Das EMS-Client-Programm '*klient*' –Benutzeranleitung–

M. Drochner

13. Juli 1994

Im folgenden wird das Programm 'klient' beschrieben. Dieses Programm stellt eine Implementierung der Client-Seite des EMS-Protokolls dar. Diese Beschreibung wendet sich an den Anwender, sie umfaßt die Bedienung sowie die Befehlssyntax. Um das Programm auch sinnvoll einsetzen zu können, sind Kenntnisse der Architektur von EMS und der Semantik der einzelnen EMS-Requests nötig. Eine Einführung dazu ist in [1] und [2] zu finden.

Inhaltsverzeichnis

1 Benutzerhandbuch

1.1 Allgemeines

Das hier beschriebene Programm 'klient' dient dazu, EMS-Requests an EMS-Server abzusetzen sowie eingehende Confirmations und uncolicited Messages auszuwerten und darzustellen. Dazu benötigt es die Dienste eines (lokalen oder über Netzwerk erreichbaren) EMS-Kommunikationsprozesses ('commu'); es verbindet sich beim Start über einen UNIX- oder IP-Domain-Socket mit diesem. Die Bedieneroberfläche ist textorientiert, Eingaben sind auch aus Script-Files oder (über die Standardeingabeumleitung) anderen UNIX-Prozessen möglich. Ursprünglich war 'klient' nur als etwas komfortableres Interface während der Entwicklung von EMS und für kleinere Testsysteme konzipiert, es erwies sich aber auch für den praktischen Einsatz zum Steuern von Experimenten als brauchbar. Das simple Konzept bedingt allerdings einige Beschränkungen, wie:

- Nur jeweils eine aktive Verbindung zu einem VED.
- Keine Kontrollstrukturen, wie bedingte Verzweigungen, Unterroutinen etc.
- Keine Fehlerbehandlung, Fehlermeldungen werden nur ausgegeben.

In folgenden werden für die Beschreibung von Syntaxelementen einige allgemein übliche Konventionen verwendet. Im einzelnen sind dies:

- [optionale Argumente stehen in eckigen Klammern]
- <*Meta-Ausdrücke sind durch spitze Klammern gekennzeichnet*>
- drei Punkte bezeichnen eine beliebige Wiederholung ...

Der aufmerksame Leser wird auch bemerken, daß `Eingaben des Benutzers` in einer etwas größeren Schrift als `Ausgaben des Programms` gesetzt sind. Auch sind Buchstaben in `Schreibmaschinenschrift` eher wörtlich zu nehmen, wohingegen *kursive Schrift* meist beschreibend oder kommentierend eingesetzt wird.

1.2 Aufruf

Der Start des Programms erfolgt mit:

klient [-s <*socketname*>] [-h <*hostname*>] [-p <*portno*>] [<*script*>]

Dabei sind:

socketname: Name des von 'commu' bereitgestellten UNIX-Domain-Sockets (bei lokaler Verbindung), default: /var/tmp/emscomm

hostname: Name des Rechners, auf dem 'commu' läuft (bei Netzverbindung)

portno: Portnummer des von 'commu' bereitgestellten Internet(TCP)-Sockets (bei Netzverbindung), default: 4096

script: Name eines Textfiles mit Kommandos, dieses wird ausgeführt und das Client-Programm beendet

Statt Kommandozeilenoptionen können für die Verbindungsparameter auch Umgebungsvariablen verwendet werden:

- `EMSSOCKET`: für *socketname*
- `EMSHOST`: für *hostname*
- `EMSPORT`: für *portno*

Falls sowohl Umgebungsvariablen als auch Kommandozeilenoptionen angegeben sind, siegen die letzteren.

Soweit vom UNIX-System unterstützt, ist es auch möglich, 'ausführbare Scripts' zu verwenden. Dies ist eine Erweiterung des gewöhnlichen Begriffs von ausführbaren Programmen auf Kommandofiles, die von einem Interpreter verarbeitet werden. Ein solches pseudo-Programm muß mit einer Kennung ('`#!`', auch 'magic number' genammt) beginnen, danach folgt der Name des Interpreters. Beim Versuch, ein solches File durch Aufruf des Scriptnamens auszuführen, startet das System den Interpreter und übergibt ihm den Namen des ursprünglichen Files als Argument. Die erste Zeile des 'klient'-Scripts müßte also:

`#!`<*Pfad zu klient*>

lauten und das File Execution-Permission besitzen. Ein Aufruf des Scripts wird dann vom System in:

<*Pfad zu klient*> <*Filename des Scripts*>

umgesetzt und ausgeführt. 'klient' wird dann mit dem Scriptnamen auf der Kommandozeile aufgerufen, führt die Befehle im Script aus und beendet sich. Die spezielle erste Zeile wird dabei, wie alle mit '`#`' beginnenden Zeilen, als Kommentar angesehen und ignoriert.

Der <*Pfad zu klient*> darf im allgemeinen nur 14 Zeichen lang sein (Workaround: Verwendung der GNU-bash). Shell-Mechanismen wie Suchpfad oder Aliases wirken hier nicht.

Falls beim Aufruf der *script*-Parameter nicht angegeben wurde, akzeptiert das Programm Kommandoeingaben vom Standardeingabepfad. Wenn dies (wie meist der Fall sein sollte) ein Terminal ist, wird der interaktive Modus verwendet, d.h., es wird ein Prompt ausgegeben.

1.3 Startup-File

Falls im aktuellen Arbeitsverzeichnis ein Textfile mit dem Namen '.klientrc' existiert, wird dieses beim Start als Script eingelesen und ausgeführt. In diesem File könnten z.B. Anweisungen zum Setzen lokaler Variablen ('`set`') oder andere Initialisierungsbefehle stehen. Falls dieses File nicht existiert, wird ein kurzer Hinweis („kein .klientrc“) ausgegeben.

1.4 Eingabe von Kommandos

1.4.1 Allgemeines

Im interaktiven Modus fordert das Programm mit einem '>' zur Eingabe von Kommandos auf. Wenn eine Verbindung zu einem VED geöffnet wird, ändert sich der Prompt in

'<*VED-Name*>:<*Instrumentierungssystem-Nummer*>>'.

Beispiel:

```
>open testved
testved:0>is 3
testved:3>
```

Ein Kommando besteht im allgemeinen aus mehreren durch Leerzeichen oder Tabulatoren getrennten Worten. Es wird durch ein Zeilenende abgeschlossen. Eine Zeile darf maximal 1024 Zeichen lang sein. (Diese Beschränkung gilt auch für Scripts.)

Dabei ist das erste Wort das eigentliche Befehlswort, ihm folgen Argumente, deren Interpretation vom Befehlswort abhängig ist. Eine Liste aller gültigen Befehlsworte und deren Argumente steht im Kommandoreferenzteil. Wenn ein Befehlswort nicht erkannt wird oder versucht wird, ein nichtlokales Kommando auszuführen, obwohl keine Verbindung zu einem VED besteht, erfolgt eine Fehlermeldung.

Prozedurlisten als Argumente können sich auch über mehrere Zeilen erstrecken; wenn die Anzahl der geöffneten und schließenden runden oder geschweiften Klammern nicht übereinstimmt, werden weiter Eingabezeilen eingelesen, bis alle Klammern wieder geschlossen sind. Im interaktiven Modus wird dies durch ein '+' als Prompt angezeigt. Außerdem kann durch Anführungszeichen ('"') erzwungen werden, daß mehrere Worte oder Zeilen der Eingabe als ein zusammenhängendes Wort betrachtet werden. Auch hier werden weitere Zeilen eingelesen, bis das Wort durch das (identisch aussehende) Ausführungszeichen beendet wird. Das führende und das schließende Anführungszeichen werden vor der Verarbeitung des Kommandos entfernt, so daß sie nicht mehr Bestandteil des erzeugten „großen" Argumentwortes sind. Falls das Programm Folgezeilen anfordert, und dem verzweifelten Benutzer die Übersicht über die geöffneten Klammern verlorengegangen ist, kann man sich durch Eingabe eines EOF-Zeichens (^D) jeweils um eine Klammerungsebene nach außen bewegen, bis das Erscheinen des normalen Promptes die Rettung verkündet.

