärtskompatibilität, bzw. Portierbarkeit der Software aus dem Auge zu verlieren. Erstmalig laufen die Rechner der neuen Familie mit dem Betriebssystem UNIX System V Release 3.1.

Vor diesem Hintergrund wird ein Konzept entwickelt das es gestattet, beide Betriebssysteme gleichzeitig auf einem Rechner zu betreiben. Der Vortrag beschreibt die Implementation der beiden Systeme unter Ausnutzung der Mehrprozessorarchitektur und die Kommunikationsmechanismen untereinander. Anhand eines Beispiels wird die Kommunikation einer PEARL-*task* unter dem MOS mit einem C-Programm unter UNIX gezeigt.

Als Zukunftsperspektiven werden weitere Entwicklungen oder Planungen geschildert, wie die Verwendung von UNIX als *file server* für das MOS, oder die Einbindung eines MOS-Filesystems in ein UNIX-Filesystem und die Anwendung des Konzepts für die Verbindung von MPR1300 MOS zu MPR2300 UNIX.

2. Ausgangssituation und Hintergrund

Die Ausgangsbasis besteht aus einer Rechnerfamilie EPR/MPR1300, deren Rechner auf einem *Bitslice*-Prozessor basieren. Die Firmware dieses *Bitslice*-Prozessors gestattete es uns, eine sehr effiziente Implementierung unserer Sprache META zu entwickeln. Die Firmware wurde in einer META-ähnlichen Sprache entwickelt, um auch auf dieser Basis portierbar auf weitere Rechner des Hauses zu sein. Auf Basis von META wurde unser Realzeitbetriebssystem MOS1300 entwickelt, das somit ebenfalls portierbar ist. Durch die gleiche Firmware war es möglich, ein ebenso effizientes PEARL zu implementieren (siehe [1]). Unter Verwendung von PEARL wurden in unserem Hause viele Softwareprodukte entwickelt. Dabei nimmt die Entwicklung des Atlas Warten Leitsystems einen wesentlichen Anteil ein. Dieses AWL wurde mehrfach in großen Leitsystemen der Ver- und Entsorgung installiert. Unter anderem aus diesem Einsatzgebiet kam dann die Anforderung nach einer Erweiterung der bestehenden Rechnerlinie. Dabei waren folgende zusätzlichen Forderungen gegenüber der bisherigen Rechnerlinie zu berücksichtigen:

- 32 bit Technologie,
  Einsatz eines Standard-Prozessors,
- Einsatz eines Standard-Bus-Systems
- zusätzlicher Einsatz eines Standard-Betriebssystems

Nach eingehenden Untersuchungen wurde die Entscheidung getroffen, einen Mehrprozessorrechner MPR2300 auf Basis des Motorola 68020/30 Prozessors zu entwickeln. Als Bussystem entschied man sich für den VMEbus. Das Betriebssystem wurde vom MPR1300 übertragen und angepaßt bzw. erweitert. Das MOS2300 läuft aber nicht wie beim MPR1300 auf der Firmware-Schale, sondern auf einem Nukleus, dem MOS-Kern. Dieser Kern wurde in der Programmiersprache META-K erstellt, die sich wesentlich an den Strukturen unser Standardsprache META orientiert, aber optimal auf Programmierung von Prozessoren der Motorola 68000 Serie abgestimmt ist. Das Konzept ähnelt dem Verfahren, das bereits für die Erstellung der Firmware des MPR1300 verwendet wurde. Die Basis-Programme des MOS1300, wie Editoren etc., wurden ebenfalls portiert. Dazu gehören natürlich auch die Compiler, die Testmonitoren und sonstige Hilfsprogramme für die Programmiersprachen META und PEARL. Zu diesen Portierungen möchte ich noch erwähnen, daß wir keine schlichte Portierung unter Mitnahme der 16-Bit-Einschränkungen wollten, sondern zusätzlich zu der reinen Übertragungsarbeit, (natürlich) eine Überarbeitung der Programmsysteme erfolgte, um an den sinnvollen Stellen auf 32 Bit umzustellen. Als letzte Forderung des Vertriebes blieb noch die Implementation eines Standard-Betriebssystems. Wir implementierten also eine Portierung des original AT&T

UNIX System V Release 3.1. Damit hatten wir einen Mehrprozessorrechner, den wir sozusagen "pur" mit einem Mehrprozessor MOS, oder alternativ als Einprozessorrechner, mit UNIX betreiben konnten. Mit Erreichen dieses Status kamen nun sofort die Forderungen, auf beiden Betriebssystemen jeweils die *tools* des anderen Betriebssystems zu implementieren. Diese Forderung hätte jedoch eine quasi Verdopplung der Entwicklungsaufwendungen zur Folge gehabt, ganz zu schweigen von den unterschiedlichen Philosophien, die die Systeme MOS und UNIX verfolgen. Hier war also eine sinnvolle Verbindung der Systeme gefordert.

