Script-Sprache für programmierbare Terminals

Autor: Wolfgang Büscher, MKT Systemtechnik
Datum: 2021-05-05 (ISO 8601)
Quelldatei ("Master"): <WB/ProgrammingTool>..help\scripting_49.htm
Online: www.mkt-sys.de/MKT-CD/upt/help/scripting_49.htm

(Hinweis: In der druckbaren Variante dieser Datei, ?/Doku/art85122_UPT_Scripting_DE.pdf, funktionieren die externen Links nicht.)

Inhalt

1. Einleitung
    1. Prinzip
    2. Freischaltbare Funktionen (erweiterte Script-Funktionen)
2. Script-Editor mit integriertem Debugger
    1. Editor Toolbar
    2. Hotkeys und Kontext-Menüs des Script-Editors
    3. Debugging
        1. Breakpoints, Single-Step
        2. Disassembler
        3. Trace
        4. Stack-Anzeige
        5. Symbol-Tabelle mit Variablen, Funktionsnamen, etc

6. Überwachte Ausdrücke ("Watch-Liste")
7. Anzeige der vom Script dynamisch allozierten Speicherblöcke
8. Testen der Applikation im RAM (statt FLASH) des Zielsystems

3. Interaktion zwischen Script und Display (d.h. den "programmierbaren Anzeigeseiten")
    1. Aufruf einer Script-Prozedur beim Betätigen eines Buttons
    2. Ändern eines Anzeige-Elements per Script (Text, Farben, usw.)
4. Sprachbeschreibung (Referenz)
    1. Numerische Werte und Konvertierung von numerischen Datentypen
    2. Strings (Zeichenketten)
        1. Zeichenketten mit verschiedenen Kodierungen ("DOS", "ANSI", "Unicode")
        2. Verwendung von Zeichenketten und deren Format im Speicher
        3. String-Konstanten mit Sonderzeichen
        4. Strings mit Backslash-Sequenzen
        5. Funktionen zur String-Verarbeitung :
           append chr ansi_chr unicode_chr CharAt char_encoding
           atoi atof itoa ftoa hex HexString BinaryString
           strlen strpos strpos2 stripos strrpos strripos substr
           ParseInteger ParseFloat ParseHexString ParseBinaryString
    3. Konstanten
        1. Fest eingebaute Konstanten
        2. Anwenderdefinierte Konstanten
        3. 'Berechnete' Konstanten
        4. Konstanten-Arrays ("Tabellen mit festem Inhalt")
    4. Datentypen (fest und benutzerdefiniert)
        1. int, float, double, string, byte, word, dword, bool, tColor
        2. anytype
        3. tMessage, tCANmsg, tScreenCell, tCanvas, tTimer, tTable, tDirEntry
        4. Explizite Typumwandlung (typecast)
    5. Variablen
        1. Variablen-Deklarationen im Script
            1. Globale Variablen
            2. Lokale Variablen
            3. Pointer-Variablen (Zeiger)
            4. Die Attribute 'private', 'public', 'logged' und 'noinit'

2. Zugriff auf Script-Variablen aus dem Anzeigeprogramm
3. Zugriff auf Anzeige-Variablen aus der Script-Sprache

6. Arrays
   1. Maximale 'Größe' (size) und momentan verwendete 'Länge' (len) eines Arrays
   2. Übergabe von Array-Referenzen (um Kopieren zu vermeiden)
   3. Beispiele für die Verwendung von Arrays
7. Operatoren
8. Anwenderdefinierte Funktionen und Prozeduren
   1. Anwenderdefinierte Prozeduren
   2. Anwenderdefinierte Funktionen
   3. Aufruf eigener Funktionen per Backslash-Sequenz aus den programmierbaren Anzeigeseiten
   4. Aufruf eigener Prozeduren aus dem Display-Interpreter
   5. Funktions- Ein- und Ausgänge in der Parameterliste ("in", "out")
   6. Rekursive Prozeduraufrufe
9. Ablaufsteuerung in der Script-Sprache (Schleifen, Verzweigungen, etc)
   1. if, then, else, endif
   2. for, to, step, next
   3. while .. endwhile
   4. repeat .. until
   5. select, case, else, endselect
10. Weitere Funktionen und Kommandos
   1. Numerische Funktionen, "Mathematik", Digitale Signalverarbeitung
   2. Timer und Stoppuhren
   3. Mehrzeilige Textfelder (text panels): cls, gotoxy, print & Co
   4. Canvas-Funktionen (Zeichnen auf einer im Script deklarierten 'Leinwand')
   5. Datei-Ein/Ausgabe (file.create, file.open, file.write, file.read, file.close, ...)
   6. Senden und Empfangen von CAN-Telegrammen (per Script), CAN-Diagnose
   7. Steuern der Anzeigeseiten per Script / Steuern von Diagrammen / Display-Variablen
   8. "System"-Funktionen (Analogeingänge, LEDs, Onboard-I/O, Tastatur, Versorgungsspannung, Temperatur, Zeitgeber, Frequenzzähler, usw.).
   9. Funktionen für Datum und Uhrzeit
   10. Funktionen zum Zugriff auf den GPS-Empfänger

11. Funktionen zum Steuern der Trace-Historie
12. Funktionen zum Steuern der virtuellen Tastatur
13. Interaktion zwischen Script und Internet-Protokoll-Stack
    Internet Application Interface ('Socket'-ähnliche API) :
    socket bind listen accept connect send recv close
    Zustandsdiagramm eines Internet-Sockets
    JSON (Javascript Object Notation)
    Testen der Internet / Ethernet - Kommunikation
14. Interaktion zwischen Script und CANopen-Protokoll-Stack
15. Erweiterungen für die Kommunikation per J1939
16. Erweiterungen für die Kommunikation per ISO 15765-2 (aka "ISO-TP")

11. Die Behandlung spezieller Ereignisse (Event-Handling in der Script-Sprache)
    1. Low-Level-Handler für System-Ereignisse (OnKeyDown, etc)
    2. Verarbeitung von Ereignissen, die von UPT-Anzeige-Elementen ausgelöst wurden ('Control Events')
    3. Timer-Events
    4. CAN-Empfangs-Handler
    5. Ereignisse von der virtuellen Tastatur
12. Präprozessor-Direktiven
    1. #pragma
    2. #include
13. Liste mit Schlüsselwörtern (Funktionen, Kommandos, Datentypen)
14. Fehlermeldungen

5. Beispiele :

   PI berechnen, Display-Steuerung, Diagramme, Mittelwertbildung, numerischer Integrator,
   Thermometer (mit NTC), Timer-Events, Button-Events, andere Control-Events,
   Datei-Ein/Ausgabe, Einlesen von INI-Dateien, Internet,Ethernet,TCP/IP, Text-Screen,
   Schleifen, Arrays,
   Error Frames, Operator-Test, Reaktionstest, Quad-Blocks, Trace-Test, CAN-ASCII-Logger, CANopen, J1939, ISO 15765-2, Busruhe erkennen,
   Wert in Editierfeld eingeben, begrenzen/skalieren, und dann per CAN senden,
   VT100/VT52-Emulator, Selbstdefiniertes Popup-Menü, 'App-Selector', Include-Dateien.

6. Bytecode (Information für erfahrene Anwender; nicht unbedingt zum *Einsatz* der Script-Sprache nötig)
   1. Übersetzen des Quelltextes in den Bytecode

2. Der Stack (für Unterprogramme, Zwischenergebnisse, Parameterübergabe, und lokale Variablen)
3. Bytecode-Spezifikation, Mnemonics und Opcodes
7. Letzte Änderungen (nach Datum)

Siehe auch (externe Links, funktionieren nur in HTML, aber leider nicht in der "besser druckbaren PDF-Variante" dieses Dokuments) :

Handbuch zum Programmiertool (ohne Script); Übersicht (des Hilfesystems im Programmiertool),
Feature Matrix (mit Übersicht in welchen Geräten die Script-Sprache verfügbar ist),
Präsentation 'Script-Sprache' (Foliensammlung, Dokument Nr. 85133, Online verfügbar),
Display-Interpreter- Kommandos,
Display-Interpreter- Funktionen .

1. Einleitung

Dieses Dokument beschreibt die Script-Sprache, die in verschiedenen *(aber nicht allen)* programmierbaren Anzeige-Geräten von MKT Systemtechnik zum Einsatz kommt.

Die Script-Sprache dient zum ...

- Verknüpfen von Signalen (d.h. Erzeugen "berechneter" Signale),
- Implementieren von Protokollen (auch für CAN), die in der Firmware nicht direkt implementiert sind, z.B. J1939, ISO 15765-2 ("ISO-TP");
- Verarbeiten von Ereignissen, deren Komplexität die Implementierung als 'Event' (in der Anzeige-Seiten-Definition) nicht zulässt,
- programmgesteuerten Zugriff auf Dateien, z.B. für Event-Logs, automatisch erzeugte Fehlerprotokolle, usw.
- Realisieren einfacher (SPS-ähnlicher) Ablaufsteuerungen, allerdings ohne 'harte' Echtzeitanforderungen
- Implementieren von Algorithmen, die mit den Display-eigenen 'Event'-Definitionen zu aufwändig oder zu unübersichtlich wären.

In vielen Anwendungen für die MKT-View-Familie wird die Script-Sprache *nicht* benötigt, da die normalen Anzeige-Funktionen auch ohne Script-Sprache realisiert werden können. In einigen Fällen reichen die 'normalen' Anzeigefunktionen (inklusive der Display-Event-Definitionen) allerdings nicht aus, oder die Realisierung mit Display-Event-Definitionen wird zu unübersichtlich oder *zu langsam*.
In diesen Fällen hilft der Einsatz der in diesem Dokument beschriebenen Script-Sprache.

Aus Sicht des Entwicklers besteht das Script aus einer Reihe von Programmzeilen, die mit einem im Programmiertool integrierten *Text-Editor* eingegeben werden. Für neue Anwender wird empfohlen, auch das folgende Kapitel zum Script-Editor zu lesen, und danach einige der im Anhang aufgeführten Beispiele im Programmiertool zu laden, um sie im Simulator laufen zu lassen. Mit Hilfe der Sprach-Referenz sollten Sie danach in der Lage sein, erste eigene Scripte (z.B. einfache Ablaufsteuerungen) zu entwickeln und zu testen.
Hier ein einfaches Script, mit dem aus den Anzeige-Variablen 'Spannung' und 'Strom' ständig die Anzeige-Variable 'Leistung' berechnet wird:

```
while(1)
  display.Leistung := display.Spannung * display.Strom;
  wait_ms(50); // 50 ms auf Display-Aktualisierung warten
endwhile;
```

Für anspruchsvollere Programmieraufgaben sollten Sie sich auch mit Funktionen und Prozeduren, Operatoren, Arrays, Strings, und dem Unterschied zwischen Integer- und Fliesskommawerten vertraut machen. Bei der Analyse 'fremder' Scripte hilft Ihnen die Liste von Schlüsselworten.

Die ***Script-Sprache*** hat nichts mit dem wesentlich älteren ***Display-Interpreter*** zu tun. In diesem Dokument wird die *Script-Sprache* beschrieben, nicht der *Display-Interpreter* (mit dem früher die

sogenannten 'globalen und lokalen Ereignisse' *innerhalb der Display-Applikation* definiert wurden). Im Gegensatz zum *Interpreter* wird das Script vor der Ausführung *einmal kompiliert*. Dabei wird der Quelltext in einen proprietären Bytecode umgewandelt. Das aus Bytecode bestehende Programm kann wesentlich schneller abgearbeitet werden, als es mit einem Interpreter möglich wäre.

Der Script-*Quelltext* wird wird i.A. mit dem Script-Editor eingegeben. Zu Testzwecken kann er bereits im Simulator (d.h. im Programmiertool) in den Bytecode übersetzt (kompiliert) werden. Auf dem Zielsystem wird ein 'abgespeckter' Bytecode (ohne Debug-Möglichkeiten) verwendet; darum ist der Script-Compiler nicht nur im Programmiertool, sondern auch in der Gerätefirmware vorhanden. Der Bytecode wird nach dem Übersetzen im Simulator oder im Zielsystem als "Programm" abgearbeitet.

Die *erweiterten* Script-Funktionen müssen im programmierbaren Gerät freigeschaltet werden, bevor sie im Zielgerät genutzt werden können. Im Programmiertool (d.h. im Simulator) ist dagegen keine Freischaltung nötig.

1 1.1 Prinzip

Das Script-Programm (Quelltext) wird sofort nach dem Einschalten des programmierbaren Terminals, bzw. nach dem Starten des Simulators im Programmiertool kompiliert.
Die Ausführung des *kompilierten* Programms beginnt dann in der ersten Zeile. Im Normalfall werden in den ersten Zeilen Initialisierungen durchgeführt, wie z.B. das Setzen von Variablen auf die Startwerte. Der Abschluss der Initialisierung *kann* im Script mit dem Kommando init_done markiert werden. Dadurch werden u.A. auch die Event-Handler aktiv, und das Script kann z.B. aus programmierbaren Anzeigeseiten aufgerufen werden.

Nach der Initialisierung warten die meisten Scripte in einer Endlosschleife, bis 'Etwas' passiert (z.B. der Empfang bestimmter CAN-Telegramme, eine bestimmte Aktion des Bedieners, etc). Im Normalfall enthält diese Hauptschleife mindestens einen Aufruf einer 'Warte-Funktion' (wait_ms). Dadurch kann das Script die CPU 'freiwillig' für die Aktualisierung der Anzeige abgeben.
Die Ereignis-Verarbeitung (event handling) ist ein wesentlicher Bestandteil der Script-Sprache, sie ist nicht mit den 'Events' im alten Display-Interpreter zu verwechseln. Beide Möglichkeiten ("globale und lokale Events" in den programmierbaren Anzeigeseiten, und Event-Handler in der Script-Sprache) können miteinander kombiniert werden. Dank der Konstrukte zur Ablaufsteuerung (if..then..else, for..to..next, select..case) können komplexere Steuerungen in der Script-Sprache wesentlich *übersichtlicher* implementiert werden als in den Event-Definitionen des alten Display-Interpreters.

2 1.2 Freischaltbare Funktionen für die Script-Sprache

In der Script-Sprache sind nur noch die *Standardfunktionen*, die als "Hobby-Projekt" in der Freizeit des Autors entstanden, ohne Einschränkungen für jedermann nutzbar. **Die *erweiterten* Script-Funktionen, die während der Arbeitszeit des Autors *bei* MKT Systemtechnik, *für* MKT Systemtechnik entstanden sind, müssen vor der Verwendung erst freigeschaltet werden** (um den Entwicklungsaufwand für MKT abzudecken). Ohne Freischaltung sind die folgenden *erweiterten* Funktionen nur im Simulator (d.h. im Programmiertool), aber -von wenigen Ausnahmen abgesehen- nicht in der Geräte-Firmware nutzbar:

- Empfang und Senden von CAN-Telegrammen per Script-Sprache (CAN.xyz)
- Funktionen zum Zugriff auf Dateien, und Geräte auf die "wie auf eine Datei" zugegriffen werden kann (file.xyz, auch für die serielle Schnittstelle).
- Funktionen zur Kommunikation per TCP/IP oder UDP
- Frequenz- und Ereigniszähler für die digitalen onboard-Eingänge
- weitere hardware-spezifische Sonderfunktionen (geplant)
- Spezialfunktionen für die digitale Signalverarbeitung, z.B. Fouriertransformation

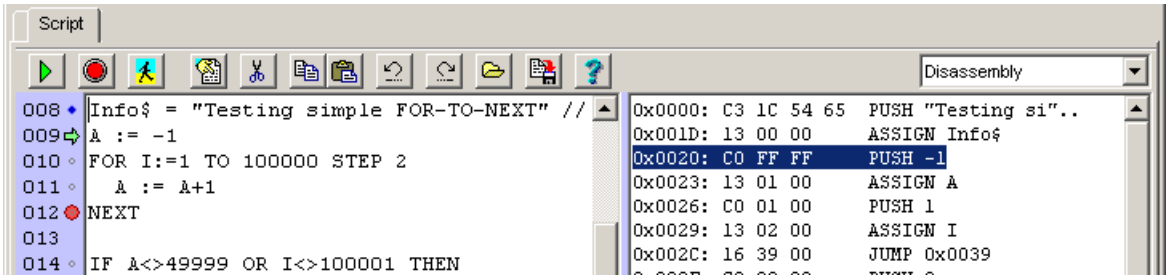
Solange die oben genannten Sonderfunktionen nicht freigeschaltet sind, "funktionieren" sie einfach nicht. Der Compiler meldet aber keine Fehler, und das Script bricht beim Aufruf dieser Funktionen nicht (mit einem Fehler) ab.
Versucht *das Script* z.B., ein CAN-Telegramm zu senden, wird das Telegramm nicht gesendet (das CAN-Sende-Kommando wird stattdessem ignoriert). Der Versuch, eine Datei oder die serielle Schnittstelle zu öffnen, liefert ein ungültiges Datei-Handle.

Die oben aufgelisteten Sonderfunktionen müssen im Bedarfsfall für JEDES GERÄT, auf dem diese Funktionen verwendet werden sollen, individuell freigeschaltet werden. Weitere Details, wie Sie den Freischaltcode für ein bestimmtes Gerät vom Hersteller anfordern können, finden Sie in diesem Dokument.

Wir bitten vorab um Entschuldigung, falls Ihnen die Freischaltung Umstände bereitet. Der Hersteller, MKT Systemtechnik, kann die neuen Funktionen nicht kostenlos zur Verfügung stellen. Andererseits bezahlen durch diese Art der 'gerätegebundenen' Freischaltung nur Anwender, die diese Funktionen benötigen, einen kleinen Anteil der Entwicklungskosten, und kein(e) Anwender(in) bezahlt für Funktionen die er/sie nicht benötigt.

2. Script-Editor und Debugger

Der Script-*Editor* befindet sich auf der Registerkarte 'Script' im Programmiertool. Ist diese Registerkarte nicht sichtbar, unterstützt das Zielsystem keine Scripte, oder das Programmiertool ist zu alt.



Registerkarte 'Script' mit Quelltext (links) und Disassembler (rechts)

Im Script-*Editor* können per Maus Informationen zu einem bestimmten Schlüsselwort, oder (je nach Programmzustand) der Inhalt einer Variablen abgerufen werden. Bewegen Sie dazu den Mauspfeil über die entsprechende Stelle im Quelltext, und warten circa eine Sekunde *ohne* Mausklick. Erkennt der *Editor* das Schlüsselwort 'unter der Maus', oder den Namen einer *zur Zeit* ***gültigen*** Variablen, wird ein entsprechender Hinweis in der Nähe der Mauspfeils angezeigt (bei Variablen: mit Namen, Datentyp, und aktuellem Wert). Der Hinweis verschwindet, sobald die Maus wieder bewegt wird.

Per Default werden Schlüsselwörter und ähnliche "besondere" Sprachkonstrukte im Editor farblich hervorgehoben ("syntax higlighting"). Schlüsselwörter werden in fetter schwarzer Schrift markiert, Kommentare in Blau, symbolische Konstanten in Grün, usw. Beachten Sie, daß (im Gegensatz zu anderen Editoren) die farbliche Markierung nicht sofort bei der Eingabe aktualisiert wird, sondern i.A. erst beim Übersetzen. (Erst beim Kompilieren wird die Symboltabelle aufgebaut, und nur anhand der Symboltabelle "erkennt" der Editor, worum es sich im Quelltext handelt).
Um die Syntax-Markierung zu aktualisieren, klicken Sie daher auf den Button 'STOP / RESET / RECOMPILE' (roter Kreis, ähnlich "Not-Aus").
Bei Nichtgefallen kann die Syntax-Markierung abgeschaltet werden, näheres dazu im folgenden Kapitel. Der Editor verwendet dann normale schwarze Buchstaben auf weißem Hintergrund.