Wenn kein Befehl eingegeben wird, werden Confirmations im Augenblick ihres Eintreffens bearbeitet (d.h., angezeigt oder einem anderen Callback übergeben, siehe 1.7). Mit dem Beginn der Eingabe einer Kommandozeile werden die Confirmations bis nach den Abschluß des Kommandos zurückgestellt, so daß die Zeile ungestört editiert werden kann.

Die Eingabe des EOF-Zeichens (^D) am Zeilenanfang beendet das 'klient'-Programm. Es ist auch möglich, Befehle aus einem File oder einem anderen Prozeß über den Standardeingabepfad dem Client-Programm zuzuführen; in diesem Fall hat das Schließen des Pfades durch die Quelle die gleiche Wirkung.

1.4.2 Zeileneditor

Zur interaktiven Eingabe von Kommandos wird die GNU-readline-Bibliothek verwendet. Dadurch stehen Emacs-ähnliche Editiermöglichkeiten sowie eine History-Funktion zur Verfügung. Die wichtigsten Funktionstasten sind:

- <*Pfeil links*> oder ^B: ein Zeichen nach links
- <*Pfeil rechts*> oder ^F: ein Zeichen nach rechts
- <*Pfeil nach oben*> oder ^P: einen History-Eintrag zurück
- <*Pfeil nach unten*> oder ^N: einen History-Eintrag vorwärts
- ^U: ganze Zeile löschen
- ^K: von der Cursorposition bis zum Zeilenende löschen
- ^A: springe an Zeilenanfang
- ^E: springe an Zeilenende
- <*Tab*>: vervollständige das Kommando

Das Verhalten der Kommandovervollständigungsfunktion ist davon abhängig, ob eine Verbindung zu einem VED aufgebaut ist; es werden nur die jeweils sinnvollen Kommandos expandiert. Beim ersten Drücken von <*Tab*> wird das begonnene Kommando soweit wie möglich komplettiert. Wenn das nicht eindeutig ist, wird beim zweiten Mal eine Liste der noch möglichen Vervollständigungen ausgegeben. Wenn am Zeilenanfang also <*Tab*> doppelt gedrückt wird, entsteht eine Übersicht über alle derzeit anwendbaren Befehle.

Die Möglichkeiten von 'readline' gehen weit über das Beschriebene hinaus, hier wird auf das Handbuch [3] verwiesen.

1.5 Variablen

Im 'klient' können lokale Variablen definiert werden, um Scripts übersichtlicher zu machen oder Zwischenergebnisse zu speichern. Außerdem ist es möglich, mit Hilfe von Variablen simple arithmetische und logische Operationen (z.B. für Adreßberechnungen) durchzuführen. Eine Variable kann eine beliebige Anzahl von Worten enthalten. Sie wird mit

`set` <*Variablenname*> <*Wort*> ...

gesetzt und mit

`set` <*Variablenname*>

gelöscht. (Es gibt also keine Variablen mit der Länge '0'.) Definierte Variablen können dann in Kommandos (auch innerhalb von Prozedurlisten) verwendet werden, indem dem Namen ein '$' (ähnlich der Syntax der UNIX-Shells) vorangestellt wird. Von aus mehreren Worten bestehenden Variablen können auch Teilbereiche gewonnen werden. Die Notation dafür lautet

<Variablenname>[*<Position>*]

<Variablenname>[*<Anfang>*:*<Länge>*]

Die erste Form liefert ein Wort aus der gegebenen Variablen, die zweite einen Bereich. Die Indices müssen (konstante) Zahlen sein.

Position bzw. *Anfang* beginnen ihre Zählung mit '0', d.h., `$x[0]` ist das erste Wort der Variablen x. *Länge*=0 bedeutet soviel wie 'Rest der Variablen'.

`$x[0:0]` ist also die gesamte Variable x.

Eine Liste der definierten Variablen und ihrer Längen (in Worten) erhält man mit

```
vars
```

Wenn die referenzierte Variable nicht existiert, wird kein Fehler gemeldet, sondern die Vaiablenreferenz wörtlich interpretiert.

Beispiel:

```
>set a 3 4 5 6
>echo $a
3 4 5 6
>echo $a[1:2]
4 5
>echo $b
$b
```

Die arithmetischen und logischen Operationen haben als Ziel jeweils eine Variable. Als Operanden stehen

- plus (Addition)
- minus (Subtraktion)
- and (logisches UND)
- or (logisches ODER)
- not (logische Negation)

zur Verfügung. Die Syntax lautet:

<Operator> *<Zielvariable>* *<Operand 1>* [*<Operand 2>*]

Beispiel:

```
>set a 7
>minus b $a 3
>echo $b
4
```

1.6 Eingabeumleitung

Zum Speichern von Parametern, Resultaten etc. über etwas längere Zeit können diese statt in Variablen auch in Files ablelegt und aus diesen wieder eingelesen werden.

Wenn in einem Kommando (oder einer Prozedurliste) ein Argumentwort die Form

<<Filename>

oder

<$<Variable> (Variable enthält den Filenamen)

hat, bewirkt dies das Einlesen und Interpretieren des Files namens *Filename* bzw. des durch den Variableninhalt bezeichneten Files. Das File wird als Textfile, wie gewöhnliche Eingabe, interpretiert, mit der Ausnahme, daß Zeilenenden wie normale Leerzeichen behandelt werden. Das ganze File erzeugt also nur Argumente innerhalb einer Eingabezeile.

Solche Files können mit einem gewöhnlichen Texteditor erstellt werden; auch durch Ausgabeumleitung (siehe 1.8) erzeugte Files können wieder eingelesen werden.

Beispiel:

```
>echo <datenfile
0 8 15 (Das sei der Inhalt von 'datenfile'.)
>set name datenfile
>echo $name
datenfile
>echo <$name
0 8 15
```

Man beachte, daß zwischen dem Zeichen '<' und dem Filenamen, im Gegensatz zu den analogen Konstrukten der UNIX-Shells, kein Leerzeichen stehen darf.

1.7 Die Callback-Queue

Die Kommandos im Client-Programm lassen sich in zwei prinzipiell verschiedene Gruppen einteilen. Die eine enthält die lokal abzuarbeitenden Befehle. Diese betreffen die Organisation von Verbindungen zu VEDs und der Verwaltung von lokalen Variablen. Die Kommandos der anderen Gruppe bewirken das Senden von EMS-Requests zu einem EMS-Server und setzen daher die Existenz einer EMS-Verbindung voraus. Der Server beantwortet jeden Request mit einer Confirmation, die über Erfolg oder Mißerfolg der Aktion informiert und eventuelle Rückgabewerte enthält. Das Client-Programm arbeitet asynchron, d.h., nach dem Absetzen eines Requests wird nicht auf die Confirmation gewartet, sondern mit der Bearbeitung von Kommandos fortgefahren. Confirmations werden

dann bei ihrem Eintreffen (das nach vielen weiteren Requests erfolgen kann) interpretiert. Manche Confirmations können nur mit Hilfe von Informationen aus dem ursprünglichen Befehl verarbeitet werden. Wenn z.B. der Inhalt einer Confirmation in ein File ausgegeben werden soll, müssen diese Tatsache und der gewünschte Filename zwischengespeichert werden. Diesem Zweck dient die Callbackqueue.

Wenn ein Request abgeschickt wird, werden Informationen über die Behandlung der Antwort zusammen mit einem Identifikator (Transaction-ID) in einer Liste abgespeichert. Wenn eine Confirmation eintrifft, werden die zugehörigen Daten gelesen und wieder aus der Liste entfernt. Fehlermeldungen von mißglückten Requests werden auf das Terminal ausgegeben, andernfalls wird die im Callback spezifizierte Aktion ausgeführt. Sofern in der Beschreibung des Befehls nichts anderes erwähnt ist und keine Ausgabeumleitung stattfindet, werden eventuelle Daten in der Confirmation durch den Default-Callback als Hexadezimalzahlen auf das Terminal ausgegeben. Einige Kommandos, deren Resultate in dieser Form ziemlich schwer zu verstehen wären (z.B., wenn sie Strings enthalten), installieren eigene Ausgabeprozeduren als Callbacks. In den anderen Fällen werden spezielle Routinen dann ausgeführt, wenn der Benutzer um Ausgabeumleitung (siehe nächstes Kapitel) nachgesucht hat.

Auf das Eintreffen aller noch ausstehenden Confirmations kann mit dem Kommando '`flush`' gewartet werden. So kann z.B. garantiert werden, daß Operationen auf mehreren VEDs auch in der beabsichtigten Reihenfolge ablaufen.