3. Der UNIX/MOS-Übergang

3.1 Das Konzept

Wie sieht in diesem Umfeld eine sinnvolle Verbindung der Systeme aus? Der erste Gedanke einer Rechnerkopplung der beiden Systeme ist sicher eine der Lösungen, die man nicht außer acht lassen kann. Im Projektgeschäft kann das aber eine zu teure Lösung sein, man benötigt immerhin einen kompletten zweiten Rechner. Also war eine Verbindung in einem Rechner gefordert. Die Idee, UNIX als eine *task* unter MOS laufen zu lassen, scheitert an zu vielen Änderungen im UNIX-Kern. Damit entfernt man sich zu weit vom Standard und hat Probleme mit der Überführung auf neue Releases von AT&T. Außerdem würde die Fehlersuche erheblich erschwert werden. Also entwickelten wir ein Konzept, daß die Systeme unter Ausnutzung der Mehrprozessorarchitektur auf einem gemeinsamen VMEbus arbeiten läßt. Jedes System sollte im ersten Ansatz seine eigene Peripherie erhalten, so daß die weitestmögliche Entkopplung der Systeme erreicht wird. Nur so kann später die Forderung nach Nutzung derselben Verbindung zwischen zwei Rechnersystemen mit MOS und UNIX und eine Verbindung mit MPR1300-Rechnern realisiert werden. Diese einfache Verbindung gestattet es später auch, I/O-Servicefunktionen des einen Systems für das andere nachzurüsten und somit einen Teil der doppelten Peripherie wieder einzusparen. Das Konzept wurde in mehreren Teilschritten realisiert. Im ersten Schritt wurden die beiden Systeme auf einen Rechner gebracht und die gesamte *Boot*- und Startphase entwickelt. Dies war der komplizierteste Teil der Entwicklung. Der zweite Schritt bestand in der Realisierung einer schnellen Kommunikationsverbindung zwischen den Systemen, wobei wir zunächst mit einer einfachen Speicherkopplung begannen und zur Zeit auf ein *buffered pipe protocol* übergehen. Im dritten Schritt wurden dann drei *tools* entwickelt, die den Datentransport zwischen den Systemen unterstützen und das Arbeiten auf dem anderen Betriebssystem ermöglichen. Diese *tools* bieten sowohl eine Bediener-, als auch eine Softwareschnittstelle.

3.2 *Boot* und Start der Systeme

Nach dem *reset* des Rechner-Systems lädt eine *boot*-Routine, entweder automatisch oder dialoggesteuert, die beiden Systeme und startet sie auf den jeweils angegebenen Prozessoren. Im automatischen *boot* wird ein UNIX-Filesystem vorrausgesetzt, in dem ein Textfile genau die Angaben enthält, die auch per Dialog eingegeben werden können. Im Dialog ist anzugeben, welches System zu laden, und auf welchem Prozessor es zu starten ist. Dabei ist die Angabe des Systems ein Filename im UNIX-Filesystem, z.B. /usr/src/uts/unix oder /kae/mos. Ebenso kann das MOS auch über ein *autoload device* (*softdisk* oder *pseudotape*) geladen werden. Wenn z.B. ein MPR2300 mit den Prozessoren 0 bis 4 ausgerüstet ist, so wird über

auto: 0-3>/kae/mos
auto: 4>/unix
auto: "*return*"

den Prozessoren 0 bis 3 das MOS und dem Prozessor 4 das UNIX zugeordnet. Dann werden alle 5 Prozessoren gestartet. Die Systeme haben generierbare disjunkte Speicherbereiche, die sie jeweils überprüfen um ihren Speicherplatz zu ermitteln. Die Systeme starten nahezu unabhängig voneinander. Lediglich die Verteilung der *clock*-Impulse wurde zunächst dem MOS zugeordnet, so daß das UNIX-System am Ende der Startphase auf den ersten *clock interrupt* wartet. Im MOS wird daher während der Initialisierung dem MOS-Kern per Schalter gesteuert mitgeteilt, daß der *clock interrupt* an den generierten UNIX-Prozessor weiterzugeben ist. Auf alle weiteren *shared resources* wurde in den ersten Implementationen bewußt verzichtet. Somit haben z.B. beide System getrennte *system operator terminals*, die später wieder auf einem Gerät abgebildet werden können. Die Peripherie des Rechners ist nun durch entsprechende Vereinbarungen und Systemgenerierungen jeweils genau einem Betriebssystem zugeordnet. Benötigen die Geräte *interrupts*, so werden sie so konfiguriert, daß die *interrupt request line* des entsprechenden Systems benutzt wird, z.B. MOS *line* 2, UNIX *line* 3.