Die maximale Größe des *Quelltextes* ist geräteabhängig; sie beträgt mindestens 32 kByte, bei vielen Geräten aber bis zu 256 kByte. Die maximale Größe des *compilierten Bytecodes* beträgt zwischen 32 und 128 kByte (ebenfalls geräteabhängig). Per se 'kennt' der Script-Editor die Eigenschaften des zu programmierenden Gerätes nicht, es sei denn, *Sie* stellen im Programmiertool die zu 'Ihrer' Hardware passenden Speichergrößen ein (auf der Registerkarte 'General Settings' unter "Max. size of script sourcecode in kByte").
Die aktuell vom Script belegten Speichergrößen (Source- und Bytecode) werden nach dem Übersetzen in der Statuszeile unterhalb des Script-Editors angezeigt, z.B.:

Compilation ok, 1234 of 65536 bytecode locations used, 6 kByte sourcecode.

Nach dem Compilieren kann das Script (auf dem PC) mit dem Debugger getestet werden. Dafür stehen Tools wie z.B. Breakpoints, Single-Step-Betrieb, ein Disassembler, die Trace-Historie, und die Anzeige der Symboltabelle mit Variablenwerten zur Verfügung.

Zum Entwickeln des Scriptes stehen im Script-Editor einige Hilfsmittel zur Verfügung, z.B.:

- die Toolbar (Leiste mit Schaltflächen oberhalb des Quelltext-Editors)
- das Kontext-Menü des Quelltext-Editors (Rechtsklick *in den Quelltext*)
- das Kontext-Menü der 'Sidebar' (Rechtsklick *in die Zeilennummern*)

Einige Einstellungen des Script-Editors (z.B. die Tabulatorgröße) können im Hauptmenü des Programmiertools unter 'Optionen' .. 'Script-Editor' eingestellt werden.
Details zu den Hilfsmitteln für die Script-Entwicklung folgen in den nächsten Kapiteln.

1 2.1 Die Script-Editor-Toolbar

In der Toolbar des Script-Editors finden Sie neben den üblichen Symbolen zum Ausschneiden, Kopieren, und Einfügen von Text die folgenden Funktionen ("Buttons"):


RUN

Startet das Script, oder setzt eine vorher gestoppte Ausführung fort. Wurde das Script seit der letzten Änderung im Editor nicht neu compiliert, wird es vor dem Start automatisch neu übersetzt.
Die Ausführung eins Scripts kann automatisch stoppen, wenn das Script 'auf einen Breakpoint läuft', oder beim Auftreten eines Fehlers.


STOP / RESET / RECOMPILE

Stoppt die Ausführung des Scripts, oder setzt, wenn das Script bereits gestoppt war, das Script auf den Startzustand zurück. Wurde das Script im Editor geändert, wird es automatisch neu compiliert.


SINGLE STEP (F7)

Führt das nächste Kommando im Einzelschritt-Betrieb aus. Das nächste auszuführende Kommando wird am linken Rand des Editorfensters durch einen grünen Pfeil markiert. Der Einzelschritt-Betrieb wird z.B. verwendet, um das Verhalten an kritischen Stellen zu analysieren, nachdem das Script per Breakpoint gestoppt wurde.


STEP OVER (F8)

Führt ebenfalls das nächste Kommando (eine Zeile) im Einzelschritt-Betrieb aus. Im Gegensatz zum normalen Einzelschritt (F7, aka 'Step In') führt 'Step Over' allerdings einen kompletten Funktions- oder Prozeduraufruf aus, d.h. es werden ggf. viele Zeilen 'übersprungen':
Steht in der mit dem grünen Pfeil markierten Zeile der Aufruf eines Unterprogramms (Prozedur oder Funktion in der Script-Sprache), dann wird die gesamte Prozedur/Funktion, inklusive aller von dort eventuell aufgerufenen Unterprogramme abgearbeitet bevor das Script für den nächsten Einzelschritt wieder gestoppt wird.


STEP OUT

Führt den Rest des aktuellen Unterprogramms aus, bis zum Rücksprung zum Aufrufer. Wird üblicherweise mit der 'Single Step' bzw 'Step-In'-Funktion verwendet.


View "CPU" (debugger code window)

Öffnet die Bytecode-Ansicht (Disassembler) auf der rechten Seite der Registerkarte 'Script'. Dient i.A. nur zum 'Hardcore-debugging', z.B. wenn Probleme mit dem Stack oder Rechenfehler auftreten.
Per Single-Step-Betrieb in der Bytecode-Ansicht werden auch alle Zwischenergebnisse sichtbar, die beim Abarbeiten einer 'kompletten Quelltext-Zeile' verborgen bleiben.


Script-Editor-Menü

Öffnet ein Popup-Menü mit seltener verwendeten Funktionen, z.B. Debugger-Optionen (z.B. Zeilen-Marker, Breakpoints, Trace-Historie, Watch-Liste,.. - Details dazu später).
In diesem Menü kann auch das farbige Hervorheben der Syntax (im Editor) abgeschaltet werden, und die für den Editor und Debugger verwendeten Zeichensätze ausgewählt werden.
Tipp: Sofern auf dem PC installiert, liefert 'DejaVu Sans Mono' oft einen besser lesbar Anzeige als 'Courier New'.

Text suchen

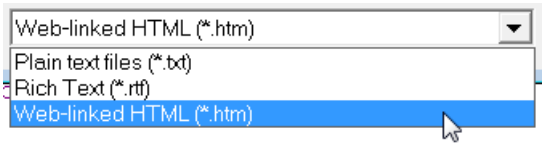
Sucht nach einem bestimmten Text (String) im Editor. Mit dem Button 'Find Next' ("Weitersuchen") kann zur nächsten Textstelle gesprungen werden.
Siehe auch: 'Globale Suche' auf allen Anzeigeseiten und in allen Script-Quelltext-Zeilen, per Rechtsklick auf das gesuchte Wort im Quelltext-Editor in dessem Kontext-Menü.

Import eines Scripts aus Datei

Importiert einen Script-Quelltext aus einer Text-Datei. Alle Breakpoints, und der alte Text im Editor werden dabei gelöscht.

Export eines Scripts als Datei

Exportiert den Quelltext als Textdatei. Da der Script-Quelltext automatisch als Teil des Display-Programms (in der *.upt bzw *.cvt-Datei) gespeichert wird, wird diese Funktion nur gelegentlich benötigt, z.B. für die Übernahme eines Scripts aus Applikation "A" in Applikation "B" (als Text-Datei), oder zu Dokumentationszwecken (mit Syntax-Hervorhebung als RTF- oder HTML-Datei).
Tipp: Beim Export als *.HTM (HTML) können 'anklickbare' Links erzeugt werden. Von dieser Möglichkeit wurde beim Erstellen *dieser* Dokumentation häufig Gebrauch gemacht, z.B. im Kapitel 'Beispiele'.



Dateiformate zum Exportieren eines Scripts

Am linken Rand des Script-Editors werden die Zeilennummern,
und spezielle Indikatoren für Zeilen mit 'ausführbarem Code' angezeigt.
Bedeutung der Indikatoren am linken Rand des Editorfensters ('Sidebar'):

(grüner Pfeil nach rechts)
Aktueller Code-Pointer (nicht ganz korrekt auch "Programmzähler" genannt).
Zeigt die Zeile, in der der nächste auszuführende Befehl steht. Details im Kapitel Debugging (Fehlersuche).

(kleiner, hohler, grauer Kreis)
In dieser Zeile steht prinzipiell ausführbarer Code, das Programm "war aber noch nicht hier", seitdem das Script gestartet wurde.
Während des Debuggens (auch im laufenden Betrieb im Simulator) kann in dieser Zeile ein Breakpoint gesetzt werden, d.h. das Programm kann beim Versuch, diese Zeile

"abzuarbeiten", automatisch gestoppt werden.
Dazu reicht ein einzelner Klick mit der linken Maustaste auf dieses Symbol.

(blauer, ausgefüllter Kreis)
"Das Programm war seit dem Start *mindestens einmal* in dieser Zeile" (englisch: "been-to"-Marker).
Ähnlich wie beim 'grauen Kreis' kann auch in dieser Zeile ein Breakpoint gesetzt, und per Mausklick wieder entfernt werden.
Per 'Reset' (roter "Not-Aus"-Knopf in der Toolbar) können alle 'war-schon-mal-hier'-Marken gelöscht werden.

(rote Kreisscheibe)
In dieser Zeile wurde ein Breakpoint gesetzt, das Programm war aber (seit dem Start) "noch nicht hier".
Wenn das laufende Programm diesen Punkt (Breakpoint) erreicht, wird es automatisch gestoppt.
Danach können z.B. Variablen inspiziert werden (weil sich deren Wert bei gestopptem Programm nicht mehr ändert).

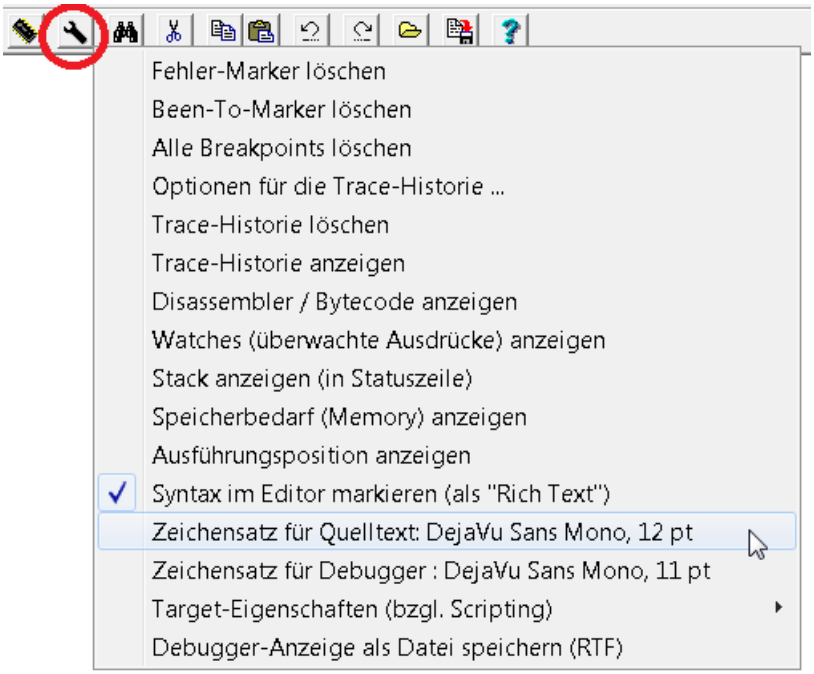
(rote Kreisscheibe mit blauem Mittelpunkt)
In dieser Zeile wurde ein Breakpoint gesetzt, und das Programm hat diese Zeile seit dem letzten "Reset" mindestens einmal abgearbeitet.

(gelbes Dreieck mit schwarzem Rahmen)
Warnung oder Fehler in dieser Zeile.
Beim Übersetzen dieser Zeile trat ein Fehler auf, oder das erfolgreich compilierte Programm blieb wegen eines Laufzeitfehlers in dieser Zeile (oder in der Nähe dieser Zeile) stehen.
Details zum Fehler (Fehlermeldung) stehen in der Statuszeile, oder (beim "Anfahren" des Symbols per Mauszeiger) direkt daneben. Zusätzliche Informationen (Ursache) zu bestimmten Fehlern und Warnungen werden auf der Registerkarte Fehler & Meldungen angezeigt.

Beim Editieren des Quelltextes verschwinden die Code-Indikator-Punkte, bis zum erneuten Compilieren.

Per Rechtsklick in die 'Sidebar' (mit den oben gezeigten Symbolen) wird das Kontext-Menü der 'Sidebar' geöffnet. Darin sind Funktionen wie z.B. 'aktuelle Ausführungsposition anzeigen', 'Zeige Fehler in Zeile xyz', 'Breakpoint umschalten' enthalten. Details zum Kontext-Menü der 'Sidebar' folgen im nächsten Kapitel.

Per Klick auf den 'Menü'-Button in der Toolbar des Script-Editors wird ein Menü geöffnet, welches hauptsächlich in Zusammenhang mit dem Debuggen verwendet wird:



Screenshot vom 'Debug-Menü' auf der Registerkarte des Script-Editors

2 2.2 Hotkeys und Kontext-Menüs des Script-Editors

STRG-C : Copy
: Kopiert den selektierten Text in die Windows-Zwischenablage

STRG-V : Paste
: Fügt den Text aus der Zwischenablage an der aktuellen Cursorposition ein. Ist im Editor ein Block markiert, so wird dieser Block mit dem Text aus der Zwischenablage *überschrieben*.

STRG-F : Find
: Öffnet den bekannten Dialog zur Suche nach Text

STRG-Z : Undo
: Den letzten Bearbeitungsschritt wieder rückgängig machen

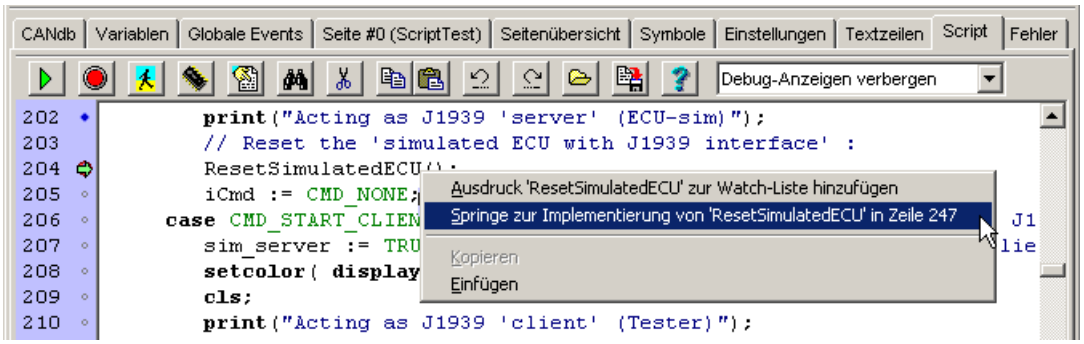
SHIFT-CTRL-Z : Redo
: Das letzte Rückgängig-machen wiederrufen ("undo undo")

F1
: Erweiterte Hilfe zum Schlüsselwort im Editor-Quelltext.
Zeigen Sie mit der Maus auf das Schlüsselwort, und warten ab bis eine 'Info-Blase' zum Schlüsselwort angezeigt wird.
Reicht Ihnen diese (knappe) Information nicht aus, drücken Sie F1 um im HTML-Browser weitere Hilfe zu erhalten.
Dies funktioniert leider nur mit 'guten' HTML-Browsern, die per Kommandozeilen-Parameter an einen bestimmte Textmarke springen können, z.B. Firefox und Iron/Chrome.

F7
: Einzelschritt (für den Debugger; Details im nächsten Kapitel)

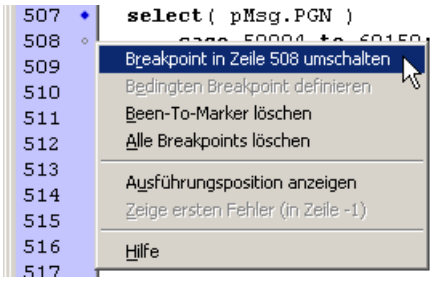
Darüberhinaus stehen im Editor alle Tastenfunktionen zur Verfügung, die Microsoft's Rich-Text-Editor 'von Haus aus' mitbringt.

Per Klick mit der rechten Maustaste (in den Script-Editor) kann dessen Kontext-Menü geöffnet werden. Für bestimmte Sonderfunktionen wird dabei auch das 'Wort' im Quelltext unter dem Mauszeiger ausgewertet, z.B. um den Namen einer globalen Variablen als 'überwachten Ausdruck' in die Watch-Liste zu übernehmen, oder Hilfe zum 'Wort' unter dem Mauspfeil zu erhalten :



Screenshot des Script-Editors mit Kontextmenü, nach Rechtsklick auf ein bestimmtes Wort

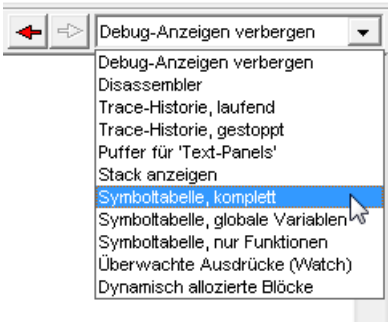
Per Rechtsklick in die 'Sidebar' (mit den Zeilennummern und Code-Ausführungs-Indikatoren) wird das Kontext-Menü der 'Sidebar' geöffnet:



Screenshot des Kontext-Menüs der 'Sidebar' des Script-Editors

3 2.3 Debugging

Die wenigsten Programme (Scripte) werden sofort fehlerfrei laufen. Um Programmfehler zu finden, kann das Script auch im Programmiertool übersetzt und ausgeführt werden. Nur dann stehen die in diesem Kapitel beschriebenen Möglichkeiten zur Fehlersuche (Debugging) zur Verfügung.
Debugging auf dem 'echten' Gerät ist nur eingeschränkt möglich (per Ethernet und Web-Browser).
Während einer Debugger-Sitzung wird der Bildschirm *möglicherweise* in zwei Teile gesplittet. In der linken Hälfte steht dann der Quelltext, in der rechten Hälfte z.B. der disassemblierte Bytecode, Symbole, oder einzelne Variablen. Die Position des bläulich gefärbten 'Splitters' (zwischen Quelltext-Editor und Debugger-Anzeige) kann wie üblich per Maus verschoben werden.
Die 'Art' der Debugger-Anzeige kann per Menü, per Hyperlink (z.B. in der Symboltabelle), oder mit der Combo-Box in der oberen rechten Ecke der Toolbar gewählt werden:

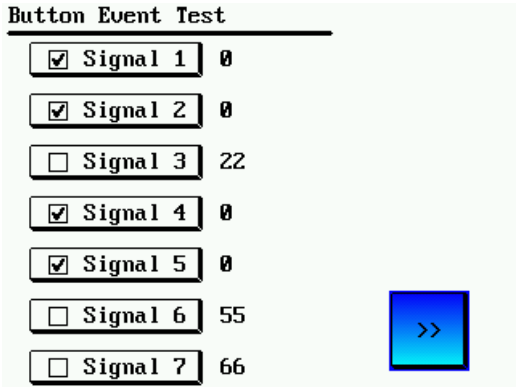


Auswahl der Debugger-Anzeige per Combo-Box,
mit Buttons 'Rückwärts' / 'Vorwärts' wie im Web-Browser.

- Breakpoints :
  Ein Breakpoint ("Anhaltepunkt") kann die Abarbeitung des Scripts unterbrechen, wenn der Programmzeiger die entsprechende Stelle im Quelltext erreicht. Um einen Breakpoint zu setzen oder zu löschen klicken Sie auf einen der Marker für ausführbaren Code am linken Rand des Quelltext-Editors.
  Mit der linken Maustaste wird ein Breakpoint 'getoggelt' (ein/aus), die rechte Maustaste öffnet ein Kontext-Menü mit weiteren Optionen.
- Single Step :
  Ermöglicht das Abarbeiten des Scripts im Einzelschritt-Betrieb, d.h. eine Quelltext-Zeile pro Schritt, ein Unterprogramm pro Schritt, oder eine Bytecode-Anweisung pro Schritt (im Disassembler-Fenster). Einzelschritte ('Step In/Over/Out') werden mit den im vorhergehenden Kapitel vorgestellten Single-Step-Buttons ausgeführt.
- Inspizieren von Variablen :
  Bewegen Sie den Mauspfeil über den Namen einer globalen (*) Variablen im Quelltext. Das Programmiertool durchsucht daraufhin die vom Compiler erzeugte Symboltabelle, und zeigt (wenn der Name dort gefunden wurde) alle verfügbaren Informationen an (Name, Datentyp, aktueller Wert). Bei *globalen* Variablen funktioniert dies auch, während das Script läuft.
  (*) : Das Inspizieren von *lokalen* Variablen, d.h. Variablen die nur innerhalb einer Funktion (oder Prozedur) gültig sind, ist nur möglich wenn der Debugger in der entsprechenden Funktion gestoppt wurde. Grund: Lokale Variablen existieren nur innerhalb des Stack Frames der aktuellen Funktion - sie haben im Gegensatz zu globalen Variablen keine 'feste Addresse' in der Symboltabelle. Alle anderen lokalen Variablen (in anderen Funktionen und

Prozeduren) sind für den Debugger nicht sichtbar, und werden bestenfalls als 'Other Object', aber ohne Wert und Datentyp, bei der Inspektion angezeigt.