1.8 Ausgabeumleitung

Resultate der EMS-Requests (sofern in deren Beschreibung nicht anders vermerkt) und des internen Kommandos '`echo`' können, statt auf das Terminal auggegeben zu werden, auch in Files abgelegt, Variablen zugewiesen oder als Eingabe für externe (UNIX-)Prozesse verwendet werden. Die jeweilige Umleitung wird dem Client-Programm durch das letzte Argumentwort des Befehls mitgeteilt. Folgende Möglichkeiten existieren:

<*Kommando*> [<*Argumente*>] ><*Filename*>: Das Resultat von *Kommando* wird in ein File namens *Filename* geschrieben. Ein eventuell existierendes File wird überschrieben. Falls in der Befehsbeschreibung nicht anders vermerkt, wird die Confirmation eines EMS-Kommandos dazu in ASCII-Hexzahlen formatiert. Die Argumente von '`echo`' werden unverändert kopiert. Zwischen die ins File geschriebenen Worte werden Trennzeichen eingefügt. Ein so geschriebenes File kann wieder mittels '`<`' als Eingabe für folgende Befehle verwendet werden.

<*Kommando*> [<*Argumente*>] >$<*Variablenname*>: Wie oben, nur wird der Filename der angegebenen Variablen entnommen.

<*Kommando*> [<*Argumente*>] :<*Variablenname*>: Das Resultat von *Kommando* wird in einer lokalen Variablen namens *Variablenname* abgelegt. Falls in der Befehsbeschreibung nicht anders vermerkt, wird die

Confirmation eines EMS-Kommandos dazu in ASCII-Hexzahlen formatiert. Ein eventueller vorheriger Inhalt wird überschrieben. Die Variable hat also nachher eine Länge, die der Anzahl der Datenworte in der Confirmation entspricht. Die Argumente von '`echo`' werden unverändert übernommen, die Variablenlänge entspricht nachher deren Anzahl.

<*Kommando*> [<*Argumente*>] |<*Programmname*>: Beim Eintreffen der Confirmation (bzw. sofort bei '`echo`', da dies lokal abgearbeitet wird) wird das Programm *Programmname* ausgeführt. Das Resultat des Kommandos (also der Inhalt der Confirmation bzw. die Argumentliste von '`echo`') wird dem gestarteten Prozeß als Standardeingabe zugeführt. Die Confirmationsdaten werden dazu nicht formatiert, d.h. sie werden binär übertragen. Die Argumente von '`echo`' werden dagegen unverändert, d.h. als Text, gefolgt von einem Zeilenendezeichen, übergeben.

Beispiele:

```
>echo Hallo >file1
>echo Leute :var1
>echo <file1 $var1
Hallo Leute
```

Man beachte, daß zwischen dem jeweiligen Umleitungszeichen ('`>`', '`:`' oder '`|`') und dem folgenden Namen, im Gegensatz zu den UNIX-Shells, kein Leerzeichen stehen darf. Da ein Leerzeichen als Trennung zwischen zwei Worten angesehen wird und nur jeweils das letzte Wort einer Kommandoeingabe überprüft wird, ob eine Ausgabeumleitung gewünscht ist, würde die Umleitung sonst nicht erkannt werden. In 1.4 wurde bereits erwähnt, daß sich die Trennung von Worten durch Leerzeichen umgehen läßt, indem die Gruppe zusammengehöriger Worte in Anführungszeichen gesetzt wird. Auf diese Weise lassen sich bei der Umleitung in einen Prozeß auch Argumente an das zu startende Programm übergeben: Um die gesamte Direktive zur Ausgabeumleitung werden Anführungszeichen geschrieben.

Beispiel:

```
>echo abc |tr b x ('tr' ist im Standard-UNIX das Zeichenübersetzungs-
(translate-) Kommando)
abc |tr b x (keine Ausgabeumleitung)
>echo abc "| tr b x"
axc
```

Das tatsächliche Schreiben der Variablen oder des Files findet erst nach dem Eintreffen der Confirmation statt. Wenn also die Variable oder das File weiterverwendet werden soll, muß vorher mit '`flush`' auf das Abarbeiten der Callbacks gewartet werden. Ähnliches gilt, wenn die Verbindung zu einem VED geschlossen werden soll. Da das Kommunikationssystem nach dem Schliessen ankommende Confirmations vernichtet, muß, wenn Ausgabeumleitungen verwendet wurden, ebenfalls ein '`flush`' angewendet werden.

1.9 Prozedurlisten

Prozedurlisten sind eine Folge von einzelnen, im Server auszuführenden, Prozeduraufrufen mit ihren Argumenten. Sie werden als Text eingegeben und vor der Übermittlung an den EMS-Server compiliert, d.h., in eine binäre Form gebracht. Dabei werden die alphanumerischen Prozedurnamen anhand einer Tabelle in numerische Indices übersetzt. Die Tabelle der implementierten Funktionen wird dazu vom jeweiligen Server angefordert (siehe '`getcaplist`').

In folgenden wird die Syntax der Prozedurlisten in einer einfachen Form dargestellt. Eine exakte (formale) Beschreibung ist im Referenzteil zu finden.

Eine Prozedurliste besteht aus von geschweiften Klammern eingefaßten, aneinandergereihten Prozeduraufrufen:

{ <*Prozeduraufruf*> ...}

Ein Prozeduraufruf besteht aus Prozedurnamen uns Argumenten:

<*Name*>[,<*Version*>](<*Argumente*>)

Der Prozedurname ist eine Folge von Buchstaben und Zahlen, das erste Zeichen muß ein Buchstabe sein. Optional kann eine Versionsnummer der Prozedur nach einem Komma angegeben werden. Wenn keine Version angegeben ist, wird die höchste Nummer verwendet.

Argumente sind kommasepariert:

<*Argument*>,<*Argument*>, ...

Zulässige Argumente sind:

- Zahlen in Dezimal-, Oktal- oder Hexadezimalformat
- Strings in Anführungszeichen (Backslash-Escape-Sequenzen sind, wie in 'C', erlaubt.) Die Zeichenketten werden als XDR-Strings übergeben.
- Variablenreferenzen ($<*Variablenname*>) Die Variable muß eine Liste von Zahlen (Dezimal- Oktal- oder Hexadezimalformat) enthalten. Ein solches Argument kann für mehrere Prozedurargumente stehen, je nach Variablenlänge.
- Eingaben aus Files mit festem oder variablem Filenamen (<<*Filename*> oder <$<*Variablenname*>) Das File muß Zahlen in ASCII-Kodierung (Dezimal- Oktal- oder Hexadezimalformat) enthalten. Ein solches Argument kann für mehrere Prozedurargumente stehen, je nach der Anzahl der Worte im File.
- Prozedurlisten (Steht für mehrere Argumente, nämlich die compilierte Form der Prozedurliste. Dies ist für spezielle Anwendungen vorgesehen.)

Zwischen den Elementen von Prozedurlisten sind Leerzeichen, Tabulatoren und Zeilenendezeichen bedeutungslos. Prozedurlisten können sich bei der Kommandoeingabe über mehrere Zeilen erstrecken (siehe 1.4).

Beispiele:

```
testved:1>set a 4 5 6
testved:1>docommand {Func1(1,$a,2)Func2()Func3("hallo")}
```
Func1 wird mit den 5 Argumenten '1', '4', '5', '6' und '2' aufgerufen, Func2 ohne Argumente, Func3 mit dem ins hardwareunabhängige XDR-Format gepackten fünfbuchstabigen String „hallo".
```
testved:1>downloadreadoutlist 10 1 { Func1 (3,
+4,5)
+}
```
Das Client-Programm fordert mit '+' weitere Eingabezeilen an, bis alle geöffnetenKlammern wieder geschlossen sind.