3.2 Das Transportsystem

Das Transportsystem stellt in der Terminologie der Rechnerkopplungen lediglich die unterste Ebene dar, die physikalische Transportebene. Ein wesentlicher Punkt für die Flexibilität der Lösung ist ihre Einfachheit. Nur dadurch, daß das Transportsystem quasi ein Draht zwischen den Systemen ist, kann diese Schicht auch durch eine komfortable Rechnerkopplung zwischen getrennten Rechnern ersetzt werden, z.B. Ethernet und TCP/IP. Das einfachste Transportsystem ist eine Speicherkopplung über *shared memory*.

Beiden Systemen wird ein gemeinsamer statischer Speicherbereich durch die Systemgenerierung bekannt gemacht, in dem zwei *mailboxes* realisiert sind. Jede *box* enthält eine Kontroll-Struktur in der über *flags* oder Variable der jeweilige Status und z.B. die Länge der enthaltenen *message* angezeigt werden. Für jedes System ist eine *box* für das Senden von Nachrichten zum Partner vorgesehen, die das System aktiv zu füllen hat. Das jeweils andere System entnimmt die Nachricht aus dieser *box*, nachdem es per *interrupt* über das Vorliegen einer Meldung informiert wurde.

3.3 Realisierung

Sowohl auf der MOS, wie auch auf der UNIX Seite sind die Transportsysteme durch Treiber realisiert, d.h. unter UNIX als einfacher *character*-orientierter Treiber. Auf der MOS-Seite gehört zu dem Treiber immer ein Software-Prozessor dazu, sozusagen der asynchrone Teil des Treibers. Für die Entwicklung war es wichtig, keinerlei Eingriffe in die UNIX-Kern-*Sources* zu machen, also nur dort zu ändern, wo die Systeme es erlauben. So kann garantiert werden, daß der UNIX/MOS-Übergang auch auf weitere UNIX-Releases übertragen werden kann. Die Speicherkopplung wird von den Treibern so genutzt, daß Nachrichten von den jeweiligen Treibern in diese Speicher kopiert, bzw. aus diesen Speichern heraus kopiert werden. Die lesenden Teile der Treiber enthalten *buffer*-Mechanismen, auf der MOS-Seite sind das *queues* (siehe [2]), auf der UNIX-Seite wurden zunächst eine Reihe statischer *buffer* dafür zur Verfügung gestellt. Der jeweils aktive oder schreibende Teil bei dieser Kommunikation prüft, ob seine entsprechende *box* zu Verfügung steht, füllt sie mit den entsprechenden Daten, setzt die Elemente der Kontrollstruktur und informiert seinen Kommunikationspartner durch einen Interprozessor-*interrupt*. Dieser *interrupt* wird durch die Ansprache des *location monitors* des entsprechenden MOS-Prozessors vom UNIX-Treiber erzeugt, bzw. umgekehrt, wenn der MOS-Treiber seine *box* gefüllt hat. Der so geweckte *interrupt*-Teil des Treibers entnimmt nun die Nachricht und kopiert sie in seinen *buffer* bzw. seine *queue*. Die weitere Bearbeitung erfolgt nach den in dem jeweiligen Betriebssystem üblichen Regeln.

3.4 Verbindungsaufbau

Verbindungen zwischen den beiden Systemen bestehen zwischen einer *queue* und einem Prozeß, vertreten durch seine *process identification* (*pid*). Vom MOS her werden die Kommunikationspartner der UNIX-Seite als *remote queues* betrachtet. Es gibt im MOS eine lokale *queue*, in die alle *remote*-Nachrichten geschickt werden. Die Nachrichten enthalten als Adresse eine *remote queue* und werden von einem Kopplungstreiber weiter