Bei Geräten mit einem ausreichend großen Display können Script-Variablen wie folgt auch im System-Menu inspiziert werden: Scrollen Sie im System-Menü (Hauptebene) bis zum Untermenü 'Script: ...', öffnen dieses per ENTER oder Touchscreen, und wählen dort den Eintrag 'Variables ..' (Variablen).
In der Liste (Übersicht) werden nur der Name (links) und der aktuelle Wert (rechts) angezeigt. Details zur momentan selektierten Variable erhalten Sie in einem weiteren Untermenü, welches durch erneutes Drücken der ENTER-Taste (bzw. Touchscreen) *am programmierbaren Gerät* geöffnet werden kann.



Screenshot aus dem 'Button-Event-Test' (programs/script_demos/ButtonEventDemo.cvt)

Mit der Funktion 'Start watching this item' kann die selektierte Variable zu Testzwecken bis zum Abschalten des Gerätes in das Display (am unteren Rand) eingeblendet werden.

- Anzeige der Symboltabelle :
  Die Symboltabelle wird vom Script-Compiler erzeugt. Im Debugger kann die *gesamte* Tabelle, oder nur die darin enthaltenen globalen Variablen, oder die Namen von anwenderdefinierten Funktionen oder Prozeduren am rechten Rand des Hauptfensters angezeigt werden. Namen, Quelltext-Zeilen-Nummern, und Adressen werden als anklickbarer 'Hyperlink' angezeigt, um die Navigation im Script-Quelltext zu erleichtern.

- Anzeige des Stacks :
  Während single-step-Debugging werden in der Statuszeile des Script-Editors die oberen Elemente des Stacks angezeigt. Diese Funktion wurde hauptsächlich während der Implementierung der Script-Sprache verwendet. Sie kann allerdings auch beim 'normalen' Debuggen hilfreich sein, z.B. wenn in einer Script-Anwendung viele Unterprogramm-Aufrufe erfolgen (oder rekursive Aufrufe der gleichen Funktion). Mit Hilfe der Stack-Anzeige können fortgeschrittene Anwender erkennen, "wie" das Programm an die aktuelle Stelle kam, welche Funktionsparameter bei jedem Aufruf übergeben wurden, und welche Zwischenergebnisse beim Abarbeiten der RPN entstehen. Details zu diesem nicht-trivialen Thema finden Sie im englischen Original dieses Dokuments.

  Der Stack kann sowohl in der Statuszeile als auch *mehrzeilig* in einer Textliste angezeigt werden.
  Beispiel für die kurze (einzeilige) Anzeige der 'Spitze' des Stacks in der Statuszeile:

```
Stack[6] : tCANmsg 0 return_to_822 1 65230 0
```

Details zur Anzeige des Stacks (auch als *mehrzeilige* Liste) finden Sie hier.

- Anzeige der aktuellen Speicherauslastung (Memory Usage Display) :
  Nach einem Testlauf des Scripts im Simulator sollten Sie gelegentlich den Speicherverbrauch Ihrer Applikation untersuchen, speziell wenn Sie eine größere Anzahl von Strings (z.B. in Arrays) verwenden, oder beim intensiven Einsatz von Unterprogrammen (Funktionen und Prozeduren). Im Gegensatz zum PC (mit seinem nahezu unerschöpflichen Hauptspeicher) steht im Zielsystem nämlich nur ein begrenzter Speicher für Script-Variablen und den Stack zur Verfügung.
  Zum Anzeigen der Speicherauslastung (RAM) klicken Sie in der Toolbar des Script-Editors auf den Menü-Button, und wählen die Funktion ***Show Memory Usage*** .
  In der Statuszeile wird dann (ständig aktualisiert) der Speicherbedarf des Scripts angezeigt, z.B.:

```
Memory Usage : 5 of 256 stack items, peak=33; 7 of 200 variables; 62
```

  Dies bedeutet:

  Momentan werden 5 von maximal 256 Elementen auf dem (RPN-)Stack verwendet;
  die maximale Stack-Auslastung (seit dem Start des Scripts) betrug 33 von maximal 256 Elementen;
  7 von maximal 200 globalen Variablen werden benutzt;
  62 von maximal 1000 Datenblöcken (mit jeweils maximal 64 Byte pro Block) werden momentan verwendet;
  die maximale Anzahl gleichzeitig verwendeter Datenblöcke betrug 85 von 1000 .

  Dies ist eine typisches, unkritisches Beispiel weil alle gemessenen 'Spitzenwerte' deutlich unter den Maximalgrößen liegen.
  Nähert sich einer der angezeigten Spitzenwerte (Stack, Anzahl globaler Variablen, Anzahl Datenblöcke) dem Maximum, dann ...
  - reduzieren Sie den Stack-Bedarf, indem Sie weniger lokale Variablen, oder weniger Rekursion bei Unterprogramm-Aufrufen verwenden;
  - verringern Sie den Bedarf an Datenblöcken durch Verkleinern von Arrays;
  - verringern Sie den Bedarf an Datenblöcken indem Sie weniger Strings verwenden, oder indem Sie *leere* Strings an String-Variablen (oder Elementen in String-Arrays) zuweisen, wenn Sie deren Inhalt nicht mehr benötigen, z.B.:

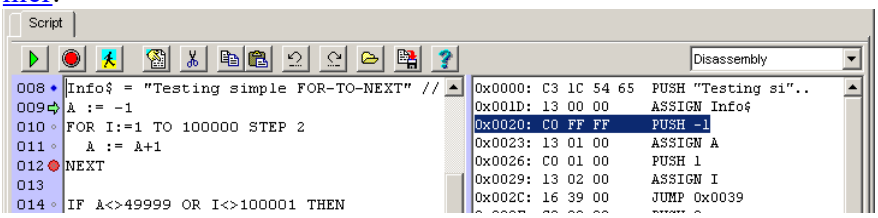
```
Info := ""; // clear this string to release its memory block
```

  Hinweis: Die Speicheranzeige (*Memory Usage*) auf der Registerkarte 'Script' zeigt nur den Speicherbedarf des Scripts an. Dies hat nichts mit dem Speicher zu tun, der für die UPT-*Anzeigefunktionalität* (z.B. für Icons, Anzeigeseiten, usw.) benötigt wird ! Die Script-Sprache verwendet einen eigenen Speicherpool, um das Funktionieren der Anzeige zu garantieren, selbst wenn dem Script (z.B. wegen eines Programmierfehlers im Script) die Resourcen ausgeben, und die Script-Bearbeitung wegen eines Fehlers stoppt.
  Alternativ kann die maximale Speicherauslastung auch im Script selbst überwacht werden, um ggf. frühzeitig auf zu knappe Resourcen zu reagieren. Dazu dient die Funktion system.resources .
  Zur Kontrolle des im Zielsystem nach Abzug des Speichers für Anzeigeseiten, Icons

(Bitmap-Grafiken), Display-Variablen, CAN-Datenbanken und Script verbleibenden ***freien Flash-Speichers*** verwenden Sie die Funktion 'Flash-Speicher-Bedarf im Target' im Hauptmenü des Programmiertools unter 'Ansicht'.
Details zur aktuellen Verwendung des dynamisch allozierten Speichers erhalten Sie (im Debugger) wie hier beschrieben.

- Anzeige der Symboltabelle :
  In der Symboltabelle sind die Namen aller vom Compiler erzeugten Script-Variablen, selbstdefinierte Datentypen, selbstdefinierte Konstanten, und (z.T.) auch *deren aktuelle Werte* angezeigt.
  In der Symboltabelle angezeigte Namen, Zeilennummern, und Code-Adressen können wie ein 'Hyperlink' im Browser angeklickt werden, um im Quelltext oder in verschiedenen Debugger-Ansichten zu navigieren.

- Disassembler (Anzeige des disassemblierten Bytecodes) :
  Nur für fortgeschrittene (und neugierige) Anwender ... diese Funktion diente hauptsächlich als Hilfe bei der Implementierung neuer Funktionen (Opcodes) im Script. Weitere Details hier.



- Trace History (Anzeige der Trace-Historie) :
  In der Trace-Historie werden die letzten 255 Ereignisse mit den folgenden Typen angezeigt:
  - Vom Terminal *gesendete* CAN-Telegramme
  - Vom Terminal *empfangene* CAN-Telegramme
  - Fehlermeldungen und Warnungen (aus der Terminal-Firmware bzw. Simulator)
  - Vom Script mit dem Kommando trace.print in die Historie 'gedruckte' Zeilen
  - Debug-Meldungen von den Internet-Funktionen im Script (optional)

  Details zur Trace-Historie folgen in Kapitel 2.3.3.

- Überwachte Ausdrücke (Watch-Liste) :
  Zeigt eine benutzerdefinierte Auswahl von 'Ausdrücken' (z.Z. beschränkt auf globale Script-Variablen) auf dem Debugger-Panel in der rechten Hälfte der Registerkarte 'Script' dar. Dies können 'einfache Variablen-Werte', aber auch komplette Arrays und selbstdefinierte Daten (Strukturen). Details zur Watch-Liste folgen in Kapitel 2.3.6.

Hinweis:

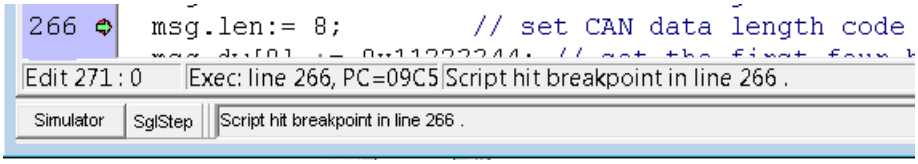
Für die Programmentwicklung empfiehlt es sich, ***zwei Monitore*** an den PC anzuschließen. Verschieben Sie das Hauptfenster des Programmiertools auf den einen, und das Fenster des LCD-Simulators auf den anderen Monitor. Sind Sie nicht in der glücklichen Lage, an Ihren PC einen zweiten Monitor anschließen zu können, machen Sie das LCD-Simulator-Fenster nur so groß wie unbedingt nötig, und wählen Sie die Option 'stay on top' für das LCD-Simulator-Fenster. Danach können Sie den LCD-Simulator in die obere rechte Ecke des

Hauptfensters im UPT-Programmiertool verschieben, so daß keine wesentlichen Bestandteile des Script-Editors verdeckt werden. Der LCD-Simulator bleibt dann sichtbar, selbst wenn das Hauptfenster den ganzen Bildschirm füllt und der Simulator *nicht* den Fokus hat.

Siehe auch: Debugging per Embedded Web Server (und HTML-Browser)

4 2.3.1 Breakpoints und Single-Step-Betrieb

Ein **Breakpoint** ("Anhaltepunkt") kann die Abarbeitung des Scripts unterbrechen, wenn der Programmzeiger die entsprechende Stelle im Quelltext erreicht. Um einen Breakpoint zu setzen oder zu löschen klicken Sie auf einen der Marker für ausführbaren Code am linken Rand des Quelltext-Editors.
Wenn das Script (im Simulator) wegen eines Breakpoints oder Programmfehlers stoppt, wird in der Statuszeile u.A. die Zeilennummer angezeigt, in der das Script gestoppt wurde. Beispiel:



Statuszeile des Script-Editors nach Stoppen durch Breakpoint

Ein Doppelklick in die Statuszeile schaltet dann in die entsprechende Quelltext-Zeile um, d.h. scrollt den Text im Editor so dass die zuletzt abgearbeitete Zeile sichtbar wird.

Single-Step-Betrieb ermöglicht das Abarbeiten des Scripts in einzelnen Schritten, d.h. eine Quelltext-Zeile pro Schritt, oder eine Bytecode-Anweisung pro Schritt (im Disassembler-Fenster).
Ein Einzelschritt wird mit dem im vorhergehenden Kapitel vorgestellten Single-Step-Button ausgeführt.
Single-Step ist z.Z. nur auf dem PC (im Programmiertool) möglich, da bei den meisten Geräten bislang keine Schnittstelle mit ausreichender Bandbreite und Netzwerkfähigkeit existierte.

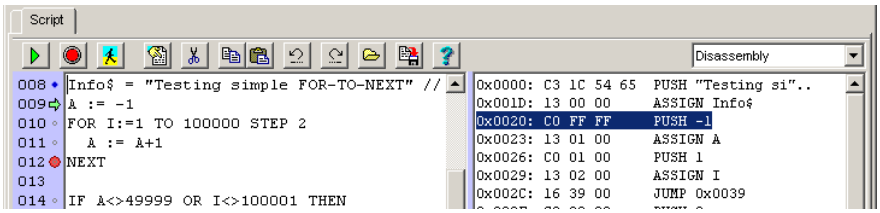
Tipp:

In manchen Geräten (z.B. MKT-View II / III, mit Ethernet-Schnittstelle und Web-Server) ist ein Script-Debugger integriert, der 'ganz ohne' UPT-Programmiertool funktioniert.
Auch dieser sogenannte 'Remote-Debugger' (fernbedienter Debugger) ermöglicht es, *während des normalen Betriebs* Haltepunkte zu setzen und das Script im Single-Step-Modus auszuführen.
Geben Sie dazu im Web-Browser den Host-Namen des Gerätes, oder dessen numerische IP-Adresse, gefolgt von **/script/d** ein (d=Option "Quelltext-Debugger").
Bei einigen (dummen) Browsern muss in der Adressleiste noch das Transportprotokoll vorangestellt werden, z.B. http://upt/script/d .
Weitere Informationen zum Debuggen per Web-Browser finden Sie hier (externer Link).

5 2.3.2 Disassembler (Code-Anzeige)

Nur für fortgeschrittene (und neugierige) Anwender !

Der Disassembler zur Anzeige des disassemblierten Bytecodes diente hauptsächlich als Hilfe bei der Implementierung neuer Funktionen (Opcodes) im Script. Die Disassembler-Anzeige kann per Button in der Editor-Toolbar geöffnet werden ('Chip'-Symbol). Der disassemblierte Bytecode wird in einem Textfenster in der rechten Hälfte des Hauptfensters aufgelistet, mit einer Zeile pro Bytecode-Anweisung. Solange diese Anzeige aktiv ist, kann das Programm im Single-Step-Modus mit *einer Bytecode-Anweisung pro Schritt* abgearbeitet werden (im normalen Single-Step-Betrieb wird dagegen *eine Quelltext-Zeile pro Schritt* abgearbeitet).
Dadurch kann z.B. beobachtet werden, wie numerische Ausdrücke ("Formeln") als RPN (Reverse Polish Notation) abgearbeitet werden, welche Zwischenergebnisse dabei auf dem RPN-Stack erscheinen, und wie beim Parameter an Funktionen und Prozeduren per Stack übergeben werden.



Screenshot von Quelltext-Editor (links) mit Disassembler-Anzeige (rechts)

Direkt nachdem das Programm bei der Ausführung per Breakpoint gestoppt wurde, oder wenn bereits einige Einzelschritte *im Disassembler-Fenster* ausgeführt wurden, bewegt sich die aktuelle Ausführungsposition (grüner Pfeil) *im Quelltext-Fenster* nicht, bzw. nicht immer. Ursache ist das jede Quelltextzeile i.A. aus *mehreren* Bytecode-Anweisungen besteht. Zum Ausführen einer einzigen Quelltext-Zeile müssen daher mehrere Bytecode- Anweisungen ausgeführt werden, d.h. mehrere Einzelschritte per Funktionstaste F7.

Um die Anzeige im Disassembler zu einer bestimmten Code-Adresse zu scrollen, kann in der Symboltabelle *die hexadezimale Adresse* einer selbstdefinierten Funktion oder Prozedur angeklickt werden.

Um das Disassembler-Fenster zu schließen, und wieder in den 'normalen' Einzelschrittbetrieb, d.h. eine Quelltext-Zeile pro Schritt zurückzukehren, schalten Sie in der Combo-Box oberhalb der Disassembler-Listings von *Disassembler* auf den Eintrag *Debug-Anzeigen verbergen* (Hide Debug View) um.

6 2.3.3 Trace History

Die Trace-Historie dient zum Verfolgen der Programmausführung ("Ablaufverfolgung"). Diese Funktion ist bei vielen Geräten in der Firmware, aber auch im Programmiertool (Simulator) enthalten. Die Historie (Verlauf) ist typischerweise auf 255 Einträge begrenzt; beim Überschreiten werden die ältesten Einträge überschrieben (FIFO = first in, first out).
Der Trace-FIFO kann aber ggf. auch vom Script selbst gestoppt werden, z.B. wenn dies eine Fehlfunktion oder einen Protokollfehler auf dem CAN-Bus festgestellt hat.
In der Historie *können* die folgenden Ereignisse aufgezeichnet werden:

- Vom Terminal *gesendete* CAN-Telegramme
- Vom Terminal *empfangene* CAN-Telegramme
- Fehlermeldungen und Warnungen (aus der Terminal-Firmware bzw. Simulator)
- Vom Script mit dem Kommando trace.print in die Historie 'gedruckte' Zeilen
- Weitere Ereignisse, wenn diese per trace.enable aktiviert wurden

Beim Abschalten des Gerätes wird die Trace-Historie gelöscht, da sie aus Performance-Gründen nur im Hauptspeicher (RAM) gepuffert wird. Sie kann (und soll) kein Ersatz für den in manchen Geräten integrierten CAN-Logger / 'Snooper' sein. Wie weiter unten beschrieben, kann der Inhalt des Trace-Speichers für eine spätere Analyse als Textdatei gespeichert werden. Die Trace History eignet sich daher auch als Hilfsmittel für die Fehlersuche im Zusammenhang mit CAN-Protokollen (Stichwort CCP / XCP !).
Übersicht: Anzeigeformat der Trace History, Verwendung der Trace History, Aufruf der Trace History .

6.1 2.3.3.1 Anzeigeformat der Trace History

Die Anzeige von CAN-Telegrammen in der Historie orientiert sich am weit verbreiteten Vector-"ASCII"-Logfile-Format. Damit ist auch bei Geräten ohne integrierten CAN-Logger eine bescheidene Fehleranalyse möglich (hilfreich bei der Implementierung von 'exotischen' CAN-Bus-Protokollen in der Script-Sprache).

Anzeigeformat von CAN-Telegrammen (ähnlich Vector-ASCII-Format) in der Trace-Historie:

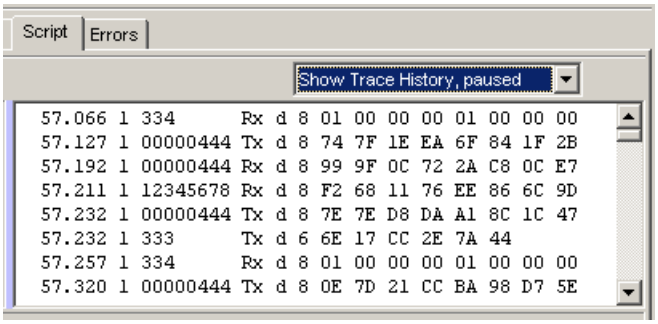
Timestamp	Bus	CAN-ID	Rx/Tx	d	Length	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	By
57.211	1	12345678	Rx	d	8	F2	68	11	76	EE	86

CAN-Identifier mit 11 Bit (standard frames) werden mit 3 hexadezimalen Ziffern angezeigt. 29-Bit-Identifier (extended frames) sind wie im Beispiel an der 8-ziffrigen Darstellung erkennbar. LIN-Bus-Frames werden (optional) in der Trace-Historie wie CAN-Telegramme angezeigt. Ein ebenfalls 'Vector-ASCII'-ähnliches Format wird auch erzeugt, wenn ein CAN-Telegramm vom Script-Datentyp tCANmsg mit der Funktion string() in eine Zeichenkette umgewandelt wird.
Per 'trace.print' in die Historie 'gedruckte' Zeilen sind formatfrei; die Zeitmarke (in Sekunden, mit drei Nachkommastellen) wird automatisch bei der Anzeige hinzugefügt. Die Zeitmessung beginnt bei Null beim Einschalten des Gerätes, bzw. beim Start des Simulators.