1.10 Weglassen von 'docommand', abgekürzte Prozedurlisten

Wenn das Client-Programm ein eingegebenes Kommando nicht deuten kann und eine Verbindung zu einem EMS-Server besteht, versucht es, die Eingabe als Prozedurliste zu interpretieren und mittels '`docommand`' an den Server zu senden. Das erlaubt, das Befehlswort '`docommand`' und auch die die Prozedurliste einschließenden geschweiften Klammern wegzulassen. Allerdings gibt bei dieser verkürzten Befehlseingabe drei Einschränkungen:

- Es kann keine Ausgabeumleitung stattfinden.
- Mehrzeilige Eingabe funktioniert nur, wenn die öffnende Klammer am Anfang eines Wortes steht. (Eine Eingabe

  ```
  testved:0>Func1(3,
  ```

  bewirkt also nicht die gewünschte Anforderung einer weiteren Eingabezeile, sondern eine „syntax error"-Meldung des Prozedurlistencompilers.)
- Wenn in einer Argumentliste auf ein Leerzeichen ein Anführungszeichen folgt, wird dies vom Parser als Start eines neuen Kommandowortes (miß-) verstanden. Das hat zur Folge, daß die in 1.4 geschilderten Mechanismen in Gang kommen (Entfernung der Anführungszeichen), was bei Prozedurlisten normalerweise nicht erwünscht ist.

1.11 Unsolicited Messages

Das Kommunikationssystem und die EMS-Server erzeugen beim Eintreffen besonderer Ereignisse Mitteilungen an die EMS-Clients. Die fatalsten von diesen werden unter allen Umständen geliefert; über den Empfang der anderen kann mit einem Kommando ('`setunsol`') entschieden werden. Die fatalen unsolicited Messages sind:

- Beendigung des Kommunikationsprozesses
  Das macht die Weiterarbeit unmöglich, das Client-Programm wird mit der Meldung „commu gestorben" beendet.

- Verlust der Verbindung zum aktuellen Server
  Es wird eine Meldung ausgegeben; der Prompt zeigt an, daß keine Verbindung mehr besteht. Danach können neue Verbindungen geöffnet werden, um die Arbeit fortzusetzen.

Andere Messagetypen sind:

- Log-Daten vom Server
- Mitteilungen über LAMs
- Hardwareprobleme oder Inkonsistenzen beim Readout, Fehler im Datenausgabegerät (Dies alles wird unter dem Oberbegriff „Runtime-Error“ zusammengefaßt. Die weitere Unterteilung erfolgt anhand einer Kennung im Datenbereich.)

Diese Nachrichten werden, wenn geliefert, formatiert auf dem Terminal ausgegeben. Dieses Verhalten kann, wenn nötig, durch Austausch einiger separat eingebundener C-Routinen geändert werden (siehe 1.13). So ist es denkbar, auf LAMs durch Abarbeitung spezieller Scripts zu reagieren.

1.12 Debug-Optionen

Zusätzlich zu den oben in 1.2 genannten Kommandozeilenoptionen existieren einige weitere, die insbesondere zur Fehlersuche verwendet werden können. Da diese vom normalen Benutzer kaum benutzt werden dürften, werden sie hier gesondert beschrieben.

Die nunmehr vollständige Syntax eines klient-Aufrufes lautet:

klient [-s <*socketname*>] [-h <*hostname*>] [-p <*portno*>] [-l]
[-- [-v] [-d] [-s]] [script]

Die Bedeutung der zusätzlichen Schalter ist:

`l`: (lokaler Modus) Es wird keine Verbindung zu einem Kommunikationsprozeß aufgenommen. Das hat zur Folge, daß keine Verbindungen zu VEDs aufgenommen werden können, also auch die meisten Kommandos unerreichbar bleiben. Der Nutzen dieses Modus liegt in Tests der Kommandozeilenverarbeitung und der lokalen Kommandos.

`v`: (verbose) Jeder Befehl wird mit seinen Argumenten vor der Abarbeitung ausgegeben.

`d`: (debug) Es werden zusätzliche Ausschriften erzeugt. In Verbindung mit der verbose-Option werden die einzelnen Worte jedes Befehles in Pfeile (z.B. '`->Wort<-`') eingefaßt, was den Test des Kommandozeilenparsers erleichtert.

`s`: (synchron) Nach jedem EMS-Befehl wird auf die zugehörige Confirmation gewartet. Diese Option vereinfacht besonders das Finden von Fehlern in Scripts, bei denen es unter Umständen schwierig ist, später eintreffende Fehlermeldungen den verursachenden Requests zuzuordnen. (Hier empfielt sich zusätzlich auch die verbose-Option.)

1.13 Erweiterungen

Für die unter „Unsolicited Messages“ (in 1.11) erwähnten, nichtfatalen Meldungen der EMS-Server gibt es keine allgemeine und zugleich optimale Reaktion. Deshalb wurde eine einfache Programmschnittstelle geschaffen, um das Standardverhalten (einfache Ausgabe) der jeweiligen Anwendung anpassen zu können.

Folgende Routinen werden beim Eintreffen solcher Nachrichten aufgerufen:

Wenn Log-Daten vom Server geschickt werden:

`user_data_handler(int type, int size, int *buf)`
`type`: Kennzeichnung des Datentyps, vom Absender vergeben
`size`: Anzahl der gesendeten Daten
`buf`: Zeiger auf das die Daten enthaltende Integerfeld

Beim Eintreten eines LAMs:

`user_lam_handler(int id, int size, int *buf)`
`id`: Kennung des LAMs (siehe '`downloaddomain`')
`size`: Anzahl der (von der zugeordneten Prozedurliste) erzeugten Daten
`buf`: Zeiger auf das die Daten enthaltende Integerfeld

Bei Problemen im Readout oder mit der Datenausgabe:

`user_rterr_handler(rterrcode err, int size, int*buf)`
`err`: Fehlercode, der Typ '`rterrcode`' ist im EMS-Quellfile common/errorcodes.h definiert.
`size`: Anzahl der gesendeten Diagnosedaten
`buf`: Zeiger auf das die Daten enthaltende Integerfeld
Die Interpretation der Daten ist vom Fehlercode abhängig. Hier wird auf die EMS-Spezifikation [4] verwiesen.

Die allgemeinen Standardroutinen für diese Messagehandler befinden sich im Quellfile dummy_userhandler.c. Für eine Anpassung an spezielle Bedürfnisse wären die genannten Funktionen zu ersetzen. Die ebenfalls dort befindliche Funktion
`user_test_handler()`
dient momentan internen Tests und sollte unverändert übernommen werden.

Einige Variablen und Routinen des Client-Programmes können von den externen Handlern verwendet werden:

- `extern char currentvedname[]`
  Name des momentan verbundenen VEDs, somit der Quelle der unsolicited Message.

- `print_formatted_hex(FILE *fd, int *buf, int size)`
  Ausgabe eines Integerfeldes bei `buf` mit der Größe `size` auf den Filedeskriptor `fd`. Dafür muß `<stdio.h>` eingebunden werden. (Für Terminalausgabe kann '`stdout`' als Filedeskriptor angegeben werden.) Die Daten werden als Hexadezimalzahlen formatiert.

- `int execute_file(char *name)`
  Ein Textfile mit dem übergebenen Filenamen wird als Kommandofile (Script) interpretiert. Die Funktion liefert als Resultat eine '1', wenn sie das File nicht öffnen kann, sonst eine '0'. Es muß damit gerechnet werden, daß beim Ausführen der Kommandos im File weitere unsolicited Messages empfangen werden und somit die Messagehandler rekursiv aufgerufen werden, sofern dem nicht entgegengewirkt wird (siehe '`setunsol`').

- `putinvar(char *varname, int datac, char **datav)`
  Der Inhalt des Stringvektors `datav` (Länge `datac`) wird der Variablen namens `varname` zugewiesen. Diese enthält dann `datac` Worte. Ein eventueller alter Inhalt wird überschrieben.

- `put_hex_in_var(char *varname, int *buf, int size)`
  Der Inhalt des Integerfeldes `buf` (Länge `size`) wird als Folge von Hexadezimalzahlen formatiert auf die Variable namens `varname` zugewiesen. Diese enthält dann `size` Worte. Ein eventueller alter Inhalt wird überschrieben.

- `char **getvar(char *varname, int *len)`
  Der Inhalt der Variablen namens `varname` wird als Stringarray zurückgegeben, die Länge im Rückgabeparameter `len`. Wenn die Variable nicht existiert, wird ein NULL-Pointer geliefert. Teilbereiche von Variablen können, wie oben in 1.5 beschrieben, durch Angabe des Bereiches mit '`[x:y]`' extrahiert werden. Das sonst zur Variablenreferenz verwendete Zeichen '`$`' darf nicht mit angegeben werden.