verteilt, d.h. in diesem Fall an UNIX gesendet. Der Verbindungsaufbau zwischen den Partnern wird durch einen MOS Treiber unterstützt. Es gibt ein *info device*, bei dem Anwendungen angemeldet werden, oder das auf Anfrage die bekannten Anwendungen mitteilt. Dieses *info device* ordnet Anmeldungen nach Themenkreisen. Zur Zeit sind definiert: Terminals, *file transport* und *file system switch*. Diese Themenkreise existieren, um im System implizite Dienstleistungen bei entsprechenden Anmeldungen zu starten. Es können jederzeit beliebige andere Themenkreise eingerichtet werden, die dann keine spezielle Behandlung erfahren. Themenkreise sind durch Nummern gekennzeichnet. Eine Anwendung, die eine Dienstleistung offeriert, meldet sich beim *info device* an, indem sie als Parameter ihre *queue* angibt, auf der sie Aufträge empfängt. Weitere Parameter sind der Themenkreis und zur Unterscheidung in einem Themenkreis ein *identifier*, bestehend aus 4 Zeichen. Eine weitere Anwendung, die einen Service in Anspruch nehmen möchte, kann sich beim *info device* erkundigen, welche Anwendungen angemeldet sind. Dazu wird ein *info* Auftrag in die *queue* des *info device* geschrieben. Als Antwort erhält der Absender eine Liste der Anmeldungen bestehend aus Themenkreis, *queue* Nummer und *identifier*. Eine MOS-*task*, die als *server* arbeitet, wird typischerweise eine *queue* eröffnen, sich mit der *queue* Nummer, dem Themenkreis und einem eigenen *identifier* beim MOS *info device* anmelden und anschließend auf Aufträge in dieser *queue* warten. Aufträge werden beantwortet, indem der *server* eine Nachricht in die *queue* schreibt, die bei Auftragserteilung als Absender angegeben wurde. Antwort*queues* werden also nicht beim *info device* angemeldet. Ein UNIX-Prozeß, der einen MOS-Service in Anspruch nehmen möchte, wird typischerweise ein *open* auf das Kommunikations*device* ausführen und einen *info* Auftrag als *write* auf die *queue* des *info devices* absetzen. Die Antwort gelangt über eine *remote queue* in den UNIX-Kern (*queue* Nummer = *pid* des UNIX-Prozesses), von wo sie über einen *read*-Auftrag zum UNIX-*user* gelangt. Aus den erhaltenen Informationen, kann das UNIX-Programm sich dann den MOS-*server* aussuchen und die Aufträge in dessen *queue* schreiben. Die Antworten erhält es durch den *server*, der Absender und Adresse in *remote header* vertauscht und sein Telegramm an die lokale Vermittlungs*queue* sendet.

3.5 Anwendungen

Basierend auf diesen Mechanismen wurden Anwendungsprogramme entwickelt, die zum Betreiben des UNIX/MOS-Übergangs benötigt werden: ein *file transfer* und ein transparentes Terminal. Ausgangspunkt dieser *tools* ist, daß UNIX seine Stärken als Entwicklungsumgebung hat, während das MOS als Ablaufumgebung dominiert. Daher wurde unter

dem MOS das *info device* realisiert, so daß Programme unter UNIX, also von der Entwicklungsumgebung aus, auf Dienstleistungen des MOS zugreifen können. Unter UNIX wurde ein Programm (fts = *file transport service*) entwickelt, das *files* zwischen den Systemen kopiert. Im MOS wird während der Initialisierung eine *task* gestartet, die sich beim *info device* als *server* für den Themenkreis *file transport* anmeldet. Das *fts* erfragt nun über das MOS *info device*, welche *queue* den *file transport* unterstützt und bearbeitet mit seinem MOS-Partner die Kopieraufträge zwischen den Systemen. Dabei sind Datenkonvertierungen zwischen den Systemen notwendig. MOS *files* enthalten typischerweise eine *record*-Struktur, während UNIX *files* lediglich als *byte stream* realisiert sind. Die beteiligten Transportprogramme konvertieren Textfiles zwischen *records* und entsprechend eingestreuten *new lines.* Bei den binären Files ist per Option wählbar, ob die *record*-Struktur des MOS-Files mit zu übertragen ist oder nicht. Soll die Struktur bei späteren Transporten wieder reproduzierbar sein, werden sämtliche *record*-Informationen in die Daten eingetragen.

Das transparente Terminal wurde in zwei Varianten entwickelt, als manuell zu bedienendes Terminal und als programmgesteuertes Termial. Damit kann ein *user* unter UNIX sein Terminal transparent auf die MOS-Seite durchschalten, das sich dann identisch zu einem MOS-Terminal verhält. Ein spezielles *control character* bringt ihn auf die *user* Ebene zurück. Das programmgesteuerte Terminal ist von einem Anwendungsprogramm aus per *fork/exec* zu starten und dann über *messages* vom Programm aus zu bedienen. Mit Hilfe dieses Programmes ist man in der Lage eine Art von *remote job entry* zu implementieren. Die beiden Terminalprogramme verhalten sich ansonsten gleich. In der Startphase des Terminalprogramms wählt es einen MOS-*server* aus dem Themenkreis Terminal des MOS *info devices*. Durch den Auftrag *request terminal* an das MOS *info device* wird die Terminalverbindung initiiert. Dies ist einer der Fälle, wo das *info device* für den Themenkreis Terminal eine spezielle Systemfunktion auszulösen hat. Es wird bei Anforderung eines Terminals ein entsprechendes Treiberprogramm, der Terminalprozessor, gestartet, der wiederum dafür sorgt, daß sich ein *remote* Terminaltreiber bei dem Programm für das transparente Terminal meldet. Nach dieser Initialisierungsphase gibt der UNIX-Prozeß alle Terminaleingaben an das MOS weiter, wo der Terminalprozessor mit dem Treiber dafür sorgt, daß die Eingaben wie Terminaleingaben aussehen und behandelt werden. Die Antworten werden über *remote queues* an den UNIX-Prozeß zurückgesandt und an das Terminal ausgegeben.