Die vom Script mit dem Kommando trace.print in die Trace-Historie 'gedruckten' Zeilen sind prinzipiell formatfrei. Lediglich die Zeitmarke wird (am Anfang der Zeile) automatisch von der Geräte-Firmware eingefügt.

6.2 2.3.3.2 Verwendung der Trace History

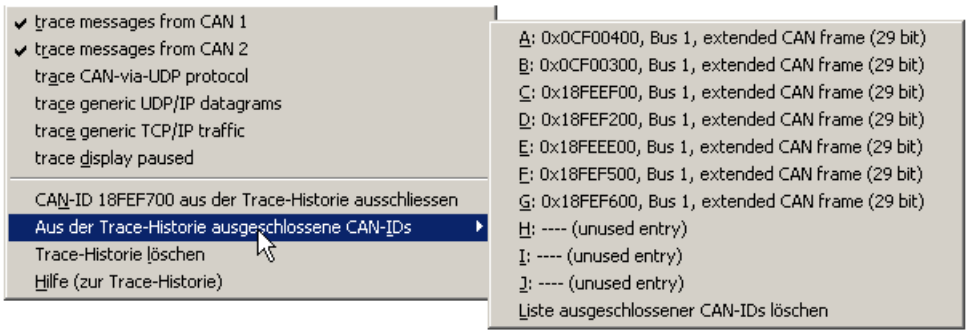
Im Simulatorbetrieb (Programmiertool) kann die Trace-Historie auf der rechten Hälfte der Registerkarte 'Script' angezeigt werden. Dazu in der Combo-Box in der Toolbar den Eintrag 'Show Trace History' auswählen:



Anzeige der Trace History im Programmiertool

Im Zustand 'Trace History paused' können mit dem vertikalen Scroller auch alte Einträge sichtbar gemacht werden.
Bei 'Trace History running' werden neue Einträge ständig am Ende angefügt, und der Scroller bei jedem neuen Eintrag an das Ende der Liste bewegt, damit der neueste Eintrag immer sichtbar ist.

Per Rechtsklick in die Trace-Historie (im Programmiertool) wird das folgende Kontext-Menü geöffnet:



Kontextmenü zum Steuern der Trace-Historie im Programmiertool

Im oben gezeigten Menü kann u.A. die Anzeige bestimmter CAN-Identifier in der Trace-Historie unterdrückt werden. Damit kann z.B. vermieden werden, dass die Trace-Historie mit besonders häufig übertragenen oder uninteressanten Telegrammen 'überflutet' wird.

Am echten Gerät kann die Trace-Historie per Systemmenü unter 'Diagnostics' .. 'TRACE History' auf dem Display angezeigt werden, oder -je nach Hardware- per serieller Schnittstelle oder per Web-Browser (Adresse: <hostname>/trace.htm) ausgelesen und in einer Textdatei gespeichert werden. Details zum Aufruf des System-Menüs finden Sie in Dokument Nr. 85115 (Systemmenü und Setup-Optionen für programmierbare Terminals von MKT).
Siehe auch: 'Aufruf der Trace History' in *diesem* Dokument, Packet Capture (Wireshark-kompatibel)

.

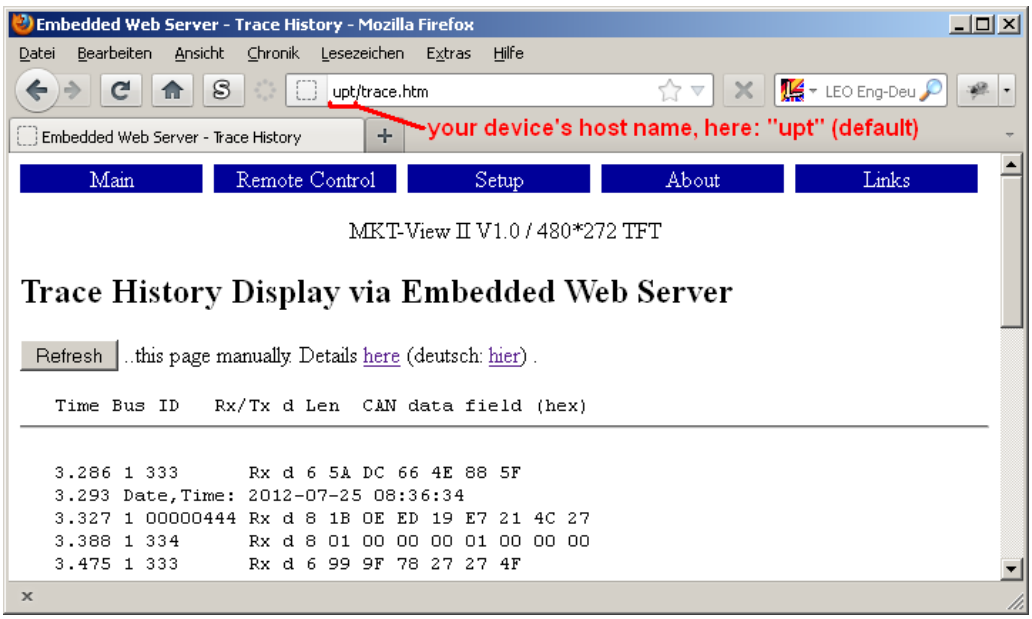
6.3 2.3.3.3 Unterdrücken bestimmter CAN-Identifier in der Trace-Historie

Um einen bestimmten CAN-Message-Identifier für die Anzeige in der Trace-Historie zu blockieren, klicken Sie zunächst mit der *rechten* Maustaste auf den entsprechenden Identifier in der Trace-Anzeige, und wählen danach im Kontext-Menü (siehe Screenshot im vorhergehenden Kapitel) die Funtion 'CAN-ID aus der Trace-Historie ausschliessen'.
Alternativ können die Einträge in der Sperr-Liste auch manuell per Kontextmenü bearbeitet werden. Öffnen Sie dazu das Untermenü 'Aus der Trace-Historie ausgeschlossene CAN-IDs', und wählen den zu ändernden Eintrag aus. Der Wert FFFFFFFF (hexadezimal) ist kein gültiger CAN-Identifier, er kennzeichnet einen 'nicht genutzten Eintrag' (im Screenshot: 'unused entry') in der maximal 10 Elemente umfassenden Sperr-Liste.
Die hier beschriebene Sperrung bestimmter CAN-Identifier wirkt sich nur auf die Trace-Historie *im Programmiertool*, aber nicht auf die Trace-Historie *im 'echten' Gerät* (z.B. MKT-View III) aus.

Im 'echten' Gerät kann (wie auch im Simulator) *das Script selbst* mit der Funktion trace.can_blacklist[i] auf die CAN-ID-Sperrliste zugreifen.

6.4 2.3.3.4 Auslesen der Trace-Historie per Web-Browser

Bei Geräten mit Ethernet-Schnittstelle kann die Trace-Historie *aus dem laufenden Gerät* per LAN und Web-Browser ausgelesen. Dies ist bei allen Geräten mit Ethernet-Anschluss und integriertem Web-Server möglich. In den meisten Fällen kann das Gerät direkt über seinen Host-Namen angesprochen werden. Dieser ist per Default auf 'UPT' eingestellt (auch bei 'MKT-View' und Co), kann aber per Netzwerk-Setup am Gerät geändert werden. Die komplette URL wäre in diesem Beispiel "http://upt/trace.htm", der Protokoll-Name (http) kann aber i.A. weggelassen werden bzw. wird vom Browser nicht in der Adressleiste angezeigt. Hier z.B. die Anzeige der Trace-Historie in einem 'MKT-View II' im vom Autor bevorzugten Browser:

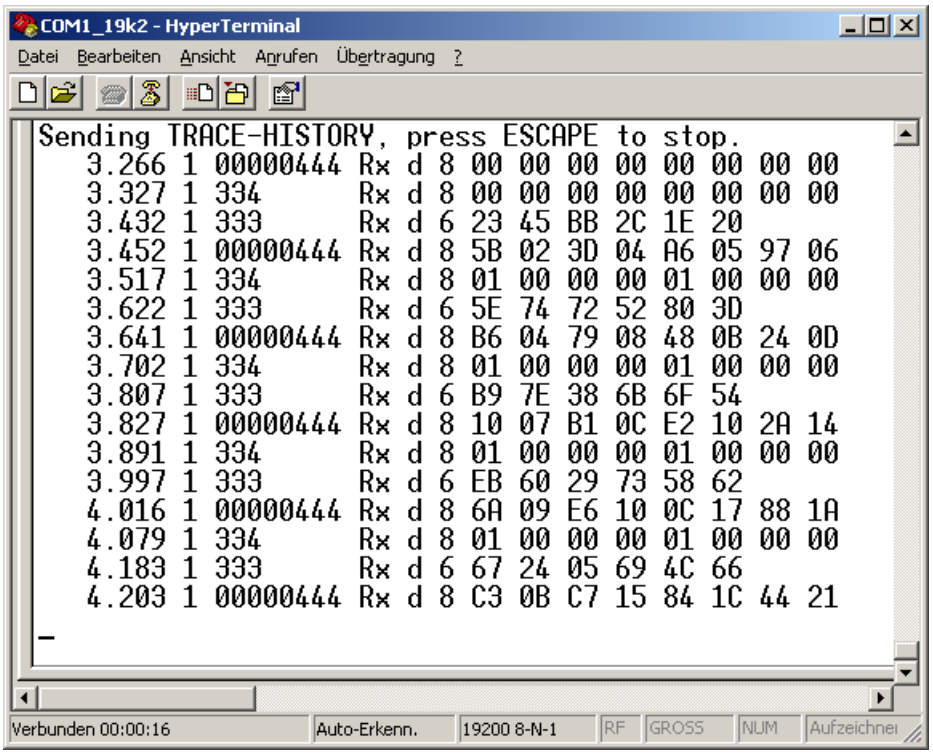


Auslesen der Trace-Historie per Embedded Web Server, und Anzeige per Web-Browser

Tipps bei Problemen mit dem Verbindungsaufbau per TCP/IP (Web-Browser) finden Sie unter 'Troubleshooting' in der Beschreibung des im Display integrierten Web-Servers.

6.5 2.3.3.5 Auslesen der Trace-Historie über die serielle Schnittstelle

Zum Auslesen per serieller Schnittstelle (RS-232 oder Virtual COM Port) geben Sie im Terminalprogramm (z.B. 'Hyperterminal') das Kommando ***trace*** ein. Wie üblich muss jedes Kommando mit der ENTER-Taste (d.h. Carriage Return) abgeschlossen werden. Die Default-Baudrate der seriellen Schnittstelle bei den meisten Geräten mit "echter" RS-232-Schnittstelle (z.B. MKT-View 2) beträgt 19200 Bit/Sekunde. Bei Geräten, die statt einer RS-232-Schnittstelle einen virtuellen COM-Port enthalten (der 'von Außen' wie eine USB-Device-Schnittstelle aussieht) sind 115 kBit/Sekunde voreingestellt. Da die serielle Schnittstelle allerdings per Kommando umkonfiguriert werden kann, sind diese Kommunikations-Parameter (115200 8-N-1 bzw. 19200 8-N-1) nicht unbedingt gültig.



Anzeige der Trace-Historie in 'HyperTerminal'

Bei Windows 'Vista' und Windows 7/8/10 gehört HyperTerminal leider nicht mehr zum Lieferumfang. Wir empfehlen in diesem Fall den Einsatz von PuTTY, um den Trace-Speicher über die serielle Schnittste auszulesen.

6.6 2.3.3.6 Abspeichern der Trace-Historie als Datei

Als letzte Möglichkeit, falls der PC (oder das lokale Netzwerk) keine TCP/IP-Verbindung mit dem Terminal aufbauen kann, kann die Trace-Historie direkt am Terminal als Text-Datei auf der Speicherkarte gespeichert werden. Rufen Sie dazu wie hier beschrieben die Trace-Historie auf, betätigen den 'Menu'-Button (Softkey), und wählen den Menüpunkt 'Save Trace as file'. Daraufhin speichert die Firmware im Display den Inhalt der Trace-Historie unter dem Namen 'TRACE000.TXT'. Bei jedem *weiteren* Aufruf dieser Funktion wird eine neue Datei angelegt (TRACE001.TXT, TRACE002.TXT, und so weiter).

Die im Simulatorbetrieb (*im Programmiertool*) erzeugte Trace-Historie kann wie folgt über die

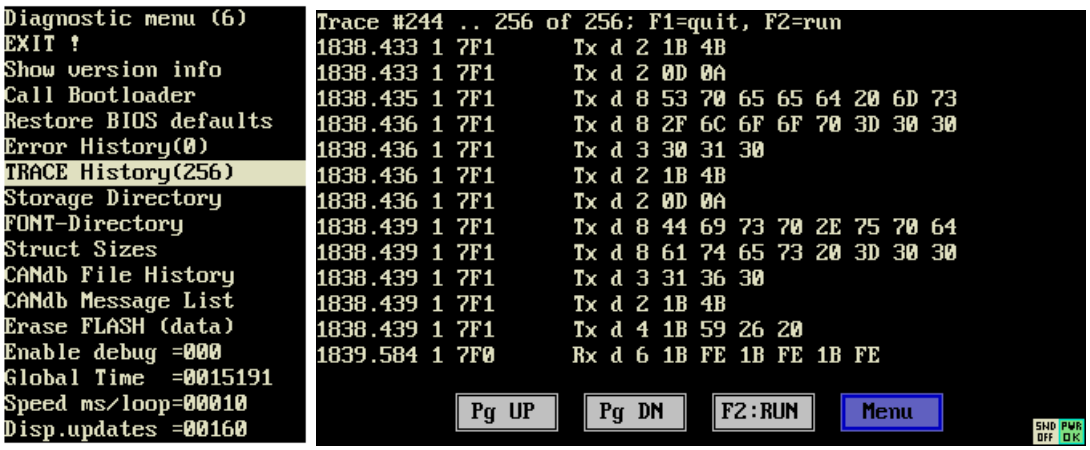
Windows-Zwischenablage (als 'Rich Text', d.h. formatierter Text) in eigene Dokumente / Dateien übernommen werden:

1. Fokus per Mausklick in die Trace-Historie setzen
2. STRG-A drücken (alles markieren) oder per Maus den 'interessanten Teil' in der Historie markieren
3. STRG-C drücken, um den markierten Text wie üblich in die Windows-Zwischenablage zu kopieren
4. In das eigene Dokument umschalten, und per STRG-V den Text aus der Zwischenablage einfügen

6.7 2.3.3.7 Aufruf der Trace History

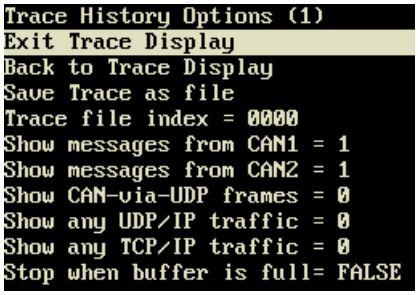
Die Anzeige der Trace History (auf dem Bildschirm des Gerätes) kann auf verschiedene Weisen geöffnet werden.
Hier z.B. der Aufruf der Historie beim MKT-View II / MKT-View III und ähnlichen Geräten mit Touchscreen:

1. Malen Sie die Geste 'U' auf dem Touchscreen, um das Shutdown-Fenster zu öffnen. Alternativ (für Geräte ohne Touchpanel): Drücken Sie F2+F3 gleichzeitig, um das System-Menü zu öffnen.
2. Im Dialogfenster 'Shutdown' wählen Sie 'SETUP'.
3. Im 'Main system setup' wählen Sie 'DIAGNOSTICS'.
4. Im 'Diagnostic menu' wählen Sie 'Trace History'. Die Nummer in Klammern zeigt die Anzahl der Einträge in der Trace-Historie.



Aufruf der Trace-Historie per Systemmenü im Terminal, und Anzeige auf dem "lokalen" Bildschirm

Über den Softkey 'Menu' (im obigen Screenshot rechts unten, blau hinterlegt = per Drehknopf / ENTER anwählbar) wird das folgende Menü geöffnet:



Optionen der Trace-Historie, hier beim MKT-View III

Bei *den meisten* (aber nicht allen) Geräten sind die folgenden Einträge im oben gezeigten Options-Menü vorhanden:

`Exit Trace Display`
Verlässt die Anzeige der Trace-Historie, und kehrt zum Aufrufer zurück (i.A. dem Systemmenü)

`Back to Trace Display`
Verlässt das Options-Menü, und kehrt zur Anzeige der Trace-Historie zurück

`Save Trace as file`
Speichert den Inhalt der Trace-Historie als Textdatei auf der Speicherkarte. Details hier.

`Show messages from CAN1 = {0,1}`
1 (Default): Telegramme vom ersten CAN-Port in der Trace-Historie anzeigen. 0: Diesen Port nicht anzeigen.

`Show messages from CAN2 = {0,1}`
1 (Default): Telegramme vom zweiten CAN-Port in der Trace-Historie anzeigen. 0: Diesen Port nicht anzeigen.

`Show messages from CAN-via-UDP = {0,1}`
1: Über ein spezielles UDP-Protokoll 'getunnelte' Telegramme anzeigen. 0 (Default): nicht anzeigen.

`Stop when buffer is full`
Spezialoption zum Überwachen von Problemen beim 'Anlassen'.
TRUE : Die Trace-Historie wird gestoppt, wenn der Speicher voll ist. Daher sind nur die 'ältesten' Einträge abrufbar.
FALSE: Die Trace-Historie läuft ständig weiter, und nur die 'neuesten' Einträge sind abrufbar.

Die Default-Einstellung ist hier 'FALSE', d.h. die Trace-Historie wird nicht (automatisch) gestoppt, und es sind jeweils nur die letzten 255 oder 511 (Firmware-abhängig) Einträge abrufbar.

Siehe auch:

Auslesen der Trace-Historie per Web-Browser (Ethernet,HTTP),
Wireshark-kompatible Packet-Capture-Funktion (auch für CAN, als Alternative zur TRACE-Historie)

Zurück zur Übersicht zum Thema 'Trace History'

7 2.3.4 Stack-Anzeige (mit Adressen für Funktionsarufrufe und lokalen Variablen)

Während single-step-Debugging werden in der Statuszeile des Script-Editors die oberen Elemente des Stacks angezeigt. Diese Funktion wurde hauptsächlich während der Implementierung der Script-Sprache verwendet. Sie kann allerdings auch beim 'normalen' Debuggen hilfreich sein, z.B. wenn in einer Script-Anwendung viele Unterprogramm-Aufrufe erfolgen (oder rekursive Aufrufe der gleichen Funktion). Mit Hilfe der Stack-Anzeige können fortgeschrittene Anwender erkennen, "wie" das Programm an die aktuelle Stelle kam, welche Funktionsparameter bei jedem Aufruf übergeben wurden, und welche Zwischenergebnisse beim Abarbeiten der RPN entstehen. Details zu diesem nicht-trivialen Thema finden Sie im englischen Original dieses Dokuments.