2 Kommandoreferenz

2.1 Lokale Kommandos

exit, quit, ende, bye

Aufruf: exit

Beendet das Programm

set

Aufruf: set <*Variablenname*> [<*Wert*> ...]

Wenn das Argument *Wert* angegeben ist, wird der Inhalt der Variablen namens *Variablenname* auf den angegebenen *Wert* gesetzt. Ein eventueller vorheriger Wert wird überschrieben. Eine Variable kann mehrere Worte beinhalten.

Wenn *Wert* fehlt, wird die Variable gelöscht.

echo

Aufruf: echo [<*Wort*> ...]

Die Argumente von '`echo`' werden auf den gewählten Pfad ausgegeben (default: Terminal). Wenn die Ausgabe auf das Terminal, in ein File oder in einen Prozeß erfolgt, wird ein Zeilenendezeichen angehängt.

Beispiele:

```
>echo Hallo
Hallo
>echo viele Worte >textfile
```
> *schreibt '`viele Worte`' in textfile. Existierende Files werden überschrieben.*

```
>echo 1 2 3 :setup
```
> *setzt die Variable '`setup`' auf den (aus 3 Worten bestehenden) Wert '`1`' '`2`' '`3`'*

```
>echo ps |sh
```
> *erzeugt einen Prozeß '`sh`' und schreibt '`ls`' in dessen Standardeingabepfad. Die Ausgabe wird auf das Terminal geleitet. (In diesem Fall sollte ein Directory-Listing ausgegeben werden.)*

source

Aufruf: source <*filename*>

Das File mit dem angegebenen Namen wird eingelesen, sein Inhalt wird zeilenweise als Kommandofolge interpretiert.

vars

Aufruf: vars

Es wird eine Tabelle ausgegeben, die die Namen der derzeitig definierten internen Variablen und die Anzahl der in ihnen enthaltenen Worte enthält.

Beispiel:

```
>vars
```
> *(Es sind noch keine Variablen definiert.)*

```
>set v1 eins 2 drei
>set v2 12345678
>vars
 v1  3   (d.h., 3 Worte)
 v2  1
>echo $v1
eins 2 drei
>set v1 (wird gelöscht, siehe 'set')
>vars
 v2  1
```

open

Aufruf: open <*VED-Name*>

Öffnet eine logische Verbindung zum angegebenen VED. Alle folgenden EMS-Kommandos werden an dieses VED gesendet. Eine eventuell bestehende Verbindung wird vorher geschlossen.

close

Aufruf: close

Schließt die Verbindung zum gegenwärtig geöffneten VED.

setunsol

Aufruf: setunsol <*Flag*>

Hiermit wird festgelegt, ob unsolicited messages des derzeit geöffneten VED empfangen werden sollen. *Flag*=0 heißt „nein“, Zahlen ungleich 0 bedeuten „ja“. Nach dem Öffnen eines VED ist „nein“ eingestellt.

is

Aufruf: is <*Nummer*>

Bewirkt, daß folgende EMS-Kommandos, die sich auf ein Instrumentierungssystem beziehen, an das angegebene Instrumentierungssystem geschickt werden. Nach dem Öffnen eines VED ist '0' eingestellt.

and

Aufruf: and <*Variablenname*> <*Operand1*> <*Operand2*>

Das logische UND der beiden (numerischen) Operanden wird in der angegebenen Variablen abgelegt.

or

Aufruf: or <*Variablenname*> <*Operand1*> <*Operand2*>

Das logische ODER der beiden (numerischen) Operanden wird in der angegebenen Variablen abgelegt.

not

Aufruf: not <*Variablenname*> <*Operand*>

Die logische Negation des (numerischen) Operanden wird in der angegebenen Variablen abgelegt.

plus

Aufruf: plus <*Variablenname*> <*Operand1*> <*Operand2*>

Die Summe der beiden (numerischen) Operanden wird in der angegebenen Variablen abgelegt.

minus

Aufruf: minus <*Variablenname*> <*Operand1*> <*Operand2*>

Die Differenz der beiden (numerischen) Operanden wird in der angegebenen Variablen abgelegt.

flush

Aufruf: flush

Wartet, bis alle vorher abgesetzten EMS-Kommandos durch Confirmations beantwortet worden sind.

convert

Aufruf: convert <*Format*> <*Zahl*> ...

Die (numerischen) Argumente werden nach den Vorgaben in *Format* konvertiert und auf den gewählten Pfad ausgegeben (default: Terminal). Wenn die Ausgabe auf das Terminal, in ein File oder in einen Prozeß erfolgt, wird ein Zeilenendezeichen angehängt. (siehe auch obige Beschreibung von '`echo`')

Format wird Zeichen für Zeichen interpretiert, es wird jeweils ein Ausgabewort im gewünschten Format aus der nötigen Anzahl von Eingabeworten erzeugt. Wenn nicht ausreichend Argumente vorhanden sind, werden die überschüssigen Zeichen des Format-Wortes ignoriert. Folgende Formate werden unterstützt:

x: Stellt ein Argument im Hexadezimalformat dar.

d: Stellt ein Argument im Dezimalformat dar.

o: Stellt ein Argument im Oktalformat dar.

f: Interpretiert ein Argument als (single-precision-) Gleitkommazahl und gibt diese aus. Das Gleitkommaformat ist durch XDR[1] vorgegeben, es entspricht IEEE 754.

%: Ein Argument wird ignoriert.

$: Alle folgenden Argumente werden ignoriert.

s: Die folgenden Argumente werden als XDR-gepackte Zeichenkette interpretiert. Das Resultat wird ausgegeben. Die Anzahl der benötigten Argumente geht aus dem XDR-String selbst hervor. Wenn nicht genug Argumente vorhanden sind, um die Zeichenkette konsistent abzuschließen, wird eine Fehlermeldung ausgegeben.

[<*Namensraum*>]: Ein Argument wird in eine symbolische Bezeichnung umgewandelt. So kann die EMS-interne numerische Repräsentation von Objekttypen und anderen Bezeichnern (Liste in 3) in lesbare Form überführt werden. Für <*Namensraum*> sind folgende symbolischen Bezeichnungen definiert (zulassige Abkürzungen sind ebenfalls aufgelistet):

[1] eXternal Data Representation[5]. Die in EMS verwendeten Datenformate folgen dieser Spezifikation.

capabylitytyp, cap: erzeugt Capabilitynamen

objecttyp, obj: erzeugt Objektnamen

domaintyp, dom: erzeugt Domainnamen

inouttyp, io: erzeugt InOut-Namen

addresstyp, addr: erzeugt Adreßtypnamen

programinvocationtyp, pi: erzeugt Programminvocationsnamen

Wenn das Argument keinem der im angegebenen Namensraum definierten Symbole entspricht, wird das Wort „`unknown`“ generiert.

Wenn das Formatwort nicht genug Einträge für alle Argumente enthält, werden die überhängenden Argumente als Hexadezimalzahlen ausgegeben.

Beispiele:

```
>convert xdo 0x00000005 0x48616c6c 0x6f000000
0x00000005 1214344300 015700000000
>convert s 0x00000005 0x48616c6c 0x6f000000
Hallo
>convert x%x$ 1 2 3 4 5 6 :setup
> setzt die Variable 'setup' auf den (aus 2 Worten bestehenden)
Wert '0x00000001' '0x00000003'
```

Das Kommando '`convert`' ist besonders für das Auswerten von Confirmations geeignet. Ein Beispiel dafür findet sich in der Beschreibung des EMS-Kommandos '`downloaddomain`'.

2.2 EMS-Kommandos

getcaplist

Aufruf: `getcaplist [<Typ>]`

Erzeugt einen GetCapabilityList-Request und installiert einen Callback, der die Confirmation als Tabelle formatiert auf das Terminal ausgibt (keine Ausgabeumleitung möglich). Als *Typ* ist endweder eine Zahl oder ein gültiger Capabilityname (siehe Liste der symbolischen Namen in 3) zulässig. Defaultwert für *Typ* ist '`Capab_listproc`'.

Beispiele:

```
testved:0>getcaplist
(Default wird wirksam!)
testved:0>0:  Funktion Echo, Ver.  1
1:  Funktion IdentIS, Ver.  1
2:  Funktion Timestamp, Ver.  1
⋮  (Tabelle enthält Index, Name, Versionsnummer)
testved:0>getcaplist trig
('trig' ist Abkürzung!)
```

```
testved:0>0:  Funktion Immer, Ver.  1
1:  Funktion Nie, Ver.  1
⋮
```

getnamelist

Aufruf: getnamelist <*Typ*> [<*Untertyp*> [<*Nummer*> ...]]