3.6 MOS PEARL Kommunikation zu UNIX C

Im nebenstehenden Programm soll anhand eines einfachen Beispiels der Einsatz des UNIX/MOS-Überganges demonstriert werden. Hintergrund ist hier der Einsatz des MOS als "Realzeit-Verstärker" des UNIX-Systems. Das PEARL-Programm offeriert als Dienstleistung beispielhaft die zyklische Messwerterfassung, deren Daten unter UNIX z.B. archiviert, oder in Form von Langzeitstatistiken einer Datenbank zugeführt werden sollen. Zunächst werden die Strukturen für die *remote queue header* und die Telegramme definiert. Der *remote queue header* kann auch bei lokalen *queues* verwendet werden. Es werden jeweils für Empfänger und Absender (*dest, source*) die *queue*-Nummern und die zugehörigen *link*-Prozessoren (Rechner) definiert. Eine weitere *queue* wird bei *remote* Verbindungen als Vermittlungs*queue* vom MOS eingetragen, um bei Antworten den lokalen Vermittler zu kennzeichnen. Telegramme bestehen aus dieser *header*-Struktur und einem Datenteil. In diesem Beispiel gibt es das Format für die Anmeldung beim *info device*, ein Auftragstelegramm zum Starten der zyklischen Erfassung und ein Datentelegramm, das als Ergebnis der Erfassung an das UNIX-Programm zurückgesendet wird.

Das Dienstleistungsprogramm eröffnet zunächst seine Auftrags*queue* und meldet sich dann beim *info device* an. Dazu wird das entsprechend formatierte Telegramm in die lokale *queue* eingetragen. Das *info device* bestätigt die Anmeldung mit einem Telegramm in der Auftrags*queue*. Anschließend erfolgt in der Schleife ein Lesen der Auftrags*queue*. Zeigt die Option einen Erfassungsauftrag an, so wird die *task* CYCL entsprechend gestartet. Im anderen Fall wird die *queue* geschlossen und das Programm terminiert.

In der *task* CYCL wird nach der Meßwerterfassung und Datenaufbereitung ein Antworttelegramm an die beauftragende UNIX-Software geschickt. Dazu werden die Daten von *destination queue* und Linkprozessor der Auftrags*queue* als Absender in den *header* des Datentelegramms eingetragen, während die Absenderparameter des Auftragstelegramms (*source queue* und Linkprozessor) als Empfänger eingetragen werden. Das Telegramm wird an die lokale Vermittlungs*queue* geschickt.