Der Stack kann sowohl in der Statuszeile als auch *mehrzeilig* in einer Textliste angezeigt werden. Beispiel für die kurze (einzeilige) Anzeige der 'Spitze' des Stacks in der Statuszeile:

```
Stack[6] : tCANmsg 0 return_to_822 1 65230 0
```

Um den Stack *komplett* anzuzeigen (inkl. Strukturen, z.B. CAN-Messages) wählen Sie in der Auswahlliste (rechts oben auf der Registerkarte 'Script') den Eintrag 'Stack anzeigen'. Beispiel:

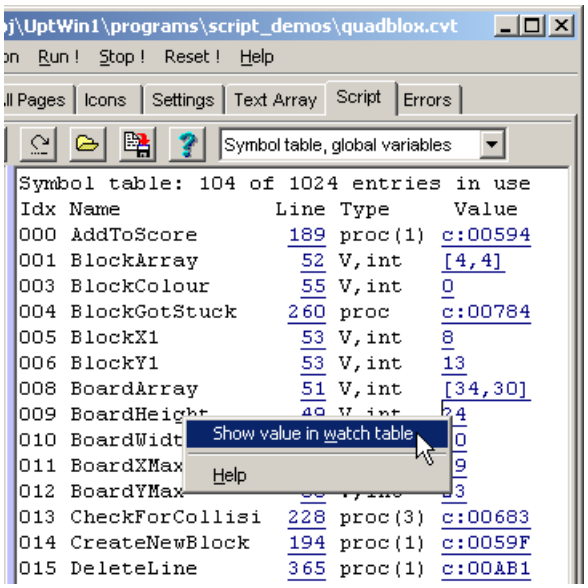
```
Stack[005]*: tCANmsg={ 1CEA00FF 3 CE FE 00 }
Stack[004] : 0
Stack[003] : return to line 822
Stack[002] : 1
Stack[001] : 65230
Stack[000] : 0
```

Im oben gezeigten Beispiel ist der obere Eintrag (Stack-Index 5) die 'Spitze' des Stacks (top of stack) mit der Anzeige eines dort abgelegten CAN-Telegramms (Format: ID, Anzahl Datenbytes, Datenbytes, hexadezimal).
Der Eintrag an Index 3 ist eine Rücksprungadresse (aus einem Prozedur- oder Funktionsaufruf). Durch Anklicken der unterstrichenen Zeilennummer (hier : Quelltext-Zeile 822) kann der Funktionsaufruf in den sichtbaren Bereich des Quelltext-Editors gescrollt werden kann.
Einträge ohne explizite Angabe des Typs sind Integer- oder Fliesskommawerte. Zeichenketten (Strings) werden wie üblich zwischen doppelten Anführungszeichen angezeigt.

Hinweis: Das Element auf der Spitze des Stapels (top of stack) hat den *höchsten* Index. Direkt nach der Initialisierung (mit 'leerem' Stack) ist der Stack-Index *Null*.

8 2.3.5 Anzeige der Symboltabelle (mit Variablen oder/und Funktionsnamen)

Die vom Compiler erzeugte Symboltabelle kann in der rechten Hälfte des Programmiertools angezeigt werden. Bei 'einfachen' globalen Script-Variablen werden dabei auch die aktuellen Werte angezeigt. Wählen Sie dazu den Eintrag 'Symboltabelle, komplett' oder 'Symboltabelle, globale Variablen' in der Combo-Box in der Toolbar des Script-Editors.
Die 'komplette' Symboltabelle enthält auch die Namen und Quelltext-Positionen von *lokalen* Variablen, deren Inhalt aus technischen Gründen bislang noch nicht inspiziert werden kann.
Die 'globalen' Symbole umfassen in erster Linie globale Script-Variablen, aber auch benutzerdefinierte Funktionen, Prozeduren, Datentypen und Konstanten.
In der tabellenähnlichen Ansicht sind die Symbole alphabetisch sortiert. Die Symboltabelle eignet sich daher auch zum 'Zurechtfinden' in einem komplexen Script: Per Klick auf die Zeilennummer ('Line', als Hyperlink dargestellt) mit der *linken* Maustaste scrollt der Script-Editor zur entprechenden Deklaration bzw Definition des Symbols im Quelltext.



Screenshot mit Anzeige der Symboltabelle im Programmiertool

Durch Anklicken des (ebenfalls als Hyperlink markierten) 'Wertes' eines Symbols mit der *rechten* Maustaste öffnet sich ein Popup-Menü (Kontext-Menü), in dem u.A. das angeklickte Symbol zu der im nächsten Kapitel beschriebenen Watch-Liste hinzugefügt werden kann.
Durck Anklichen eines blau unterstrichenen *Variablennamens* mit der *linken Maustaste* können Sie zur Deklaration der entsprechenden Variablen im Script-Quelltext umschalten. Entsprechendes gilt (bei Anzeige der 'kompletten' Symboltabelle) auch für Datentypen, Prozedur- und Funktionsnamen.
Durch Anklicken einer blau unterstrichenen *Code-Adresse* (z.B. c:0ABC) in der Symboltabelle kann in den Disassembler umgeschaltet werden, der dann zur angeklickten Adresse scrollt.

Hinweis:

Bei Geräten mit Ethernet-Schnittstelle und HTTP-Server (z.B. MKT-View III/IV) können *globale Script-Variablen* alternativ auch per Web-Browser inspiziert werden.

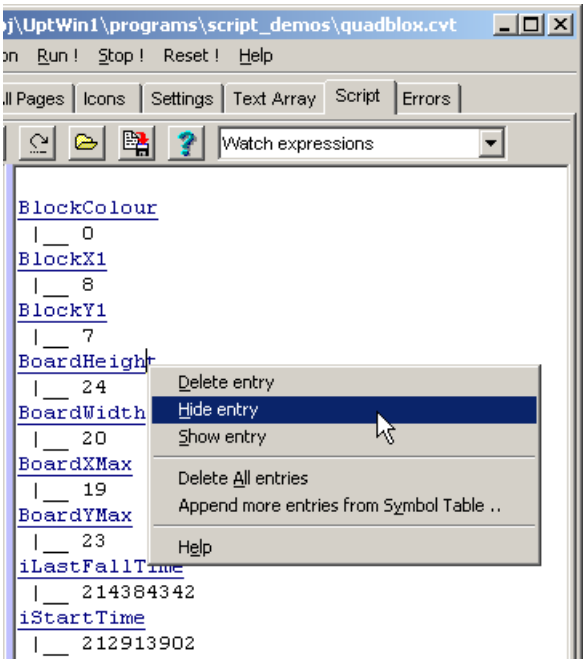
Tipp:

Bei Geräten mit einem ausreichend großen Display können Script-Variablen auch im System-Menu inspiziert werden. Scrollen Sie dazu im System-Menü bis zum Untermenü 'Script:' /

'Variables ..' . Details zum Beobachten ('Watch') einer Variablen *im programmierbaren Gerät* finden Sie hier.

9 2.3.6 Anzeige einzelner Variablen ("Watch-Liste")

Im Gegensatz zur im vorherigen Kapitel vorgestellten *Symboltabelle* enthält die *Watch-Liste* eine benutzerdefinierte Auswahl von globalen Script-Variablen, die während des Debugger-Betriebs im Programmiertool angezeigt werden können.
Dies funktioniert mit 'einfachen' Script-Variablen, aber auch mit Arrays und benutzerdefinierten Datentypen.
(für Experten: Die Auswertung von 'beliebigen' Ausdrücken ist bislang noch nicht möglich)
Um neue Einträge zur Watch-Liste hinzuzufügen verwenden Sie z.B. die Symbol-Tabelle (Klick auf den Wert, danach "Wert in Watch-Liste anzeigen"), oder das Kontextmenü des Script-Editors.



Screenshot einer Watch-Liste auf der Registerkarte "Script" im Programmiertool

Durch Anklicken des Variablennamens (bzw in Zukunft des auszuwertenden Ausdrucks) in der Watch-Liste wird ein Kontext-Menü mit den folgenden Funktionen geöffnet:

Eintrag löschen (Delete entry)
: Löscht den angeklickten Eintrag aus der Watch-Liste

Eintrag verbergen (Hide entry)
: Unterdrückt die Anzeige des *Wertes* in der Watch-Liste, ohne den Eintrag zu löschen. Lediglich der Variablenname bleibt sichtbar. Dadurch kann z.B. bei der Anzeige von Arrays und größeren Strukturen viel Platz gespart werden, ein ein bestimmtes Element momentan nicht benötigt wird.

Eintrag wieder anzeigen (Show entry)
: Macht den *Wert* wieder sichtbar, nachdem er mit der obigen Funktion verborgen wurde.

Alle Einträge löschen (Delete all entries)
: Entfernt mit einem Mausklick *alle* Einträge in der Watch-Liste. Hilfreich z.B. beim Umschalten von einem Projekt zum anderen, bevor eine 'komplett neue' Watch-Liste zusammengestellt wird.

Weitere Einträge aus Symboltabelle (Append more entries from symbol table)

Schaltet von der Watch-Liste zur Anzeige der Symboltabelle um, z.B. um dort neue Einträge (globale Variablen) für die Watch-Liste auszuwählen.

Siehe auch: Watch-Fenster des Programmiertools (in dem mit dem Vorsatz 'script.' auch Script-Variablen untersucht werden können)

10 2.3.7 Anzeige der dynamisch allozierten Speicherblöcke

Um Details zur aktuellen Verwendung des dynamisch allozierten Speichers zu erhalten, wählen Sie in der Combo-Box (rechts oben in der Script-Editor-Toolbar) die Funktion *Dynamisch allozierte Blöcke*. In der rechten Hälfte des Script-Panels werden dann alle momentan verwendeten Speicherblöcke angezeigt, zusammen mit den Namen der Variablen, mit denen diese Blöcke zusammenhängen.

```
Dynamisch allozierte Blöcke
Block[0000] :   64 byte, Timer1
Block[0004] :   64 byte, Info
Summary: 2 blocks in 2 objects, 4094 blocks free.
```

(Beispiel zur Anzeige der dynamisch allozierten Speicherblöcke)

Ähnlich wie in anderen Debugger-Ansichten können Sie auch hier per Mausklick (linke Taste) auf einen blau unterstrichenen Variablennamen zur Deklaration der entsprechenden Variablen im Script-Quelltext umschalten.

11 2.3.8 Testen der Applikation im RAM (statt FLASH) des Zielsystems

Um während der Entwicklungsphase beim Hochladen der Applikation in das Zielsystem Zeit zu sparen, kann die zu testende Applikation dort auch direkt in den Hauptspeicher (RAM) geladen werden, ohne sie -wie üblich- dauerhaft im Flash-Speicher des Zielsystems abzulegen.
Dadurch entfällt das bei einigen Geräten recht langwierige Löschen der Flash-Sektoren während der Übertragung, unabhängig vom Übertragungsmedium (CAN, serielle Schnittstelle, Ethernet).
Wählen Sie dazu im Hauptmenü des Programmiertools die Funktion
Transfer .. ***Applikation ins Terminal laden OHNE zu Flashen*** .

Hinweis:

Es wird dringend empfohlen, die Applikation nicht nur im Simulator, sondern auch im 'echten' Gerät zu testen !
Die Ausführungsgeschwindigkeiten unterscheiden sich, abhängig von der verwendeten CPU, z.T. deutlich. So kann z.B. eine Script-Applikation, die auf dem MKT-View IV 'flüssig' und fehlerfrei funktioniert, auf dem MKT-View II deutlich langsamer laufen, was beim intensiven Einsatz von Event-Handlern die im Kapitel 'Event-Handling' beschriebenen Timeout-Probleme verursachen könnte.

Zurück zur Übersicht zum Thema 'Debugging'

3. Interaktion zwischen Script und Display (d.h. den "programmierbaren Anzeigeseiten")

In manchen Anwendungen arbeitet das Script nur "im Hintergrund", z.B. um per CAN empfangene Telegramme zu verarbeiten, oder um Protokolle abzuarbeiten die nicht in der Firmware implementiert sind.
In vielen Anwendungen greift das Script aber auch direkt in die "programmierbaren Anzeigeseiten" ein, z.B.:

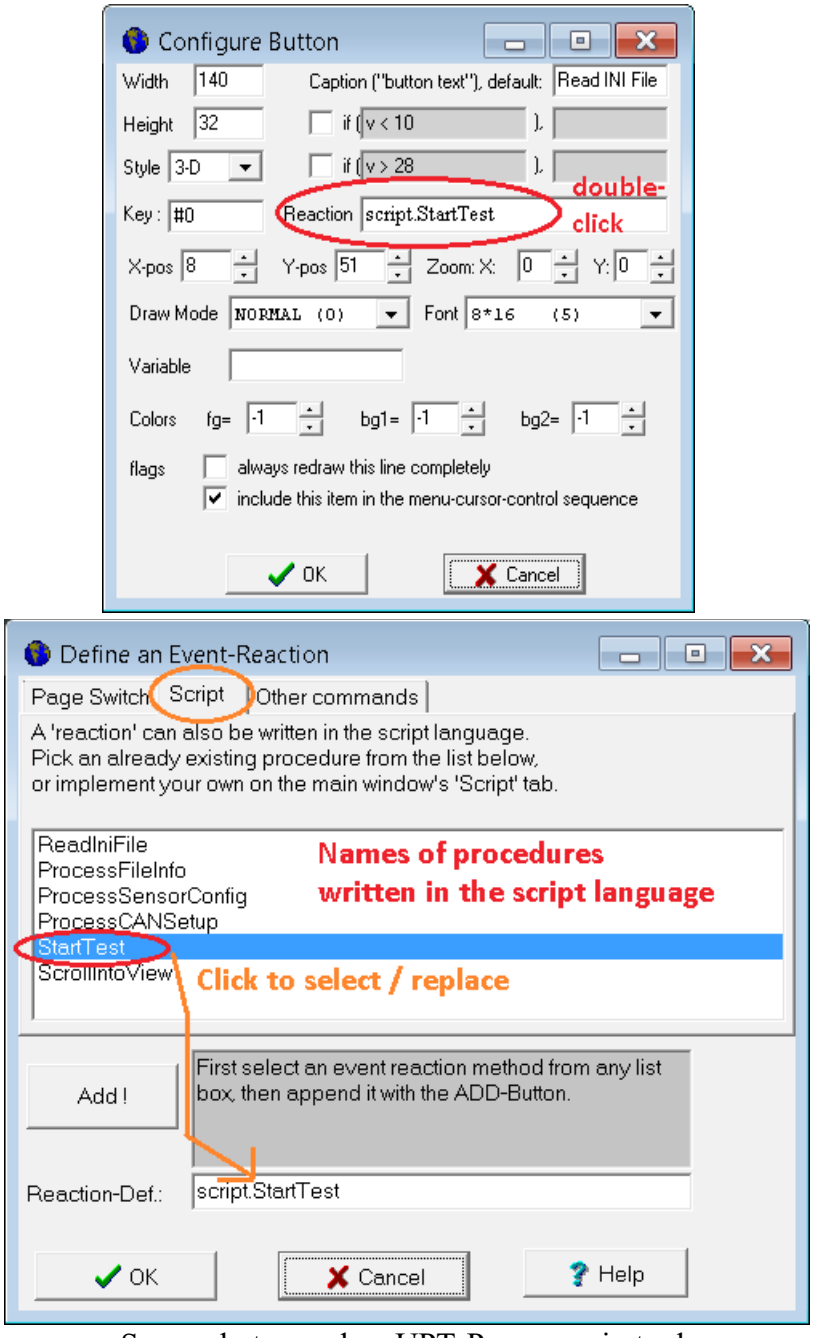
- Der Bediener betätigt einen Button (per Touchscreen), und als "Reaktion" (vom Button aufgerufene Kommandozeile) wird eine Script-Variable gesetzt (die dann, wenige Millisekunden später, in der Hauptschleife des Scripts weiterverarbeitet wird);
- Der Bediener betätigt einen Button (per Touchscreen), und als "Reaktion" (des Buttons) wird direkt eine in der Script-Sprache geschriebene Prozedur aufgerufen;
- Das Script erkennt einen kritischen Wert in einer der (z.B. per CAN empfangenen) 'Display-Variablen', und ändert daraufhin ein Anzeige-Element, z.B. wechselt die Farbe von 'Grün' nach 'Rot' um den Bediener darauf hinzuweisen;
- Das Script fängt bestimmte Benutzeraktionen per Event-Handler ab (ein fortgeschrittenes Thema "für später").

Siehe auch (Links zu anderen Kapiteln mit weiterführenden Informationen) :

- Zugriff auf ***Display-Variablen*** per Script
- Zugriff auf ***Script-Variablen*** per Display-Interpreter
- Zugriff auf ***Anzeigeelemente*** (auf der aktuellen Anzeigeseite) per Script
- Aufruf von ***Script-Prozeduren*** aus dem Display-Interpreter
- Aufruf von ***Script-Prozeduren*** aus Display-Seiten (z.B. um mehrsprachige Anzeigetexte für die Anzeige zu gewinnen, d.h. 'Internationalisierung')

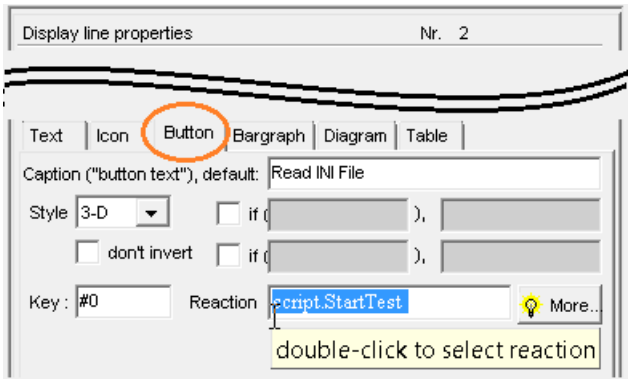
1 3.1 Aufruf einer Script-Prozedur beim Betätigen eines Buttons

Vor der Implementierung der Script-Sprache bestand die 'Reaktion' beim Betätigen eines Buttons *immer* aus einer Kommandozeile *für den (alten) Display-Interpreter* (z.B. "g(pn+1)" zum Umschalten auf die nächste Anzeigeseite).
Aus Kompatibilitätsgründen ist dies weiterhin möglich. Für fortgeschrittene Anwendungen (und Entwickler) empfiehlt es sich aber, die per Button ausgelöste "Reaktion" stattdessem direkt (und nur) in der Script-Sprache zu implementieren.
Dazu bieten sich mehrere Möglichkeiten (die bereits in der Einleitung von Kapitel 3 beschrieben wurden).
Im folgenden Beispiel wird per Button eine benutzerdefinierte Script-Prozedur aufgerufen.



Screenshots aus dem UPT-Programmiertool.
Details im Dokument über programmierbare Buttons

Per Doppelklick in das Feld 'Reaktion' (s.O., linkes Bild) öffnen Sie ein Auswahlfenster zum Definieren der Reaktion (rechtes Bild). Unter 'Script' finden Sie dort eine Auflistung aller *bereits im Script vorhandenen Prozeduren*, die sich prinzipiell für den Aufruf 'per Button' eignen würden. Wählen Sie eine der aufgelisteten Prozeduren (in diesem Beispiel "StartTest"), oder (für fortgeschrittene Anwender, die den Einsatz der Tastatur bevorzugen) geben Sie den Aufruf der Script-Prozedur direkt unter 'Eigenschaften..' ... 'Schaltfläche' ein (im Screenshot: Display line properties ... Button ... Reaction):



Screenshot aus dem Programmiertool, "Display Line Properties" / *Button*

Dem Namen der Prozedur sollte das Schlüsselwort "script." vorangestellt werden, um die Arbeit des *Display-Interpreters* zu vereinfachen.
Wenn die "Reaktion" eines Buttons bereits den Aufruf einer Script-Prozedur enthält, können Sie direkt aus der Definition des Buttons (per Doppelklick im oben gezeigten "Display Line Properties" / Button / Reaction) zur Implementierung der Prozedur im Script-Editor umschalten. Der Editor scrollt automatisch zur Prozedur, und markiert die Zeilennummer mit dem Prozedurkopf in grüner Farbe.
Der unten gezeigte Script-Quelltext enthält eine "per Button" aufgerufene Prozedur (Quelle: "Ini-File"-Demo):

```
proc StartTest // Aufruf vom DISPLAY (Reaktion auf Button")...
  iStartTest := TRUE; // nur ein Flag setzen, der Rest erfolgt in der
Hauptschleife
endproc;
```

In der oben gezeigten Prozedur wird lediglich ein 'Flag' ("Merker", hier: iStartTest) gesetzt. Die eigentliche Arbeit (die u.U. etliche Sekunden dauern könnte) erfolgt per Polling in der Hauptschleife des Scripts.
Dadurch werden Laufzeitprobleme (z.B. Timeouts oder Blockieren der Anzeige) vermieden. Es gelten die gleichen Hinweise zum Aufruf von Script-Event-Handlern aus Kapitel 4.11 (gelbe Textbox).