Erzeugt einen GetNameList-Request. Als *Typ* ist entweder eine Zahl oder ein gültiger Objektname (siehe 3) zulässig. Falls *Typ* den Objekttyp '`Domain`' bezeichnet, ist als *Untertyp* ein gültiger Domainname zulässig. Analoges gilt für den *Typ* '`Programminvocation`'. Als *Untertyp* und <*Nummer*> werden numerische Argumente akzeptiert.

resetved

Aufruf: resetved

Sendet einen ResetVED-Request.

identved

Aufruf: identved [<*was*>]

Erzeugt einen Identify-Request und installiert einen Callback, der die Confirmation formatiert auf das Terminal ausgibt (keine Ausgabeumleitung möglich). Der Parameter *was* (Zahl oder Schlüsselwort) bestimmt die Menge der gelieferten Informationen: '`version`' (`0`) liefert nur die Versionsnummern der Server-Implementierung und der Requesttabelle, '`name`' (`1`) zusätzlich Namen und Erzeugungsdatum des Server-Programmes und '`all`' (`2`) außerdem den Wortlaut des die installierten Optionen enthaltenden Konfigurationsfiles. Default ist '`all`'.

docommand

Aufruf: docommand <*Prozedurliste*>

Erzeugt einen DoCommand-Request für das derzeit gewählte Instrumentierungssystem (siehe '`is`'). Dazu wird die *Prozedurliste* compiliert, d.h., aus der Textform in eine Binärform überführt. Falls noch nicht vorher geschehen, wird dafür mittels GetCapabilityList-Request die Liste der im aktuellen VED implementierten Prozeduren ausgelesen.

Beispiel:

```
testved:0>docommand {Echo(1,2,3)}
```
(Voraussetzung: in '`testved`' ist eine Prozedur '`Echo`' implementiert.)
```
testved:0>0x00000001 0x00000002 0x00000003
```
(Das ist die Confirmation, ohne spezielle Formatierung.)

uploaddomain

Aufruf: uploaddomain <*Untertyp*> [<*Nummer*>]

Erzeugt einen UploadDomain-Request. Als *Untertyp* ist eine Zahl oder ein gültiger Domainname (siehe 3) zulässig. *Nummer* ist ein numerischer Index, Default ist '0'. Falls *Untertyp* den Domaintyp '`Dom_Event`' bezeichnet, wird ein Callback installiert, der die Eventdaten lesbar aufbereitet. Die üblichen Ausgabeumleitungen werden unterstützt, bei Ausgabe von Eventdaten in einen Prozeß (| <*Programmname* >) werden diese binär übergeben.

downloaddomain

Aufruf: downloaddomain <*Untertyp*> <*Nummer*> <*Daten*>

Erzeugt einen DownloadDomain-Request. Als *Untertyp* ist eine Zahl oder ein gültiger Domainname (siehe 3) zulässig. *Nummer* ist ein numerischer Index. Das Format von *Daten* ist vom *Untertyp* abhängig. Folgende Formate sind denkbar:

Dom_Modullist: {<*Adresse*> <*Typ*>} ...
: *Adresse* und *Typ* sind numerische Argumente, die immer paarweise angegeben werden müssen. Für *Typ* werden auch sie in 3 aufgelisteten Schlüsselworte akzeptiert. Die oben erwähnte *Nummer* ist bedeutungslos, da nur jeweils eine Modulliste pro VED geladen werden kann.

Dom_Trigger: <*Funktion*>[,<*Version*>] <*Funktionsargumente*>
: *Funktion* ist der Name einer im aktuellen VED implementierten Triggerfunktion. Dieser wird automatisch in einen numerischen Index umgerechnet. Falls noch nicht vorher geschehen, wird dafür mittels GetCapabilityList-Request die Liste der im aktuellen VED implementierten Triggerprozeduren ausgelesen. Durch Angabe einer *Version* kann eine bestimmte Versionsnummer ausgewählt werden, sonst wird die letzte (höchste) Nummer verwendet. Die folgenden (numerischen) Argumente werden der Triggerprozedur bei der Initialisierung als Parameter übergeben. Die oben erwähnte *Nummer* ist bedeutungslos, da nur jeweils eine Triggerdefinition pro VED geladen werden kann.

Dom_LAMproclist: <*Outputflag*> <*Prozedurliste*>
: *Outputflag* entscheided, ob beim Auftreten des LAMs eine unsolicited message, bestehend aus der in '`createprograminvocation`' (mit Untertyp `LAM` und dem gleichen Index *Nummer* wie hier) festgelegten `ID` und eventuellen Ausgabedaten von *Prozedurliste*, abgeschickt werden soll. <*Outputflag*>=0 heißt „nein“, Zahlen ungleich 0 bedeuten „ja“. *Prozedurliste* wird beim Auftreten des LAMs mit dem (hardwareabhängig interpretierten) Index <*Nummer*> abgearbeitet. Vor dem Abschicken wird die <*Prozedurliste*> compiliert, d.h., aus der Textform in eine Binärform überführt. Falls noch nicht vorher geschehen, wird dafür mittels GetCapabilityList-Request die Liste der im aktuellen VED implementierten Prozeduren ausgelesen.

Dom_Datain: <*InOutTyp*> <*Adreßtyp*> <*Adresse*>

InOutTyp ist entweder eine Zahl oder ein gültiger InOut-Name (siehe 3). *Adreßtyp* ist entweder ebenfalls eine Zahl oder ein Adreßtypname. Die Interpretation von <*Adresse*> hängt vom <*Adreßtyp*> ab. Folgende Formate sind denkbar:

Addr_Raw: <*Speicheradresse*>

Das Argument ist eine numerisch angegebene Speicherposition.

Addr_Modul: <*Modulname*>

Das Argument ist eine Zeichenkette, die vom Server-System als Speichermodulname interpretiert wird.

Addr_Driver_syscall:

Addr_Driver_mixed:

Addr_Driver_mapped: <*Pfadname*> [<*Space*> [<*Offset*> [<*Option*>]]]

Pfadname ist eine Zeichenkette, die im Server-System als Treibername interpretiert wird. *Space* ist eine Zahl, die der Unterscheidung mehrerer Adreßräume dient. *Offset* ist eine numerische Adresse, die vom Treiber interpretiert wird. Mit der Zahl *Option* können spezielle (hardwareabhängige) Aktionen im Server veranlaßt werden. Defaultwert für <*Space*>, <*Offset*> und <*Option*> ist '0'.

Dom_Dataout: <*InOutTyp*> <*Buffersize*> <*Priorität*> <*Adreßtyp*> <*Adresse*>

InOutTyp ist entweder eine Zahl oder ein gültiger InOut-Name (siehe 3). *Buffersize* und *Priorität* sind numerische Werte. *Adreßtyp* ist entweder ebenfalls eine Zahl oder ein Adreßtypname. Die Interpretation von *Adresse* hängt vom *Adreßtyp* ab. Folgende Formate sind denkbar:

Addr_Raw: <*Speicheradresse*>

Das Argument ist eine numerisch angegebene Speicherposition.

Addr_Modul: <*Modulname*>

Das Argument ist eine Zeichenkette, die vom Server-System als Speichermodulname interpretiert wird.

Addr_LocalSocket: <*Socketname*>

Das Argument ist eine Zeichenkette, die vom Server-System als Name eines UNIX-Domain-Sockets interpretiert wird.

Addr_File:

Addr_Tape: <*Pfadname*>

Das Argument ist eine Zeichenkette, die vom Server-System als Pfadname interpretiert wird.

Addr_Socket: <*Host*> <*Port*>

Host ist die Internet-Adresse des Rechners, der die Daten empfangen soll, in Dot-Notation, als Zahl oder als Hostname (wird vom Clientprogramm aufgelöst). *Port* ist die Nummer des empfangenden TCP-Ports.

Addr_Null:

Es sind keine weiteren Angaben nötig.