Das zugehörige C-Programm führt ein *open* auf ein *special file* /dev/muc aus und schreibt über dieses MOS-UNIX-Kopplungs*device* einen Informationsauftrag (OPT = 1) an das *info device* auf der MOS-Seite. Es erhält als Antwort auf einen weiteren *read* die Liste der angemeldeten Dienstleistungen und wählt über den Themenkreis "zyklische Erfassung" (= MESSEN) und den *identifier* "CYCL" seinen Partner, bzw. dessen *queue* aus. In diese *queue* wird nun der Auftrag für eine Meßperiode geschrieben. Als Absender trägt der Treiber die *process id* ein. Die Antworttelegramme werden in einer Schleife durch *read* Aufträge gelesen und verarbeitet.

```
MODULE MESRV; /* MESSWERT ERFASSUNGS SERVER */
PROBLEM;
TYPE     RMQHD    STRUCT  [ DESTQ    FIXED,     /* DESTINATION QUEUE     */
                            DESTL    FIXED,     /* DEST. LINK PROCESSOR */
                            SRCQ     FIXED,     /* SOURCE QUEUE          */
                            SRCL     FIXED,     /* SRC. LINK PROCESSOR   */
                            LOCALQ   FIXED];    /* LOCAL QUEUE           */
DCL      INFO     STRUCT  [ QHD      RMQHD,     /* REMOTE QUEUE HEADER */
                            OPT      FIXED,     /* INFO OPTION           */
                            LNG      FIXED,     /* EXTENDED LENGTH       */
                            STA      FIXED,     /* STATUS WORD           */
                            THEMA    FIXED,     /* THEMENKREIS           */
                            ID       CHAR (4),  /* IDENTIFIER            */
                            QUEUE    FIXED,     /* SERVICE QUEUE NUMBER*/
                            QSTA     FIXED];    /* QUEUE STATUS          */
DCL      JOB      STRUCT  [ QHD      RMQHD,     /* REMOTE QUEUE HEADER */
                            OPT      FIXED,     /* MESS-OPTION           */
                            INTERVALL  DURATION,
                            ENDTIME  CLOCK];
DCL      MEDAT    STRUCT  [ QHD      RMQHD,     /* REMOTE QUEUE HEADER */
                            WERT     FIXED];    /* MESSWERT              */
DCL  STV  STRUCT [ ( STA , LNG , QNR )  FIXED ];
DCL  OPB  STRUCT [ ( TYP , MAXL , MAXN )  FIXED ];
MAIN: TASK;
   OPB.TYP = 0;  OPB.MAXL = 50;  OPB.MAXN = 5;
   OPEN_QUEUE( QNR , STV , OPB );
   INFO.THEMA = MESSEN; INFO.ID = "CYCL"; INFO.QUEUE = QNR; INFO.STA = 0;
   PUT_REQUEST ( IQNR , STV , INFO , INFOLNG ); /* ANMELDUNG INFO DEVICE */
   GET_REQUEST ( QNR , STV , INFO , INFOLNG );  /* ANMELDUNGSQUITTUNG    */
   REPEAT;
      GET_REQUEST ( QNR , STV , JOB , JOBLNG ); /* AUFTRAG LESEN */
      IF ( JOB.OPT  EQ  MESSEN )
         THEN  ALL JOB.INTERVALL  UNTIL  JOB.ENDTIME  ACTIVATE  CYCL;
         ELSE  CLOSE_QUEUE ( QNR , STV );     TERMINATE;
      FIN;
   END;
END;
CYCL:  TASK;
   /* MESSWERTERFASSUNG UND DATENAUFBEREITUNG */
   MEDAT.QHD.DESTQ   = JOB.QHD.SRCQ;      /* SET DESTINATION QUEUE     */
   MEDAT.QHD.DESTL   = JOB.QHD.SRCL;      /* SET DESTINATION LINK-PROC */
   MEDAT.QHD.SRCQ    = JOB.QHD.DESTQ;     /* SET SOURCE QUEUE          */
   MEDAT.QHD.SRCL    = JOB.QHD.DESTL;     /* SET SOURCE LINK PROC      */
   PUT_REQUEST ( MEDAT.QHD.LOCALQ , STV , MEDAT , MEDATLNG );
END;
```

Unter UNIX würde man für solche Aufgaben typischerweise ein Startprogramm für die Auswahl des Servicepartners erstellen. In diesem Programm würde dann die Dialogführung mit dem Bediener und die Übermittlung der Erfassungsaufträge an das MOS durchgeführt. Für die Auswertung würde über *fork/exec* ein eigenes Programm gestartet.

4. Weiterentwicklungen

4.1. *Buffered Pipe Protocol*

Das *buffered pipe protocol* liefert eine Standardmethode zum Austausch von Nachrichten zwischen Prozessoren, die auf einem VMEbus arbeiten und Zugriff auf einen gemeinsamen Speicher oder einen gemeinsamen Adressbereich haben [3]. Schon diese Zieldefinition von Motorola zeigt die Eignung dieses Protokoll für die hier vorliegende Aufgabe. Als weitere Entwicklung wird zusätzlich zur geschilderten Speicherkopplung dieses *buffered pipe protocol* implementiert. Der Hintergrund dabei ist, daß dieses Protokoll für sogenannte *targets* angeboten wird. Das heißt für die Entwicklung von *firmware* oder *controller* Software für Baugruppen mit Motorola Prozessoren kann dieses Protokoll implementiert werden. Wir werden über dieses Protokoll mit dem Konzept des UNIX/MOS -Übergangs solche *targets* an unsere Betriebssysteme koppeln. Sowohl für den UNIX/MOS-Übergang, wie auch für die Entwicklungsumgebung für *target*-Systeme spezifizieren wir ein gemeinsames Testkonzept. Dabei soll ein *low level debugger* auf dem System des zu testenden Objekts laufen und das Objekt kontrollieren. Unter UNIX wird dann der sprachorientierte Teil des *debuggers* laufen und die Eingaben behandeln. Die Kommunikation erfolgt über den UNIX/MOS-Übergang bzw. über die *target* Anbindung. Für den Bediener steht damit immer eine einheitliche Umgebung zur Verfügung. Inwieweit dieses Verfahren auch zum Test von UNIX-*usern* geeignet ist, wird zur Zeit geprüft. Grundlage dieses *debuggers* wird der PEARL-Testmonitor des MPR1300 werden.