2 3.2 Ändern eines Anzeige-Elements per Script (Text, Farben, usw.)

Das Script hat die volle Kontrolle über die programmierbaren Anzeigeseiten, und kann die Anzeige-Elemente gegebenenfalls auch *während der Laufzeit* "umprogrammieren" (siehe Einleitung in Kapitel 3). Wenn nötig, kann so z.B. die Farbe, der Text, die Position, Größe, und die Sichtbarkeit eines Anzeige-Elements geändert werden. Dazu dienen die in Kapitel 4 spezifizierten Zugriffsmethoden (display.elem[] oder display.elem_by_id[]).

Die in den meisten Fällen bevorzugte Methode ist der Zugriff auf ein Anzeigelement über dessen Namen (siehe folgendes Beispiel), oder dessen symbolischen Control-ID (der beim Entwurf der Anzeigeseite im Designer festgelegt wurde, als String mit maximal 20 Zeichen).
Das folgende Fragment aus der Hauptschleife eines Scripts ändert die Hintergrundfarbe eines Buttons, abhängig vom "Erfolg" oder "Misserfolg" beim Einlesen einer Konfigurationsdatei :

```
while(1)  // endless loop for the script's MAIN THREAD

  if( iStartTest ) then
    if( ReadIniFile( "memory_card/IniDemo1.ini") ) then
       display.elem["ReadIni"].bc = clGreen; // paint the button with a GREEN
background
    else  // could NOT read the ini-file :
       display.elem["ReadIni"].bc = clRed;   // paint the button with a RED
background
    endif;
    iStartTest := FALSE;
  endif;

  // ... weitere in der 'Main Task' zu erledigende Aktionen hier ...

  wait_ms(50);  // give the CPU to someone else for 50 milliseconds
endwhile; // end of the main thread loop
```

Das oben gezeigte Fragment stammt aus dem 'Ini-Files'-Demo, zu finden im Unterverzeichnis 'programs/script_demos' nach der Installation des Programmiertools. Weitere Beispiele finden Sie in Kapitel 5.

4. Sprachbeschreibung (Referenz)

Die Script-Sprache basiert auf einer strukturierten, BASIC-ähnlichen Programmiersprache ("BASIC ohne Zeilennummern"). Sie wurde später um einige Elemente aus Programmiersprachen wie PASCAL und IEC 61131 "Structured Text" erweitert. Von diesen Programmiersprachen erbten Teile der Script-Sprache (leider) auch die case-***in***sensitivity, d.h. bei der Sprache fest eingebauten Schlüsselwörter wurde (früher) nicht zwischen Groß- und Kleinschift unterschieden. Trotzdem sollte unbedingt ein einheitlicher Stil in Quelltexten eingehalten werden (näheres zum Stil folgt später). Hier die wichtigsten **Schlüsselwörter** (top-level keywords) der Script-Sprache:

if..then..else..endif for..to..next while..endwhile repeat..until
select..case..else..endselect

proc..endproc func..endfunc

const..endconst var..endvar typedef..endtypedef

addr append float int limit local ptr string

CAN cop.(CANopen) display. file. inet. Math. setTimer system. time. gps.
trace. tscreen. vkey. wait_ms wait_resume

Weitere Schlüsselwörter finden Sie in der alphabetisch sortierten Liste, im Kapitel 'Operatoren', im Kapitel 'Konstanten', und in der Liste mit Datentypen.

Um die Kompatibilität mit der ursprünglichen (Basic-ähnlichen) Sprache zu wahren, können mehrere Kommandos in einer Zeile per Doppelpunkt (:) getrennt werden. Der Autor rät allerdings davon ab, mehrere Kommandos "in eine Zeile zu quetschen" - verwenden Sie nur ein Kommando pro Zeile, denn nur so können Sie später (beim Debuggen) einen Breakpoint auf das Kommando setzen.

Um die Script-Sprache an 'moderne' Programmiersprachen anzupassen, kann auch das Semikolon (;) als Trennzeichen zwischen zwei Kommandos, oder (wie in Pascal) grundsätzlich am Zeilenende eingefügt werden. Im Gegensatz zu Pascal, "C", und Java hat das Zeilenende in der Script-Sprache eine syntaktische Bedeutung: Es ist ebenfalls ein Trennzeichen zwischen zwei Kommandos. Darum sind in *den meisten* Fällen weder Doppelpunkt noch Semikolon wirklich notwendig. Ein Semikolon am Zeilenende "schadet allerdings nicht", und wird vom Autor der Scriptsprache wie in "C" verwendet... selbst wenn nur EINE ZEILE PRO STATEMENT verwendet wird. Ein paar Beispiele zum empfohlenen Quelltext-Stil folgen weiter unten.

Das Doppelkreuz, aka 'Hash' (#) am Zeilenanfang kennzeichnet eine einzelne Zeile als Compiler- oder Präprozessor-Direktive. Damit kann der Compiler z.B. angewiesen werden, nur deklarierte Variablen zu verwenden (#pragma strict).

Führende Leerzeichen haben für den Compiler in 'normalen' Quelltextzeilen keine Bedeutung. Sie sollten diese trotzdem freigiebig zum Einrücken ineinander geschachtelter Statements einsetzen, denn die Lesbarkeit des Quelltextes steigt dadurch enorm (es könnte passieren, daß sich der Entwickler weigert, Fehler in einem "grauenhaft verstümmelten" Quelltext zu suchen). Hier ein

einfaches Beispiel, in dem die Verschachtelung von for..to..next und if..then..else..endif durch die Einrückung deutlich wird:

```
Sum := 0.0; // calculate PI ...
for Loop:=1 to 10000  // do 10000 iterations
   if (Loop & 1) <> 0 // odd or even loop count ?
     then Sum := Sum + 4.0 / (2 * Loop + 1); // odd
     else Sum := Sum - 4.0 / (2 * Loop + 1); // even
   endif;
next Loop;
print( "PI is roughly ", Sum );
```

Hinweise zum **Stil in Script-Quelltexten** (nicht zwingend vorgeschrieben, aber *dringend empfohlen*):

- Benutzen Sie keine Tabulator-Zeichen in Quelltexten..
- Verwenden Sie mindestens zwei, besser drei Leerzeichen pro Ebene beim Einrücken. Der Funktionsrumpf (zwischen 'proc' und 'endproc', bzw 'func' und 'endfunc') soll auch eingerückt werden.
  Nur das 'Hauptprogramm' (am Anfang des Script-Quelltextes, die beim Programmstart als erstes abgearbeitet wird), und die Schlüsselwort-Paare **const**/endconst, **var**/endvar, **proc**/endproc, **func**/endfunc werden nicht eingerückt (denn sie befinden sich auf der Haupt-Ebene des Programms - im Gegensatz zu Pascal gibt es in der Script-Sprache keine ineinander verschachtelten Funktionen, d.h. keine Funktion, die nur innerhalb einer anderen Funktion existiert).
- Benutzen Sie keine Tabulator-Zeichen... :)
- Es ist nicht nötig, Schlüsselwörter in GROSSBUCHSTABEN zu schreiben. Dies wurde nur in älteren Versionen dieses Dokuments gemacht (vor der Umstellung von 'plain Text' auf HTML), als Schlüsselwörter noch nicht durch HTML-Tags markiert werden konnten, und auch der Script-Editor noch keine Syntax-Hervorhebung unterstützte. Seitdem sowohl in der Dokumentation, als auch im Script-Editor Schlüsselwörter deutlich erkennbar sind, werden nur noch manche benuzterdeinierte Konstanten (wie in "C") in UPPER CASE geschrieben.
- Benutzen Sie niemals Tabulator-Zeichen in Quelltexten, denn verschiedene Editoren verwenden verschiedene Tab-Größen-Einstellen (Steinzeit-Editoren verwenden 8 Schritte pro Tab, andere 4, einige 3 per Default, und so weiter...). Die Verwendung von Tabulator-Zeichen in Quelltexten wird diese früher oder später in einen grauenhaften Murks verwandeln (speziell wenn mehrere Autoren mit unterschiedlich konfigurierten Editoren an einem Projekt arbeiten, was leider bei vielen Open-Source-Projekten auffällt). Verwenden Sie stattdessem, wie bereits erwähnt, zwei bis drei Spaces (Leeerzeichen) pro Ebene zum Einrücken, und richten Sie das 'beendende' Schlüsselwort (z.B. **next**, **until**, **endif**) passend unter dem dazugehörenden 'beginnenden' Schlüsselwort (z.B. **for**, **repeat**, **if**) aus.
- Ihre Kollegen/Partner/Kunden werden sich freuen, wenn beim ersten Blick auf den Quelltext bereits erkennbar ist, was das Programm später machen soll. Kommt Ihnen das Einrücken oder das Einfügen von Kommentaren in Quelltexten als Zeitverschwendung vor, entwickeln Sie besser keine Automatisierungs-Software.

In den folgenden Kapiteln werden die wichtigsten Elemente der Script-Sprache vorgestellt. Spezielle Kommandos und seltener verwendete Funktionen folgen später.

Siehe auch: Liste mit Schlüsselwörtern, Operatoren (numerisch), Anwenderdefinierte Funktionen und Prozeduren, Ablaufsteuerung, Weitere Funktionen und Kommandos .

1 4.1 Zahlen und numerische Ausdrücke

Numerische Werte werden per Default als Ganzzahl (integer) verarbeitet, darüberhinaus werden auch Fliesskommazahlen unterstützt. Die Notation ist dezimal, für Integer-Zahlen auch *hexadezimal* oder *binär*. Als Trennzeichen zwischen Vor- und Nachkommaanteil dient der Dezimal*punkt* (kein Dezimal*komma*). Beispiele für numerische Konstanten:

- 1234 ist eine Integerzahl in Dezimalschreibweise (default)
- 1234.0 ist eine Fliesskommazahl (der Compiler erkennt dies am Dezimal***punkt***)
- 0xABCDEF ist eine Integerzahl in hexadezimaler ("sedezimaler") Schreibweise (erkennbar am Prefix "0x")
- 0b10000001 ist eine Integerzahl in binärer Schreibweise (der Prefix "0b" bedeutet "binär").

Wird in einem numerischen Ausdruck eine Berechnung mit Fließkommazahlen (oder Fließkommavariablen) benötigt, dann sollten alle Operanden innerhalb des Ausdrucks Fließkommawerte (oder Fliesskommakonstanten) sein. Dadurch entfallen Typ-Konvertierungen (von Integer nach Float) während der Laufzeit, wodurch das Script deutlich schneller läuft (Grund: Die z.B. im MKT-View II verwendete ARM-7-CPU enthält keine Fließkommaeinheit).
Beispiel (mit Sum = Fliesskomma-Variable, und Loop = Integer-Variable):

```
Sum := Sum + 4.0 / (2 * Loop + 1);
```

Die obige Formel führt *nicht* zum gleichen Ergebnis wie die folgende:

```
Sum := Sum + 4 / (2 * Loop + 1);
```

Beachten Sie den rechten Teil der Formel im zweiten Beispiel: Dort kommen nur Integer-Werte vor. Beim Erzeugen des Bytecodes (RPN) für den Term "4 / (2 + Loop +1)" verwendet der Compiler nur Integer-Zahlen (und Integer-Operationen), da diese, wie schon erwähnt, auf den meisten Zielsystemen deutlich schneller laufen als Fließkomma-Operationen. Dazu gehört auch die DIVIDE-Anweisung (im Bytecode) : Sind beide Operanden (Zähler und Nenner) Integer-Werte, dann ist auch das Ergebnis der Division ein Integer-Wert (d.h. Ganzzahl ohne "Divisionsrest"). Sind einer oder beide Operand(en) der DIVIDE-Anweisung Fliesskommawerte, dann verwendet auch DIVIDE eine (langsame) Fliesskomma-Operation, und liefert als Ergebis einen Fliesskommawert. Benötigen Sie daher definitiv einen Fliesskommawert als Zwischenergebnis innerhalb eines Ausdrucks, verwenden Sie Fliesskommazahlen (bzw. wie im ersten Beispiel die Fliesskommakonstanten "4.0" statt der Integerkonstanten "4"). Das oben gezeigte Beispiel stammt aus dem Demo-Programm 'ScriptTest2.cvt', enthalten im Installationsarchiv, in dem die Kreiszahl PI mit Hilfe der Gregory-Leibniz-Formel berechnet wird.

Um 'Binärdaten' (z.B. empfangene CAN-Datenfelder) aus einer Sequenz von Bytes in Fliesskommazahlen zu konvertieren, verwenden Sie die Funktionen BytesToFloat, BinaryToFloat oder BytesToDouble. Beispiele für diesse Sonderfunktionen finden Sie in der Applikation 'ScriptTest1.cvt' .

Siehe auch: Datentypen 'int' vs. 'float', Numerische Funktionen, "Mathematik", Digitale Signalverarbeitung.

2 4.2 Strings

String-Konstanten werden, wie in fast allen Programmiersprachen (außer Pascal), mit Anführungszeichen (double quotes) umschlossen. Zum Einfügen von Sonderzeichen wie 'carriage return' dienen die aus "C" oder Java bekannten Backslash-Sequenzen.

Um eine Variable als String-Variable zu deklarieren, verwenden Sie das Schlüsselwort ***string***.
~~Alternativ (wie in uralten BASIC-Dialekten, für globale Variablen) verwenden Sie den "Dollar-Suffix" ($) um dem Compiler mitzuteilen, daß die Variable eine String-Variable sein soll.~~
Empfohlen wird aber, alle Variablen vor deren Verwendung mit einem eindeutigen Datentyp zu deklarieren (dies nur als Vorgriff zum Kapitel 'Variablen').
An den meisten Stellen, an denen der Compiler einen String erwartet, können Sie stattdessem auch einen *String-Ausdruck* einsetzen.

Beispiel (mit Variablendeklaration) :

```
var
   string MyString;
endvar;
...
 MyString := "This is another string";
```

Zum Standardumfang der Script-Sprache gehören einige Funktionen zur Verarbeitung von Zeichenketten, z.B. like itoa ("integer to ASCII"), hex (integer to hexadecimal ASCII), chr (wandelt einen ASCII-Wert in einen aus einem einzelnen Zeichen bestehenden String um) .

In bestimmten Fällen können auch statisch deklarierte Byte-Arrays wie Zeichenketten verwendet werden. Dabei geht allerdings die Information über die Zeichencodierung verloren. Beispiel:

```
        TP_Transmitter.buffer := "Test string sent via ISO 16765-2 'TP' .";
        TP_Transmitter.iTotalSizeInByte := strlen( TP_Transmitter.buffer );
        IsoTP_StartSending( &TP_Transmitter ); // start sending an ISO-TP
message
```

(darin ist TP_Transmitter.buffer eine als 'byte buffer[1024]' deklarierte Komponente einer Struktur, und TP_Transmitter.iTotalSizeInByte die Anzahl per ISO-TP zu sendender 'Nutzdatenbytes'.
Beim Kopieren der Zeichenkette wird ein Nullbyte zum Markieren des Endes der Zeichenkette angehängt, wenn der Platz im Ziel (hier: "Byte-Array") dafür ausreicht.)

Siehe auch: Strings mit unterschiedlichen Zeichen-Codierungen (DOS, ANSI, Unicode), Funktionen zur String-Verarbeitung.

2.1 4.2.1 Strings mit unterschiedlichen Zeichen-Codierungen (character encodings)

Die Zeichencodierung eines Strings ist variabel. Beim Datentyp 'string' werden bei der internen Ablage (im Speicher) einige Bits dazu verwendet, um die aktuelle Zeichencodierung des Strings zu definieren.
Dies funktioniert allerdings nur bei 'einzelnen' String-Variablen.
Strings in Arrays und anwenderdefinierten Datentypen (z.B. Strukturen) werden intern

grundsätzlich als UTF-8 codiert.
Wird z.B. ein String aus einer Unicode-Textdatei eingelesen (mit der Funktion file.read_line), dann enthält der String danach eine Reihe von UTF-8-codierten Zeichen. Beim Umkopieren, oder bei der Übergabe des Strings als Argument beim Aufruf von Funktionen und ähnlichen Unterprogrammen werden nicht nur die Zeichen selbst, sondern auch der Datentyp mit aktueller Zeichencodierung übergeben. Wenn nötig, kann wie im folgenden Beispiel die in einem String verwendete Zeichencodierung auch abgefragt werden :

```
select( char_encoding( MyString ) )
   case ceDOS : // der String enthält "DOS"-codierte Zeichen
(codepage 850)
       ...

   case ceANSI : // der String enthält "ANSI"-codierte Zeichen
(Windows-1252)
       ...

   case ceUnicode : // der String enthält "Unicode"-Zeichen
(codiert als UTF-8)
       ...

   case ceUnknown : // die Zeichencodierung dieses Strings ist
unbekannt
      // weil sie nirgendwo spezifiziert wurde,
      // oder weil sie keine Rolle spielt weil
      // alle Zeichencodes kleiner als 128 sind
      // (in dem Fall sind "DOS", "ANSI", und UTF-8 fast
identisch)

endselect;
```

Die Script-Sprache unterstützt zwar Unicode (genau gesagt, UTF-8-codierte Zeichenketten), dies bedeutet aber nicht dass die Firmware in der Lage ist, diese Zeichen auch auf dem Display anzuzeigen ! Die von MKT's "LCD-Treiber" verwendeten Zeichensätze (Bitmap-Fonts) stammen aus DOS-Zeiten, und enthalten daher nur Bildzeichen (Glyphen) aus dem 'DOS-Zeichensatz' (z.B. Codepage 437 oder 850) !
Bei der Ausgabe einer UTF-8-codierten Zeichenkette auf dem Bildschirm versucht die Firmware, für jedes Zeichen (in Unicode 'code point' genannt) ein möglichst gut passendes 'DOS-Zeichen' zu finden. Daher werden die meisten westlichen Zeichen (inkl. deutscher Umlaute) auch korrekt angezeigt, unabhängig von der Zeichencodierung (**c**haracter **e**ncoding : ceDOS, ceANSI, ceUnicode).

Arrays und Struktur-Komponenten mit dem Typ 'string' werden seit V0.2.6 intern *immer* als UTF-8 gespeichert. Wird z.B. eine 'DOS'- oder 'ANSI'-codierte Zeichenkette an ein Array-Element zugewiesen, werden automatisch alle Zeichen mit Codes > 127 von "DOS" oder ANSI in die entsprechenden UTF-8-Sequenzen umgewandelt. Jeder aus dem Array gelesene String hat daher die Zeichencodierung **ceUnicode** !

Wenn nötig, kann die Zeichencodierung von String-Literalen ("String-Konstanten") im Quelltext definiert werden. Dazu dienen die folgenden Kleinbuchstaben dienen als Präfix (Vorsatz):

- **a**"Text"
  "ANSI"-encoded characters (precisely: Windows CP-1252, 8 bits per character)
- **d**"Text"
  "DOS"-encoded characters (precisely: DOS Codepage 850, 8 bits per character)
- **u**"Unicode-Test"
  Unicode (precisely: characters encoded in UTF-8, with a variable number of bytes per character)

Das doppelte Anführungszeichen, mit dem der Anfang eines String-Literals im Quelltext markiert wird, muss *direkt* nach dem Präfix (a,d,u) folgen !

Beispiel (Zuweisung einer Unicode-String-Konstante an eine String-Variable) :

```
   MyString := u"Falsches Üben von Xylophonmusik quält jeden größeren Zwerg.";
```

Sofern möglich werden Sonderzeichen (wie im obigen Pangramm) vom Compiler in die gewünschte Zeichencodierung umgewandelt.
Der Compiler geht dabei davon aus, daß die Zeichen im *Script-Quelltext* als "ANSI-Zeichen" (Windows CP-1252), nicht als "DOS"-Zeichen eingegeben wurden !
Dies könnte sich in ferner Zukunft ändern, wenn der im Programmiertool integrierte Script-Editor statt des "Windows-ANSI"-Zeichensatzes auf UTF-8 umgestellt wird.
Bis dahin verwenden Sie ggf. Unicode-Escapes (nach backslash-u) für alle Zeichen, die Sie nicht direkt mit Ihrer PC-Tastatur im Editor eingeben können. Beispiel:

```
   MyString := u"Falsches \u00DCben von Xylophonmusik qu\u00E4lt jeden gr\u00F6\u00DFeren Zwerg.";
```

Weitere Beispiele finden Sie im Beispielprogramm 'String-Test' application.
Siehe auch: String (Datentyp) .