Beispiele:

`testved:0>downloaddomain trig 0 Immer 1`
(Voraussetzung: in '`testved`' ist eine Triggerprozedur '`Immer`' implementiert. Hiermit wird die lokale Triggerprozedur '`Immer`' mit einem Argument '`1`' für ein folgendes Readout ausgewählt.)
`testved:0>downloaddomain dataout 3 ringbuffer 100000 0 tape /xa0`
(Hiermit wird die Domain Nr. 3 vom Typ '`dataout`' geladen. Sie definiert die Datenausgabe auf ein Bandgerät, das unter dem Treibernamen '`/xa0`' angesprochen wird. Die Daten werden in einem 100'000 *Byte großen Puffer zwischengespeichert.)*
`testved:0>uploaddomain dataout 3 :td`
Die eben geladene Domain wird aus dem VED zurückgelesen. Das Resultat wird in der Variablen '`td`' abgelegt.
`testved:0>convert [io]dd[addr]s $td`
`InOut_Ringbuffer 100000 0 Addr_Tape /xa0`
Die Daten in der Variablen '`td`' werden in das durch das erste Argument von `convert` *gegebene Format umgewandelt und angezeigt. (Näheres in der Beschreibung zu* `convert`*.)*

deletedomain

Aufruf: deletedomain <*Untertyp*> [<*Nummer*>]

Erzeugt einen DeleteDomain-Request. Als *Untertyp* ist eine Zahl oder ein gültiger Domainname (siehe 3) zulässig. *Nummer* ist ein numerischer Index, Default ist '0'.

createprograminvocation

Aufruf: createprograminvocation <*Untertyp*> [<*Nummer*> [<*Definition*>]]

Erzeugt einen CreateProgramInvocation-Request. *Untertyp* ist entweder ein gültiger Programminvocations-Name (siehe 3) oder eine Zahl. *Nummer* ist ein numerischer Index, Default ist '0'. Falls *Untertyp* den Typ '`Invocation_LAM`' bezeichnet, wird *Definition* benötigt. Diese hat die Form:

<*ID*> <*Instrumentierungssystem*> [<*Argument*>]

Die 2 oder 3 numerischen Argumente haben folgende Bedeutung: *ID* ist die Identifikation der bei einer LAM-Behandlung anfallenden Daten, *Instrumentierungssystem* wird zur Ausführung der zugeordneten Prozedurliste (siehe '`downloaddomain`') verwendet und *Argument* wird beim Start der LAM-Behandlung der (hardwareabhängigen) Initialisierungsroutine übergeben. *Nummer* ist dann der hardwareabhängig interpretierte Index des LAMs.

deleteprogramminvocation

Aufruf: deleteprograminvocation <*Untertyp*> [<*Nummer*>]

Erzeugt einen DeleteProgramInvocation-Request. *Untertyp* ist entweder ein gültiger Programminvocations-Name (siehe 3) oder eine Zahl. *Nummer* ist ein numerischer Index, Default ist '0'.

startprograminvocation, start

Aufruf: startprograminvocation <*Untertyp*> [<*Nummer*>]

Erzeugt einen StartProgramInvocation-Request. *Untertyp* ist entweder ein gültiger Programminvocations-Name (siehe 3) oder eine Zahl. *Nummer* ist ein numerischer Index, Default ist '0'.

resetprograminvocation, reset

Aufruf: resetprograminvocation <*Untertyp*> [<*Nummer*>]

Erzeugt einen ResetProgramInvocation-Request. *Untertyp* ist entweder ein gültiger Programminvocations-Name (siehe 3) oder eine Zahl. *Nummer* ist ein numerischer Index, Default ist '0'.

stopprograminvocation, stop

Aufruf: stopprograminvocation <*Untertyp*> [<*Nummer*>]

Erzeugt einen StopProgramInvocation-Request. *Untertyp* ist entweder ein gültiger Programminvocations-Name (siehe 3) oder eine Zahl. *Nummer* ist ein numerischer Index, Default ist '0'.

resumeprograminvocation, resume

Aufruf: resumeprograminvocation <*Untertyp*> [<*Nummer*>]

Erzeugt einen ResumeProgramInvocation-Request. *Untertyp* ist entweder ein gültiger Programminvocations-Name (siehe 3) oder eine Zahl. *Nummer* ist ein numerischer Index, Default ist '0'.

getprograminvocationattributes

Aufruf: getprograminvocationattributes <*Untertyp*> [<*Nummer*>]

Erzeugt einen GetProgramInvocationAttributes-Request und installiert einen Callback, der die Confirmation formatiert auf das Terminal ausgibt (keine Ausgabeumleitung möglich). *Untertyp* ist entweder ein gültiger Programminvocations-Name (siehe 3) oder eine Zahl. *Nummer* ist ein numerischer Index, Default ist '0'.

createvariable

Aufruf: createvariable <*Index*> <*Größe*>

Erzeugt einen CreateVariable-Request. *Index* und *Größe* sind numerisch.

deletevariable

Aufruf: deletevariable <*Index*>

Erzeugt einen DeleteVariable-Request. *Index* ist numerisch.

readvariable

Aufruf: `readvariable <Index>`

Erzeugt einen ReadVariable-Request und installiert einen Callback, der den Variableninhalt lesbar aufbereitet. Die übichen Ausgabeumleitungen werden unterstützt, außer der Ausgabe in einen Prozeß. *Index* ist numerisch.

writevariable

Aufruf: `writevariable <Index> <Inhalt>`

Erzeugt einen WriteVariable-Request. *Index* ist numerisch. *Inhalt* ist eine Folge von Zahlen.

getvariableattributes

Aufruf: `getvariableattributes <Index>`

Erzeugt einen GetVariableAttributes-Request. *Index* ist numerisch.

createis

Aufruf: `createis <Index> <ID>`

Erzeugt einen CreateIS-Request. *Index* ist numerisch. *ID* ist die (numerische) Identifikation der bei einer Auslese des Instrumentierungssystems anfallenden Daten. Der Instrumentierungssystem-Index für nachfolgende Requests wird auf *Index* gesetzt (siehe '`is`').

deleteis

Aufruf: `deleteis`

Erzeugt einen DeleteIS-Request. Es wird das gegenwärtig aktive Instrumentierungssystem gelöscht und der Instrumentierungssystem-Index für nachfolgende Requests auf '0' gesetzt (siehe '`is`').

enableis

Aufruf: `enableis`

Erzeugt einen EnableIS-Request. Es wird das gegenwärtig aktive Instrumentierungssystem angewählt.

disableis

Aufruf: `disableis`

Erzeugt einen DisableIS-Request. Es wird das gegenwärtig aktive Instrumentierungssystem angewählt (und dadurch vom nächsten zu startenden Readout ausgeschlossen).

getisstatus

Aufruf: `getisstatus`

Erzeugt einen GetISStatus-Request. Es wird das gegenwärtig aktive Instrumentierungssystem angewählt.

downloadismodullist

Aufruf: `downloadismodullist <Adressen>`

Erzeugt einen DownloadISModulList-Request. Dieser wird an das derzeit aktive Instrumentierungssystem gesendet. *Adressen* ist eine Folge von numerischen Argumenten.

uploadismodullist

Aufruf: `uploadismodullist`

Erzeugt einen UploadISModulList-Request. Dieser wird an das derzeit aktive Instrumentierungssystem gesendet.

deleteismodullist

Aufruf: `deleteismodullist`

Erzeugt einen DeleteISModulList-Request. Dieser wird an das derzeit aktive Instrumentierungssystem gesendet.

downloadreadoutlist

Aufruf: `downloadreadoutlist <Priorität> <Triggerliste> <Readoutliste>`

Erzeugt einen DownloadReadoutList-Request. Dieser wird an das derzeit aktive Instrumentierungssystem gesendet. *Priorität* ist eine Zahl. *Triggerliste* ist eine Folge von Zahlen, die im Server als Triggerbedingungen interpretiert werden. Die *Prozedurliste* wird vor dem Absenden compiliert, d.h., aus der Textform in eine Binärform überführt. Falls noch nicht vorher geschehen, wird dafür mittels GetCapabilityList-Request die Liste der im aktuellen VED implementierten Prozeduren ausgelesen.

uploadreadoutlist

Aufruf: `uploadreadoutlist <Triggerbedingung>`

Erzeugt einen UploadReadoutList-Request. Dieser wird an das derzeit aktive Instrumentierungssystem gesendet. *Triggerbedingung* ist eine Zahl im Bereich der in Server definierten Triggerbedingungen.