4.2 *Streams*

Der oben geschilderte Treiber und auch der Treiber für das *buffered pipe protocol* werden auf *streams* Treiber bzw. Module umgestellt. *Streams* sind ein neuer Mechanismus im AT&T UNIX (ähnlich den *sockets* im Berkley UNIX), der hauptsächlich als Ablaufumgebung und Unterstützung von Protkoll-Software von Kommunikationen entwickelt wurde. Durch die Umstellung auf *streams* können Teile des Verbindungsaufbaus und des *handshakes* zwischen den Systemen aus den Anwendungsprogrammen in *streams modules* übernommen werden. Desweiteren kann dann der "Draht" zwischen den Systemen, die Speicherkopplung (oder das *buffered pipe protocol*), problemlos durch andere Kopplungen und Prozeduren

ersetzt werden. Das setzt natürlich voraus, daß die Kopplungsprozeduren als *streams modules* und Treiber realisiert sind. Mit dieser Umstellung wird es möglich, die *queues* unter UNIX einzuführen. Ein UNIX-Prozeß kann dann mehrere Verbindungen gleichzeitig unterhalten (Entkopplung von der *process id*), da über *streams* ein *multiplex/demultiplex* Modul einfach implementiert werden kann. Zusätzlich wird es dann einfach möglich, ein *info device* unter UNIX zu entwickeln. Damit würde der Übergang zwischen beiden Systemen symmetrisch, was insbesondere bei der Verbindung mehrerer Rechner von Vorteil ist. Jeder Rechner verwaltet dann eine Liste seiner Dienstleistungen, die über *remote* Verbindungen von anderen System abgefragt und genutzt werden können.

4.3 VME-Subsysteme und Rechner-Rechner Übergang

Unter Verwendung dieser Kopplungssoftware und einer VME-VME Kopplungshardware werden wir den UNIX/MOS-Übergang von einem VMEbus-System auf ein System von einem VMEbus mit bis zu 3 VME-Subsystemen übertragen. Diese Konfigurationen wurden bereits beim EPR/MPR1300 durch Grund- und Erweiterungseinschübe realisiert. Die Koppelhardware verbindet zwei VMEbus-Systeme miteinander, wobei ein VMEbus die Funktion eines *master* übernimmt. Zugriffe vom *master* in ein VME-Subsystem sind für die Programme transparent. Vom Subsystem zum *master* ist eine *interrupt* gesteuerte Kommunikation realisiert. Damit können die Systeme noch weiter entkoppelt werden, und die gegenseitigen Einflüße werden weiter reduziert, die Fehlersuche wird erleichtert. Diese Konfiguration wird dann auch auf die Kopplung zweier Rechner mit getrenntem VMEbus erweitert, um größere Systeme zu realisieren. Der UNIX/MOS-Übergang ist nach dem vorgestellten Konzept geeignet, die Systeme MPR1300 MOS mit dem MPR2300 UNIX zu verbinden. Diese Systemverbindung wird dadurch realisiert, daß die Erweiterungen des MOS2300 auf das MOS1300 übertragen werden. Das ist einfach möglich, da beide Betriebssysteme in META programmiert sind.

4.4 *File system switch* und *file server*

Als weitere Entwicklung läuft zur Zeit die Einbindung eines MOS Filesystems in ein UNIX Filesystem. Unter Benutzung des UNIX *file system switch* kann ein MOS Filesystem in den Baum des UNIX Filesystems eingehängt werden (*mount*). Die *files* des MOS sind dann für den UNIX Bediener und die Programme nicht von UNIX *files* zu unterscheiden. Erstmalig mit dem Release 3 des System V hat AT&T diesen *file system switch* im UNIX Kern implementiert (siehe auch [4]). Es handelt sich um eine Schnittstelle im Kern, dessen Syntax und Semantik für die verschiedenartigen Zugriffe auf *files* und Filesysteme

spezifiziert ist. Zur Einbindung eines neuen Filesystems sind ca. 25 neue Kernroutinen zu entwickeln, die für das MOS-Filesystem an den notwendigen Stellen über den UNIX/MOS-Übergang mit einem entsprechenden Partner kommunizieren.
Der umgekehrte Weg, die Implementation eines MOS *file systems* auf einer UNIX-Platte, und die *server* Funktionen, sind bereits spezifiziert. Diese Entwicklung ist bei Kleinsystemen sinnvoll, wenn in der Realzeitumgebung wenig Plattenaktivitäten erforderlich sind. Auf der UNIX-Platte wird ein *minor device* für ein MOS Filesystem reserviert. Das UNIX übernimmt dann eine *server* Funktion auf der Basis von Block-I/O Aufträgen. Unter dem MOS ist dafür ein Pseudotreiber zu entwickeln, der als neuer Plattentreiber oder *harddisk*-Treiber in das MOS eingebunden wird und die Kommunikation mit dem UNIX *server* abwickelt.