2.2 4.2.2 Verwendung von Zeichenketten und deren Format im Speicher

Die meisten Zeichen in einer Zeichenkette belegen 8 Bit (ein "Byte") im Speicher. Ein Null-Byte kennzeichnet das Ende der Zeichenkette (darum müssen Sie sich als Anwender allerdings nicht kümmern, denn der Compiler hängt automatisch ein Null-Byte am Ende einer String-Konstante an). Je nach verwendeter Zeichencodierung belegen Zeichen mit einem 'Code-Wert' über 127 (!) ebenfalls ein, bei UTF-8-codierten Strings aber auch zwei oder mehr Bytes im Speicher. 8-bit "DOS"- oder "ANSI"-Zeichen werden ohne Konvertierung im Speicher abgelegt (allerdings mit einem speziellen Flag im Datentyp, der die Codierung als "DOS"- oder "ANSI"-Zeichen enthält). Einige der 255 möglichen Zeichencodes sind für spezielle Steuerzeichen reserviert, z.B. dient das Zeichen mit dem Code 13 bei (nahezu) allen gebräuchlichen Codierungen als "carriage return" zum Markieren des Zeilenendes - siehe auch 'Backslash-Sequences' im folgenden Kapitel. Seit September 2011 können Zeichen auch als UTF-8-codierte Strings im Speicher abgelegt werden, womit dann circa 1114112 verschiedene Zeichen codiert werden können ("Unicode") . Bei UTF-8

werden pro Zeichen bis zu 4 Byte im Speicher benötigt, für 'normale' (westliche) Zeichen aber weiterhin ein Byte pro Zeichen.

Während der Script-Laufzeit wird der Speicher für Strings (und ähnliche Objekte mit 'variabler Länge') aus einem gemeinsamen Speicherpool alloziert. Die Größe des benötigten Speichers hängt (u.A.) von der Anzahl von Strings ab, die im Script gleichzeitig verwendet werden, und deren individueller Länge. Als Beispiel betrachten wir ein Array aus Strukturen, die folgendermaßen deklariert sind :

```
typedef tStringTableEntry = // user defined data type..
   struct // structure for an entry in a "string table"
      int valid;    // 0: invalid or "deleted" entry, 1:valid
      int iRefNo;   // string reference number (integer)
      string sInfo; // the string itself (any length!)
   endstruct; // end of a structure definition
endtypedef; // end of type definitions


var // declare GLOBAL variables, here: an array of structs
   tStringTableEntry StringTable[1000];
endvar; // end of variable declarations
```

Nach dem Start des Script-Programms belegt jeder Eintrag vom Typ **tStringTableEntry** nur 12 Bytes im Speicher (2 * 4 bytes für die beiden Integer-Zahlen, plus 4 Bytes für einen POINTER auf ein String-Objekt).
Erst später, wenn die Einträge im Array '**StringTable**' angefüllt werden (d.h. wenn die Strings nicht mehr "leer" sind), wird pro Zeichenkette *zusätzlicher Speicher* benötigt.
Mit anderen Worten: Der Speicherbedarf eines Scripts kann *während der Laufzeit noch ansteigen*.
Darum sollten Sie zumindest grob abschätzen, wieviel zusätzlicher Speicherplatz von Ihrer Applikation schlimmstenfalls für Zeichenketten benötigt werden könnte (der Compiler kann dies nicht erkennen; denn Strings sind dynamische Objekte, die nahezu beliebig lang werden könnten).
Lassen Sie das Programm dazu ausreichend lange im Simulator laufen, und beobachten die maximale Speicherauslastung im Debugger.

Ähnlich wie in der Programmiersprache 'C' kann auf *einzelne* Zeichen innerhalb eines Strings zugegriffen werden, wie dies bei einem Array der Fall wäre.
Beispiel:

```
var
  string s1;
  int i;
endvar;
s1 := "This is a test";
i  := s1[0];   // note that indices start counting at ZERO
```

Da diese Art des Zugriffs auf einzelne Zeichen nicht bei UTF-8 codierten 'Unicode-Strings' funktioniert, empfiehlt sich stattdessem aber die Funktion substr().

Siehe auch: Funktionen zur String-Verarbeitung, .

2.3 4.2.3 String-Konstanten mit Sonderzeichen

Der Compiler kann nicht wissen, für welchen "Zweck" ein String später verwendet wird. Er kennt weder die verwendete Sprache noch den Zeichensatz, mit dem der String eventuell auf dem Display angezeigt wird. Aus dem Grund versucht der Compiler auch nicht, irgendwelche 'Sonderzeichen' (z.B. deutsche Umlaute) im String an dem im Display *meistens* verwendeten DOS-kompatiblen Zeichensatz anzupassen. Ist Ihnen (als Autor eines Scripts) bekannt, daß der String später mit einem DOS-kompatiblen Zeichensatz angezeigt werden wird, können Sie z.B. die Umlaute im Script-Quelltext durch entsprechende hexadezimale Äquivalente ersetzen. Eine Alternative wäre der Einsatz von Unicode (siehe vorheriges Kapitel).
Werden Sonderzeichen im String z.B. als hexadezimale Sequenzen für einen "DOS-Zeichensatz" codiert, dann sollte der String mit dem Prefix 'd' ("DOS") als solcher gekennzeichnet werden. Die String-Runtime-Library kann dann, wenn z.B. ein "DOS-String" an einen "ANSI-String" oder einen "Unicode-String" angefügt werden soll, die Zeichencodes entsprechend konvertieren.

Beispiele:

```
Test := d"\x99rtliche Bet\x84ubung \x9Abelkeit"; // string with hex codes for Ö, ä, Ü if a 'DOS font' is used for rendering
Test := d"Örtliche Betäubung kann Übelkeit hervorrufen"; // string converted into 'DOS characters' by the compiler
```

Hexadezimale Codes für bestimmte "DOS"-Sonderzeichen, die von fast allen programmierbaren MKT-Geräten unterstützt werden :

Hex. Code	Zeichen (hier: ANSI)	Name
84	ä	a diaeresis
94	ö	o diaeresis
81	ü	u diaeresis
8E	Ä	A diaeresis
99	Ö	O diaeresis
9A	Ü	U diaeresis
DF	ß	German sharp s
.		
.		

Eine komplette Tabelle mit einem 'DOS'-Zeichensatz ("Codepage 437") finden Sie hier . Zeilen und Spalten sind hexadezimal numeriert, wodurch der entsprechende 8-bit-Code (bestehend aus zwei Hex-Ziffern) sehr einfach zusammengestellt werden kann. Das Beispielprogramm Text Screen verwendet viele dieser speziellen 'DOS-Zeichen' um auf dem Bildschirm Linien und Rechtecke zu zeichnen.
Siehe auch: Unicode- und ANSI-Strings.

2.4 4.2.4 Strings mit Backslash-Sequenzen

Neben der bereits erwähnten Sequenz **\x** (hexadezimaler Code für ein Sonderzeichen) haben die folgenden ***Backslash-Sequenzen*** eine besondere Bedeutung im *Script-Quelltext* (die z.T. von der Bedeutung von Backslash-Sequenzen im Display-Interpreter abweichen):

****

Fügt einen *einzelnen* Backslash in die Zeichenkette ein .

\r

Fügt ein Carriage-Return-Zeichen ein (aka CR, chr(13)) .

\n

Fügt ein New-Line-Zeichen ein (aka "Linefeed", chr(10)) .

\x

Hexadezimaler Code für ein 8-Bit-Sonderzeichen (kein Unicode!).
Siehe Details im Kapitel 'Strings mit Sonderzeichen' .

\u

Fügt ein nahezu beliebiges Unicode-"Zeichen" (code point), definiert als ***mindestens 4-ziffrige Hexadezimalzahl***, in den String ein.
Der Compiler(!) ersetzt dieses "Unicode-Zeichen" durch eine entsprechende UTF-8 - Sequenz in der Zeichenkette.
Beachten Sie, daß nur sehr wenige der 1114112 theoretisch möglichen 'Code Points' auch auf dem Display angezeigt werden können !
Details zum Thema Unicode finden Sie im Kapitel 'Strings mit verschiedenen Zeichen-Kodierungen' .

\"

Fügt das doppelte Anführungszeichen (double quote character), ohne Backslash, in den String ein. Dies ist nötig, weil das doppelte Anführungszeichen (ohne Backslash) das Ende des Strings markieren würde.

Die oben aufgeführten Backslash-Sequenzen in der Script-Sprache sind nicht mit den Backslash-Sequenzen im Format-String eines UPT-Anzeige-Elements zu verwechseln !
Lediglich die 'einfachen' Steuerzeichen (carriage return, new line, etc) haben die gleiche Bedeutung; die interne Funktion ist aber völlig inkompatibel !

Siehe auch: Aufruf von Script-Funktionen aus Backslash-Sequenzen im Format-String eines UPT-Anzeige-Elements

2.5 4.2.5 Funktionen zur String-Verarbeitung

Strings können mit dem Operator '+' aneinander gefügt werden (formale "Addition"). Beispiel:

```
var
  string Info; // Deklaration einer String-Variablen
endvar;
Info := "First part";
Info := Info + " second part";
```

Die folgenden Funktionen zur String-Verarbeitung wurden bislang in der Script-Sprache implementiert:

append(<destination>, <source> [, <index_variable>])

Hängt einen String (oder auch andere Daten) an das Ende eines anderen Strings, oder eines Byte-Arrays an.
Die Index-Variable ist dabei optional (siehe Beispiel 3).

Beispiel 1: Anhängen eines Strings an einen String

```
var
  string s1,s2; // Deklaration von zwei String-Variablen
endvar;
s1 := "Verwechseln Sie nicht Äpfel";
s2 := " und Birnen";
append(s1,s2);  // Anhängen von  s2 an s1, Ergebnis in s1
print( s1 );    // Ergebnis auf Text-Panel anzeigen
```

Der dritte Funktionsparameter ('index') wird in diesem Fall nicht benötigt.
Das Anhängen von Zeichenketten per append() ist schneller als die 'Addition' von Strings (z.B. **append(s1,"Hallo")** statt **s1 := s1+"Hallo"**), weil append() keinen String-Speicher freigeben und erneut allozieren muss wenn in der Zielvariablen (im Beispiel 's1') noch genügend Platz für weitere Zeichen vorhanden ist. Aufgrund der internen Speicherverwaltung (Allozierung in Blöcken von N mal 64 Bytes) ist dies oftmals der Fall, d.h. beim Anfügen von weniger als 64 Zeichen ist keine aufwändige Allozierung nötig.

Beispiel 2: Anhängen mehrerer Strings an einen "binären Datenblock" (Byte-Array)
mit expliziter Angabe einer Index-Variablen.

```
var
  byte TxBuffer[1024]; // Deklaration eines Byte-Arrays
  int  TxByteIndex;    // Index-Variable für 'TxBuffer'
endvar;

TxByteIndex := 0;  // begin filling TxBuffer[0] here
append( TxBuffer, "First string.\r\n", TxByteIndex );
// Note: append() will increment TxByteIndex by the
//       NUMBER OF BYTES appended to the buffer !
TxBuffer[TxByteIndex++] := 0x00;   // append a ZERO BYTE as string-
end-marker
append( TxBuffer, "Second string.\r\n", TxByteIndex );
TxBuffer[TxByteIndex++] := 0x00;   // append another ZERO BYTE
append( TxBuffer, "Third string.\r\n", TxByteIndex );
```

```
        StartSendingBlock( TxBuffer, TxByteIndex/*nBytes*/ ); // user-
defined procedure
```

Der dritte, optionale Funktionsparameter ('index_variable') ist in diesem Fall eine Integer-Variable, deren Wert jeweils um die Anzahl angehängter Bytes erhöht wird. Im obigen Beipiel wird das Array 'TxBuffer' mit mehreren Zeichenketten gefüllt, die jeweils durch ein Null-Byte (ähnlich wie in der Programmiersprache 'C') voneinander getrennt werden.
Da ein Null-Byte das Ende eines Strings markiert, kann es selbst nicht in einer String-Variablen enthalten sein. Im obigen Beispiel wird das Null-Byte daher mit der Anweisung

```
TxBuffer[TxByteIndex++] := 0x00;
```

an den 'Sendepuffer' (Variable 'TxBuffer', deklariert als Array mit maximal 1024 Byte) angehängt. Der Post-Inkrement-Operator '++' erhöht den Wert der Variablen 'TxByteIndex' *nach* dem Zugriff.
Der entsprechende Code wäre ohne '++'-Operator etwas länger und langsamer:

```
TxBuffer[TxByteIndex] := 0x00;
TxByteIndex := TxByteIndex + 1;
```

Beispiel 3: Anhängen von Byte-Werten an einen "binären Datenblock" (Byte-Array) *ohne* Index-Variable.

```
var
  byte TxBuffer[1024]; // declaration of a fixed-size byte-array
endvar;

    .... (other code that may have used TxBuffer) ....

TxBuffer.len := 0;     // begin filling TxBuffer at the first
element
append(TxBuffer, (byte)0x02);     // ASCII control character 'STX'
// Note: append() will increment TxBuffer.len by the
//       NUMBER OF ARRAY ELEMENTS appended to the buffer !
append(TxBuffer, "First string.\r\n");
append(TxBuffer, "Second string.\r\n");
append(TxBuffer, (byte)0x03);     // ASCII control character 'ETX'
file.write(hSerialPort,TxBuffer); // send via serial port
```

Im obigen Beipiel wird statt einer Index-Variablen die *Länge* des Arrays (hier: TxBuffer.len) bei jedem Aufruf von append() erhöht - allerdings nur bis zur maximale Größe (.size). Da es sich um ein Byte-Array handelt, kann nach den oben gezeigten Anweisungen die Länge *in Bytes* aus TxBuffer.len ausgelesen werden.

chr(N)

Wandelt einen Integerwert (N, 0..255, oftmals der Code eines "DOS"-Zeichens) in einen aus einem einzelnen Zeichen bestehenden String um.
Liegt der Wert von 'N' über 255, vermutet(!) die Runtime-Funktion dass es sich um ein UNICODE-Zeichen handeln soll. Verwenden Sie für Unicode aber besser die Funktion unicode_chr(N), denn nur so ist sichergestellt, dass auch Zeichencodes zwischen 128 und 255 korrekt interpretiert werden.

Beispiel: chr(32) liefert das Leerzeichen (Space, ASCII # 32).
Für ANSI-Zeichencodes verwenden Sie statt 'chr' die Funktion ansi_chr.

ansi_chr(N)

Wandelt einen Integerwert (N, 0..255, ANSI-Code eines Zeichens) in einen aus einem einzelnen Zeichen bestehenden String um.
Liegt der Wert von 'N' über 255, vermutet(!) die Runtime-Funktion dass es sich um ein UNICODE-Zeichen handeln soll.
Beispiel: ansi_chr(176) liefert das 'Grad'-Zeichen (°), z.B. für '°C' = 'Grad Celsius'.

unicode_chr(N)

Ähnlich wie chr(N), in diesem Fall wird N allerdings immer als Unicode aufgefasst (N = "codepoint", 0..0x10FFFF), und in einen UTF-8-codierten String umgewandelt, der zwar (wie bei chr) aus nur einem Zeichen, aber möglicherweise bis zu 6 (!) Bytes bestehen könnte. Diese Konvertierung ist immer möglich, **unabhängig davon ob das Zeichen später auf dem "Bildschirm" angezeigt werden kann oder nicht !** Beispiel: unicode_chr(0x20AC) liefert einen UTF-8-codierten String mit dem 'Euro-Zeichen'.

CharAt(string s, int char_index)

Liefert *den Code* des N-ten Zeichens (N=char_index) aus der Zeichenkette 's'. Wie in den meisten Programmiersprachen üblich, beginnt die Index-Zählung bei *Null* für das *erste* Zeichen in der Zeichenkette. Das Ergebnis ist ein **32-bit Unicode - Wert** ("codepoint number"), der für 'normale' westliche Zeichen mit dem ASCII-Wert (Zeichencodes 1 bis 127) identisch ist. Mit anderen Worten: CharAt funktioniert sowohl mit "ASCII-Strings" als auch mit "Unicode". Dies funktioniert nur, weil 'CharAt' die Zeichencodierung des Strings (s) beachtet, und auch mit UTF-8-codierten Strings klarkommt. Dazu zählt auch der wichtige Unterschied zwischen 'Zeichen-Index' und 'Byte-Index' (CharAt verwendet den Zeichen-Index, nicht den Byte-Index, und liefert nicht nur das N-te BYTE, sondern das N-te ZEICHEN, wenn das Zeichen aus mehreren Bytes besteht.
Ist <char_index> negativ, oder überschreitet die Länge des Strings, liefert CharAt den Integer-Wert 0 (Null).

char_encoding(string s)

Liefert den Zeichen-Codierungs-Typ des angegebenen Strings. Das Ergebnis ist eine der folgenden Konstanten:
ceDOS ("DOS"-codierte Zeichen), ceANSI ("ANSI" aka "Windows CP-1252"), ceUnicode, oder ceUnknown.

ftoa(float value, int nDigitsBeforeDot, int nDigitsAfterDot)

"floating-to-ascii" .
Wandelt eine Fliesskommazahl (1. Argument, 'value') in einen **dezimalen String** um, wobei die spezifizierte Anzahl Ziffern *vor* und *nach* dem Dezimalpunkt verwendet wird. Beispiel:

```
   Info := "Reifendruck = "+ftoa(fltTirePressure, 4, 1)+"
bar";
```

Ist die 'Anzahl Ziffern **vor** dem Dezimalpunkt' (zweiter Parameter) höher als nötig, dann werden entsprechend viele *Leerzeichen* (keine Nullen) vor der Zahl eingefügt. Ist die 'Anzahl Ziffern **nach** dem Dezimalpunkt' (dritter Parameter) höher als nötig, dann werden nach dem

Nachkomma-Anteil weitere *Nullen* (keine Leerzeichen) angehängt. Ist die 'Anzahl Ziffern **nach** dem Dezimalpunkt' Null, dann wird auch der Dezimalpunkt weggelassen.
Hinweis: Im Gegensatz zu 'ftoa' (welches ggf. Leerzeichen voranstellt) verwendet die Funktion itoa, wenn eine feste Anzahl Ziffern erzeugt werden soll, *führende Nullen* !
Inverse Funktion: atof() alias ParseFloat().

`itoa( int value [, number of digits])`

"integer-to-ascii" .
Wandelt einen Integer-Wert (erstes Argument) in einen **dezimalen String** um, wobei optional die spezifizierte Anzahl Ziffern (zweites Argument) erzeugt wird. Im Gegensatz zu ftoa erzeugt 'itoa' dabei *führende Nullen*. Beispiel:

```
Info := "Timestamp="+itoa(system.timestamp,8);
```

Ist die Anzahl zu erzeugender Ziffern spezifiziert, der Wert (value) aber zu hoch, werden nur die *niederwertigen(!) Ziffern* emittiert. Beispiel:

`itoa(1234,2)` liefert 24 (als String), nicht "1234" !

Fehlt die Angabe der zu erzeugenden Ziffern, oder ist max_digits Null, produziert itoa nur soviel Ziffern wie nötig (ohne führende Nullen).
Inverse Funktion: atoi() alias ParseInteger().