deletereadoutlist

Aufruf: `deletereadoutlist <Triggerbedingung>`

Erzeugt einen DeleteReadoutList-Request. Dieser wird an das derzeit aktive Instrumentierungssystem gesendet. *Triggerbedingung* ist eine Zahl im Bereich der in Server definierten Triggerbedingungen.

winddataout, wind

Aufruf: winddataout <*Index*> <*Offset*>

Erzeugt einen WindDataout-Request. *Index* ist eine Zahl, *Offset* ein numerisches Argument, das die (positive uder negative) Anzahl der auf dem Datenspeichermedium zu überspringenden Files vorgibt.

writedataout, write

Aufruf: writedataout <*Index*> <*Header*> <*Datenworte*>

Erzeugt einen WriteDataout-Request. *Index* ist eine Zahl. *Header* ist ein Flag, das festlegt, ob vor den Datenblock ein an das Eventdatenformat angepaßter Header geschrieben werden soll. *Header*=0 heißt „nein“, Zahlen ungleich '0' bedeuten „ja“. Alle folgenden Argumente werden als Zeichenketten interpretiert und auf das angegebene Datenspeichergerät geschrieben. Runde und geschweifte Klammern sollten hier vermieden werden, da diese den primitiven Parser des Client-Programmes desorientieren können.

dumpfile

Aufruf: dumpfile <*Index*> <*Header*> <*Filename*>

Erzeugt einen WriteDataout-Request. *Index* ist eine Zahl. *Header* ist ein Flag, das festlegt, ob vor den Datenblock ein an das Eventdatenformat angepaßter Header geschrieben werden soll. *Header*=0 heißt „nein“, Zahlen ungleich '0' bedeuten „ja“. Der Datenblock besteht aus dem Inhalt des Files mit dem angegebenen Namen. Dieses File wird zeilenweise eingelesen und in XDR-String-Form konvertiert. Die maximale Zeilenlänge ist auf 1024 Zeichen begrenzt. Im Gegensatz zum vorherigen Kommando wird hier der Inhalt des Files nicht interpretiert, d.h., es können beliebige Textzeichen verwendet werden.

getdataoutstatus

Aufruf: getdataoutstatus <*Index*>

Erzeugt einen GetDataoutStatus-Request. *Index* ist eine Zahl.

enabledataout, enable

Aufruf: enabledataout <*Index*>

Erzeugt einen EnableDataout-Request. *Index* ist eine Zahl.

disabledataout, disable

Aufruf: disabledataout <*Index*>

Erzeugt einen DisableDataout-Request. *Index* ist eine Zahl.

3 Gültige symbolische Namen

Die Namen entsprechen den Definitionen im EMS-Quellfile common/objecttypes.h. Wo angebracht, wurden zusätzlich Abkürzungen zugelassen. Diese sind jeweils hinter den Namen aufgelistet.
Capabilitynamen:

```
Capab_listproc, proc
Capab_trigproc, trig
```

Objektnamen:

```
Object_ved, ved
Object_domain, domain, dom
Object_is, is
Object_var, var
Object_pi, pi
Object_do, do
```

Domainnamen:

```
Dom_Modullist, ml
Dom_LAMproclist, LAM
Dom_Trigger, trig
Dom_Event, ev
Dom_Datain, datain
Dom_Dataout, dataout
```

Programminvocationsnamen:

```
Invocation_readout, readout, ro
Invocation_LAM, LAM
```

InOut-Namen:

```
InOut_Ringbuffer, ringbuffer
```

Adreßtypnamen:

```
Addr_Raw, raw
Addr_Modul, modul
Addr_Driver_mapped, driver_mapped
Addr_Driver_mixed, driver_mixed
Addr_Driver_syscall, driver_syscall
Addr_Socket, socket
Addr_LocalSocket, localsocket
Addr_File, file
Addr_Tape, tape
Addr_Null, null
```

Modultypnamen:

Es werden (noch) keine Modultypnamen unterstützt.

4 Grammatik der Prozedurlisten

4.1 Formale Grammatik

Die Grammatik ist in einer vereinfachten Backus-Naur-Form beschrieben. Sonderzeichen wie '`{`', '`<`' und '`,`' sind unverändert zu übernehmen, Worte in Klammern ('*(...)*') sind Kommentare. Alternativen stehen untereinander und sind durch '|' abgetrennt. 'ε' ist ein Metazeichen und bedeutet ein leeres (nicht vorhandenes) Element.

<Prozedurliste>	⇒	`{`*<Prozeduraufrufe>*`}`
<Prozeduraufrufe>	⇒	*<Prozeduraufrufe><Prozeduraufruf>*
	|	ε
<Prozedurliste>	⇒	`{`*<Prozeduraufrufe>*`}`
<Prozeduraufrufe>	⇒	*<Prozeduraufrufe><Prozeduraufruf>*
	|	ε
<Prozeduraufruf>	⇒	*<Funktionsname>*`(`*<Argumente>*`)`
<Funktionsname>	⇒	*<Zahl> (Index der Prozedur)*
	|	*<Wort> (Prozedurname)*
	|	*<Wort>*`,`*<Zahl> (Name mit Versionsnummer)*
<Argumente>	⇒	*<Argumente>*`,`*<Argument>*
	|	*<Argument>*
	|	ε
<Argument>	⇒	*<Zahl>*
	|	*<Wort>*
	|	*<String> (wird in XDR konvertiert)*
	|	*<Prozedurliste>*
<Wort>	⇒	*<Name>*
	|	`<`*<Name> (in Argumenten: Eingabe aus File)*
	|	`$`*<Name> (in Argumenten: Eingabe aus Variabler)*
	|	`<$`*<Name>*
		(in Argumenten: Eingabe aus File mit variablem Namen)

4.2 Syntax der einzelnen Bestandteile

Zur Beschreibung der verwendbaren Zeichen wird die Form der „regulären Ausdrücke“ (regular expressions) verwendet. In eckigen Klammern ('[...]') werden Mengen aufgelistet, aus denen jeweils ein beliebiges Element verwendet werden kann. Mit Bindestrich ('−') wird dabei jeweils eine zusammenhängende Folge von Zeichen beschrieben. Ein Stern ('∗') bezeichnet ein beliebig häufiges $(0 - \infty)$ Auftreten des vorhergehenden Elements, ein Plus ('+') die beliebige Wiederholung $(1 - \infty)$ des Vorgängers.

<Zahl>	$\Rightarrow$	`0`
	|	`[1 - 9][0 - 9]` $*$ *(Dezimalzahl)*
	|	`0[Xx][0 - 9A - Fa - f]` $+$ *(Hexadezimalzahl)*
	|	`0[0 - 7]` $+$ *(Oktalzahl)*
<Name>	$\Rightarrow$	`[A - Za - z_][A - Za - z_0 - 9]` $*$
<String>	$\Rightarrow$	`"`*<beliebig>* $*$ `"`*(entspricht Strings in 'C', '\' ist Escape-Zeichen)*

Anmerkung: Anführungszeichen ('`"`') innerhalb von Strings müssen durch ein vorhergehendes Escape-Zeichen gekennzeichnet werden. Ein Null-Byte im String ('`\0`') wird nicht als Ende-Zeichen angesehen, sondern wörtlich übertragen.

Literatur

[1] K. Zwoll, M. Drochner, W. Erven, J. Holzer, H. Kopp, and P. Wüstner. Objektorientierte Modellierung des on-line-Datenerfassungssystems für COSY-Experimente. Internal Report IB-KFA-ZEL 501392, KFA Jülich / ZEL, 1992.

[2] K. Zwoll, M. Drochner, W. Erven, J. Holzer, H. Kopp, H. W. Loevenich, P. Wüstner, K.-H. Watzlawik, N. Brummund, M. Karnadi, R. Nellen, J. Stock, S. Dienel, K.-H. Leege, and W. Oehme. Flexible data aquisition system for experiments at COSY. *IEEE Trans. Nucl. Sci.*, 41(1):37–44, 1994.

[3] Brian Fox. *GNU Readline Library.* Free Software Foundation, 1.1 edition, April 1991.

[4] M. Drochner. Spezifikation der EMS-Messages (Typen und Formate). Technical report, KFA Jülich / ZEL, 1994.

[5] Sun Mikrosystems. RFC 1014: XDR: External data representation standard, 1987.

Verzeichnis der Kommandos