5. Perspektiven

5.1 Multiprozessor-UNIX

Zur Zeit wird im Rahmen eines Verbundprojektes für unseren Rechner ein Multiprozessor-UNIX auf Basis des *local shared memory* Konzeptes entwickelt [5]. Nach dem bisherigen Kenntnisstand ist es möglich, den UNIX/MOS-Übergang auch für dieses Multiprozessor-UNIX zu implementieren, ohne dessen *source code* zu benötigen. Wir werden das nach Abschluß der Multiprozessor-UNIX-Entwicklung im Laufe des nächsten Jahres implementieren und damit den Nachweis erbringen, das die Portierung des UNIX/MOS-Übergangs auf ein "anderes UNIX" lediglich eine Portierung von Treibern ist.
Damit bildet die Rechnerfamilie, angefangen vom EPR/MPR1300 mit Ein- und Mehrprozessor-MOS bis zum MPR2300 mit Mehrprozessor-MOS und Mehrprozessor-UNIX zusammen mit dem UNIX-MOS-Übergang, die Basis für die Realisierung komplexer dezentraler Systeme.

5.2 Realzeit-UNIX

Zur Zeit der Drucklegung dieses Papiers gibt es noch keine, bzw. noch wenig Informationen über die sogenannten Realzeit-Erweiterungen des UNIX System V Release 4, der Ende 1989 als ß-Release für den 3B2 angekündigt ist. Diese und ähnliche Entwicklung, z.B. AIX, wird man weiter aufmerksam beobachten müssen. Daraus wird sich evtl. einmal ein standardisiertes Echtzeit-Betriebssystem entwickeln. Ebensolche Aussichten bestehen allerdings für das VMEexec Projekt von Motorola, bzw. die RTEID. Im Augenblick ist jedenfalls kein Anbieter zu erkennen, der einen deutlichen Vorsprung in Form eines entsprechenden Marktanteils hat, so daß man einen de facto Standard feststellen

könnte. Diese Strategie wird unter anderem von X/OPEN angewendet um Standards festzustellen, in den *Portability Guides* festzuschreiben und damit zu fördern. Dort gibt es zur Zeit keine Verlautbarungen über die Standardisierung von Realzeiteigenschaften. Allerdings arbeiten sowohl AT&T, wie auch X/OPEN im IEEE–Komitee P 1003 mit, die im Rahmen der herstellerunabhängigen Schnittstellendefinition von Standard-Betriebssystemen unter P 1003.4 *Realtime facilities* spezifizieren (siehe [6]). Damit ist anzunehmen, daß das neue Release 4 von AT&T diesem Standard entspricht. Es wird aber einige Zeit dauern, bis dieses System verbreitet ist und es bleibt abzuwarten, ob es sich gegen andere Produkte durchsetzen kann. Somit wird uns ein vom Markt anerkanntes standardisiertes Realzeitsystem wohl noch für etliche Zeit vorenthalten werden. Diese Lücke wollen wir bei Krupp Atlas Elektronik für unsere Rechner mit der Kombination unserer bewährten MOS-Realzeitsysteme und dem standardisierten UNIX ausfüllen.

Literatur

[1] Brüning, C., Landsberg, G., Krupp Atlas Elektronik, Bremen:
Implementation von PEARL auf dem Mehrprozessorrechner MPR1300.
Tagungsband PEARL 84, Düsseldorf, Hrsg.: PEARL-Verein e.V.

[2] Bockhoff, W., Krupp Atlas Elektronik, Bremen:
Ein Botschaftssystem mit PEARL-Schnittstelle zur Kommunikation in verteilten Systemen. Tagungsband PEARL 83, Düsseldorf, Hrsg.: PEARL-Verein e.V.

[3] Motorola Inc.:
Common Enviroment User's Manual, Appendix A Buffered Pipe Protocol.
Motorola Microcomputer Division 1986

[4] Meyer, A., Stollmann GmbH, Hamburg:
Ein schnelles Filesystem unter dem UNIX 5.3 Filesystem-Switch
Tagungsunterlagen GUUG-Jahrestagung '88

[5] Klemke, G., Stollmann GmbH, Hamburg:
SUPRENUM - ein Parallel-Superrechner.
Tagungsunterlagen GUUG-Jahrestagung '88

[6] Windauer, H., Werum GmbH, Lüneburg:
UNIX als Entwicklungsumgebung für Realzeit-Anwendungen - Ein Überblick –
Tagungsband PEARL 87, Boppard, Hrsg.: PEARL-Verein e.V.