`atoi( string value [, int max_digits [, int start_index]] ),`
`atof( string value [, int max_digits [, int start_index]] )`

"ascii-to-integer" bzw "ascii-to-float", alias ParseInteger(), ParseFloat() .
Konvertiert einen **dezimalen String** in einen Integer- bzw Fliesskomma-Wert (numerisch), wobei optional nur eine bestimmte Anzahl Ziffern ausgewertet wird (zweites Argument), oder der gesamte String.
Als optionales drittes Argument kann noch die Stelle (Zeichen-Index) angegeben werden, an dem die Konvertierung mit der höchstwertigen Ziffer beginnen soll.
Das optionale Argument ('start_index') kann ein Integerwert, oder die Adresse einer Integer-Variablen sein. Bei Übergabe *der Addresse* einer Integer-Variablen wird deren Wert um die Anzahl geparster Zeichen erhöht (siehe viertes Beispiel).
Beispiele:

`atoi("1234567");` liefert den Integer-Wert 1234567
`atoi("1234567",3);` liefert den Wert 123 (wegen Begrenzung auf 3 Ziffern).
`atoi("1234567",3,2);` liefert den Wert 345 (3 Ziffern, Start bei Index 2 d.h. dem *dritten* Zeichen im String).
`atoi("1234567",10, &start_index);` Variable start_index wird um die Anzahl geparster Zeichen erhöht.

Wie üblich, beginnt auch hier die Index-Zählung bei NULL. Das erste Zeichen in einer Zeichenkette befindet sich an Index 'Null' (nicht Index 'Eins').
Die Funktionen atoi bzw atof erkennen ein führendes 'Minus'-Zeichen als Vorzeichen.

`hex( int value, int number_of_digits )` alias **`HexString( .. )`**

Wandelt eine Integerzahl (1. Argument) in einen **hexadezimalen** String, mit der spezifizierten Anzahl Ziffern (2. Argument).
Inverse Funktion: ParseHexString().

`BinaryString( int value, int number_of_digits )`

Wandelt eine Integerzahl (1. Argument) in einen **binären** String, mit der spezifizierten Anzahl Ziffern (2. Argument).
Beispiel:
`BinaryString( 0x12345678, 32 )` liefert 00010010000110100010101100111000 *als String*.
Inverse Funktion: ParseBinaryString().

`strlen( string s )`

Ermittelt die *Anzahl von Zeichen* in der Zeichenkette (nicht die 'Länge des Strings in BYTES'; besonders wichtig für UTF-8-codierte Zeichenketten).
Beispiel:
`strlen("How long is this string ?")` liefert den Wert 25 .

`strpos( string haystack, string needle[, int start_index] ),`
`stripos( string haystack, string needle[, int start_index] ),`
`strrpos( string haystack, string needle[, int start_index] ),`
`strripos( string haystack, string needle[, int start_index] ),`
`strpos2( string haystack, string needle[, int start_index] )`

Diese Funktionen suchen nach dem ersten bzw letzten Vorkommen des Suchstrings (needle, "Nadel") in einem größeren String (haystack, "Heuhaufen"), beginnend bei Zeichen-Index Null(!) oder dem optional spezifizierten Start-Index (Das erste Element in einer Array-ähnlichen Struktur hat, wie immer, den Index Null, nicht den Index Eins).

Die Funktionen strpos/strrpos unterscheiden zwischen Groß- und Kleinbuchstaben, die Funktionen stripos/strripos nicht.
Die Funktionen strpos/stripos suchen das *erste* Vorkommen des Strings, die Funktionen strrpos/strripos das *letzte*.

Der Rückgabewert ist der **Zeichen-Index** des ersten Zeichens, unter dem die "Nadel" im "Heuhaufen" gefunden wurde, oder -falls die Suche nach der Nadel im Heuhaufen erfolglos war- ein negativer Integerwert.

Im Gegensatz zu strpos() liefert strpos**2**() als Rückgabewert nicht den Index des *ersten Zeichens* der Nadel im Heuhaufen, sondern den Index des nächsten Zeichens, welches ***nach*** der gefundenen Nadel folgt. Mit anderen Worten wird die "Nadel" durch 'strpos2' überlesen. Ein Parser kann mit dem Index z.B. den *danach folgenden Rest* analysieren.

Fehlt das dritte Funktionsargument (start_index), beginnt die Suche beim ersten Zeichen im Hauhaufen (d.h. Index Null für strpos+stripos). Damit kann z.B. ein einfacher String-Parser realisiert werden. Beispiel:

```
 haystack := "This test string is 38 characters long";
 needle   := "is"; // string to be found in the haystack
 i := strpos(haystack,needle);     // find the first needle
(result: i=2)
 i := strpos(haystack,needle,i+1); // find the next needle
```

```
(result: i=17)
```

Um danach den Wert nach dem per strpos() gefundenen Schlüsselwort nach integer oder float zu konvertieren, verwenden Sie atoi (ascii-to-integer) bzw. atof (ascii-to-float):

```
 i := strpos2(haystack,"VBat=");   // find index of the next
char AFTER the needle
 if( i>0 ) then // found the needle (keyword) in the haystack,
parse the following value
    iValue := atoi( haystack, 5/*digits*/, i/*start*/ ); //
parse number after keyword
 endif;
```

Da auch die Funktionen atoi und atof den *Index des ersten Zeichens* ('start index') als optionalen dritten Parameter unterstützen, kann wie im obigen Beispiel das zeitraubende Umkopieren von Zeichenketten im Speicher vermieden werden.

substr(string s, int start [, int length])

Liefert einen Ausschnitt (sub-string) aus s, beginnend mit dem Zeichen am angegebenen Start-Index (Index-Zählung beginnt bei Null), mit einer maximalen Länge von <length> Zeichen (dritter, optionaler, Parameter).
Fehlt die Angabe von 'length', enthält der zurückgegebene String alle Zeichen ab Index 'start' bis zum Ende der Zeichenkette 's'.
Ist der Parameter 'length' spezifiziert und *positiv*, wird das Ergebnis niemals mehr als <length> Zeichen enthalten (es könnte aber, je nach Startindex und Länge des Quellstrings, durchaus *weniger* Zeichen enthalten).
Ist der Parameter 'start' spezifiziert aber *negativ*, wird als Ergebnis ein leerer String zurückgeliefert.

ParseInteger(string value [, int max_digits [, int start_index]])

Alias für die "C"-Funktion atoi() ("ASCII to integer").
Bedeutung des optionalen Parameters 'start_index' wie bei atoi().
Fehlt die Angabe von 'start_index', beginnt der Parser beim ersten Zeichen im String (Index Null).
Fehlt auch die Angabe von 'max_digits', ist die Anzahl geparster Ziffern nur durch die Länge des Strings (value) begrenzt.
Wird der optionalen Parameters 'start_index' *als Adresse* einer Integer-Variablen (Pointer, Referenz) übergeben, so erhält diese Variable nach dem Aufruf den nullbasierten Index des nächsten Zeichens nach dem geparsten Wert.
Dies gilt auch für die weiter unten aufgeführten String-Parser-Funktionen. Beispiel: Siehe ParseFloat().

ParseFloat(string value [, int max_digits [, int start_index]])

Alias für die "C"-Funktion atof() ("ASCII to float").
Bedeutung des optionalen Parameters 'start_index' wie bei atoi().
Dies gilt auch für die weiter unten aufgeführten String-Parser-Funktionen. Beispiel:

```
  sLine := "U=23.4V T=22.7°C";
  start_index := strpos( sLine, "U=" );
  Ubat := ParseFloat( sLine, 10/*max_digits*/,
```

```
start_index+2 );
  start_index := strpos( sLine, "T=" );
  Temp := ParseFloat( sLine, 10/*max_digits*/,
start_index+2 );
```

ParseHexString(string value [, int max_digits [, int start_index]])

Umkehrfunktion zu HexString. Bedeutung des optionalen Parameters 'start_index' wie bei atoi().
Beginnt der String (value) mit der Zeichenkette 0x (Präfix für 'Hexadezimal' wie in "C" oder Python), wird dieser automatisch überlesen. In der Zählung von 'max_digits' sind auch die zwei optionalen Buchstaben für den Präfix enthalten.
Der Parameter max_digits begrenzt die Anzahl zu parsender Zeichen. Fehlt diese Angabe, oder ist max_digits Null, dann endet der Parser beim ersten Zeichen welches keine gültige Hex-Ziffer sein kann.
Beispiele:

```
  iValue := ParseHexString ( "0x1234" ); // returns 0x1234 as
integer (decimal 4660)
  iValue := ParseHexString("0x1234", 0, 4 ); // returns 0x34
as integer (decimal 52)
  iValue := ParseHexString("0x1234", 4, 1 ); // returns 3 as
integer (max_digits=1)
```

ToDo: Beschreiben, warum (und wofür) start_index auch als 'Call-by-Reference' übergeben werden kann.

ParseBinaryString(string value [, int max_digits [, int start_index]])

Umkehrfunktion zu BinaryString. Bedeutung des optionalen Parameters 'start_index' wie bei atoi().
Beginnt der String (value) mit der Zeichenkette 0b (Präfix für 'Binär' wie in Python), wird dieser automatisch überlesen. In der Zählung von 'max_digits' sind auch die zwei optionalen Buchstaben für den Präfix enthalten.

string(source_value [, number of characters])

Diese String-erzeugende Funktion konvertiert den als erstes Argument übergebenen Wert (source_value) in eine Zeichenkette, mit optionaler Begrenzung der Anzahl erzeugter Zeichen.
Ist der Wert ein Byte-Array (oder Pointer auf Bytes), wird er als Kette UTF-8-codierter Zeichen aufgefasst. Beispiel:

```
  ParseReceivedString( string(pbPayload,iPayloadLength) );
```

string(source_array, iFirstByte, iMaxBytes)

Diese String-erzeugende Funktion konvertiert den Inhalt eines Byte-Arrays (erstes Argument), beginnend an einem nullbasierten Array-Index (zweites Argument), mit maximal N Bytes (drittes Argument), in eine Zeichenkette (string).
Der Inhalt des Byte-Arrays wird auch hier als eine Kette UTF-8-codierter Zeichen aufgefasst. Beispiel:

```
  s := string( b256ReceivedData, iBeginOfPayload,
iPayloadLength ); // inefficient call-by-value
```

Wie in Kapitel 4 beschrieben, sollten große Strukturen und Arrays bei der Parameterübergabe besser als Pointer übergeben (d.h. 'call by reference' statt 'call by value'). So kann auch hier das unnötige Kopieren des Arrays vermieden werden:

```
  s := string( &b256ReceivedData, iBeginOfPayload,
iPayloadLength ); // more efficient call-by-reference
```

Weitere Beispiele und Tests zur Verarbeitung von Zeichenketten in der Script-Sprache finden Sie in der Applikation 'String-Test' .

< Fortsetzung folgt ... >

Siehe auch : Schlüsselwörter , Datei-Ein/Ausgabe, Inhaltsübersicht .

3 4.3 Konstanten

3.1 4.3.1 Fest eingebaute Konstanten

Einige Konstanten sind fest in der Symboltabelle des Compilers enthalten. Der numerische Wert dieser Konstanten sollte Sie (als Anwender) nicht interessieren, darum sind die Werte in der folgenden Tabelle auch nicht aufgeführt. Einige dieser Konstanten sind versions- oder hardware-abhängig.

Im Interesse der Unterscheidbarkeit dieser Konstanten von Schlüsselwörtern und Variablennamen beginnen viele der fest eingebauten Konstanten mit einem kleinen 'c' (constant). Manche Konstanten beginnen mit dem Prefix 'cl' (=color, Farbe), 'ce' (=character encoding, Zeichencodierung), oder 'key' (=Tastencode).

Name der Konstanten	Beschreibung
__LINE__	Liefert die aktuelle Zeilennummer während des Compilierens, ähnlich wie in "C". Beispiel: trace.print("Problem in Zeile ", **__LINE__** , " :\r\n");
arLeftScale,arRightScale, arTopScale,arBottomScale, arCurve1, arCurve2	'area codes', verwendet in Touchscreen-Event-Handlern für Diagramme und als Array-Index beim Zugriff auf display.dia.scale[ar..Scale].
ceDOS	Character encoding (Zeichencodierung) für 'DOS', genauer: DOS 'Codepage 850' . Historisch bedingt ist dies die Zeichencodierung in den 'eingebauten' Bitmap-Fonts.
ceANSI	Character encoding (Zeichencodierung) 'ANSI', genauer: Windows 'CP-1252' .
ceUnicode	Zeichencodierung für Unicode-Strings.
ceUnknown	Dummy für unbekannte Zeichencodierung, bzw. für Strings ohne Sonderzeichen.
clBlack	Schwarze 'Farbe' (bitte keine Vermutung anstellen, ob diese Konstante den Wert "Null" hat. Farbwerte sind im Zweifelsfall hardwarespezifisch (4, 8, 16, oder 24 Bit pro Farbe) ! Wie die unten folgenden Symbole dient diese Konstante als Parameter für setcolor. *Mischfarben* können gegebenenfalls mit der Funktion rgb erzeugt werden.
clWhite	Hellweiß. Dies ist (neben Schwarz) die einzige Standard-"Farbe", die *immer* verfügbar ist - auch bei alten Geräten mit Monochrom-Display.
clBlue	pures, gesättigtes Blau
clGreen	pures, gesättigtes Grün
clRed	pures, gesättigtes Rot
clLtBlue	Hellblau
clLtGreen	Hellgrün
clLtRed	Hellrot
clCyan	Cyan (Mischung aus Blau + Grün)
clMagenta	alias 'Purpur', 'Fuchsia', z.T. eher 'Violett', manchmal (fälschlicherweise) auch "Pink" genannt. Ist dies nicht der gewünschte Farbton, verwenden Sie die Funktion rgb um die Farbe als Mischung aus Rot, Grün und Blau zu definieren.
clYellow	Gelb
clOrange	Orange

lBrown	Braun
lDkGray	Dunkelgrau
lLtGray	Hellgrau
lTransparent	Dummy-Farbwert. Als Hintergrundfarbe bei bestimmten Anzeige-Elementen verwendbar.
RedrawAll (etc)	"Alles neu zeichnen". Wird z.B. beim erzwungenen Neu-Zeichnen von Tabellen verwendet.
sOff	Cursor Shape "Off" (Text-Cursor unsichtbar)
sUnderscore	Cursor Shape "Underscore" (Text-Cursor = Tiefstrich)
sSolidBlock	Cursor Shape "Solid Block" (Text-Cursor = gefülltes Rechteck)
sBlinking	Cursor Style "Blinking" (blinkender Text-Cursor)
CanRTR	RTR-flag (Remote Transmission Request) für den CAN-Bus. Kann bitweise mit der Länge des angeforderten Datenfelds kombiniert werden.
CANStatus...	CAN Status Flags. Details: Siehe CAN.status
CanIdBit_Bus2	Bitmaske (Bit 30) zum Codieren der CAN-Bus-Nummer als Teil von tCANmsg.id . Ist dieses Bit gesetzt, wurde die Message von der zweiten Schnittstelle empfangen, bzw soll au der zweiten CAN-Schnittstelle gesendet werden (wenn nicht auch cCanIdBit_Bus3 gesetzt ist) --- In der Tat bilden cCanIdBit_Bus2 (Bit 30) und cCanIdBit_Bus3 (Bit 31) eine **Zwei**-Bit-Zahl mit der bis zu **vier** verschiedene CAN-Busse unterschieden werden könner cCanIdBit_Bus2 gelöscht, cCanIdBit_Bus3 gelöscht: 1. CAN-Interface ("CAN1"). cCanIdBit_Bus2 gesetzt , cCanIdBit_Bus3 gelöscht: 2. CAN-Interface ("CAN2"). cCanIdBit_Bus2 gelöscht, cCanIdBit_Bus3 gesetzt: 3. CAN-Interface (z.B. CAN-via-UDP) cCanIdBit_Bus2 gesetzt , cCanIdBit_Bus3 gesetzt: 4. CAN-Interface oder LIN-Bus. Ein Beispiel zur Verwendung dieser Bits finden Sie im "CAN-Gateway"-Demo (CANgate1.cv in der CAN-Telegramme von einem Bus zum anderen 'durchgereicht' werden.
CanIdBit_Bus3	Bitmaske (Bit 31) zum Codieren der Bus-Nummer in tCANmsg.id - siehe cCanIdBit_Bus2.
CanIdBit_LIN	Bitmaske (Bit 31+30) um eine CAN-"Message" auch für den LIN-Bus nutzen zu können.
CanIdBit_Extd	Bitmaske (Bit 29) zur Unterscheidung von "Standard"- oder "Extended"-ID in tCANmsg.id. Ist cCanIdBit_Extd gelöscht, enthalten die unteren 11 Bits im Identifier-Feld den "Standard"-I Ist cCanIdBit_Extd gesetzt, enthalten die unteren 29 Bits im Identifier-Feld einen "Extended"-
CanTx_Normal, CanTx_NoWait	Optionen für das CAN-Senden per can_transmit.
FileAttrNormal	Keine gesetzten Dateiattributs (0) : "Normale, schreib- und lesbare Datei, die nicht archiviert werden soll. Wird beim Lesen von Verzeichnissen in tDirEntry.attributes verwendet.
FileAttrRdOnly	Read only attribute (1). "Diese Datei darf gelesen, aber nicht geschrieben werden."
FileAttrHidden	Hidden file (2). Versteckte Datei, in FAT-Dateisystemen (DOS, Windows).
FileAttrSystem	System file (4). "Finger weg"... zumindest in FAT-Dateisystemen.
FileAttrLabel	Label (8). Keine echte Datei sondern der Name des Datenträgers (bei FAT-Dateisystemen)
FileAttrDir	Directory (16). Verzeichnis oder Unterverzeichnis bei FAT-Dateisystemem.

FileAttrArch	"Archivieren" (32). Dieses Dateiattribut wird unter DOS und Windows gesetzt, wenn eine Datei seit der letzten "Archivierung" (Backup) geändert wurde, und daher beim nächsten Backup erneut gesichert werden sollte. Eine "normale" Datei hat daher in vielen Fällen nicht das Attribut Null (=cFileAttrNormal) sondern 32 (=cFileAttrArch) !
FileAttrDevice	Kein Speichermedium sondern ein 'Gerät' (serieller Port, etc)
FirmwareCompDate	Kompilationsdatum der Firmware als String, z.B. "Aug 25 2010" .
PI	Die Zahl "PI" (ungefähr 3.14159265358979323846264 3)
TimestampFrequency	systemspezifische Taktfrequenz des timestamp-Generators, gemessen in Hz ("Ticks / Sekunde
tUnknown	Datentypen-Code für 'unbekannten Datentyp'. Siehe Hinweise zu typeof()
tFloat	Datentypen-Code für 'floating point' (32 bit, einfache Genauigkeit)
tDouble	Datentypen-Code für 'double' (64 bit, doppelte Genauigkeit)
tInteger	Datentypen-Code für 'integer'
tString	Datentypen-Code für 'string'
tByte	Datentypen-Code für ein einzelnes 'Byte' (8 bit unsigned)
tWord	Datentypen-Code für einen 16-Bit-Wert ohne Vorzeichen ("WORD")
tDWord	Datentypen-Code für '32 Bit ohne Vorzeichen' aka 'Doubleword'. Wird oft beim Zugriff auf Objekte im CANopen-OD per SDO verwendet.
tColor	Datentypen-Code für eine Farbe (hardware-abhängiges Farbmodell)
tChar	Datentypen-Code für ein einzelnes Zeichen ('character')
tError	Datentypen-Code für einen 'Fehlercode', wird z.B. von cop.sdo als Return-Wert verwendet.
TRUE	boolsches 'Wahr', bzw. 1 (Eins) als Integer-Wert
FALSE	boolsches 'Unwahr', bzw. 0 (Null) als Integer-Wert
keyEnter, keyEscape, .. keyF1, keyF2, keyF3, .. keyLeft, keyRight, keyUp, keyDown	Tastencodes für die Funktion getkey, und für einigen Low-Level Event-Handlern. Eine Übersicht der symbolischen Tastencodes finden Sie in der Beschreibung von getkey.
kmEnter, kmEscape,..	Bitmasken für die Tastaturmatrix. Werden von der Funktion system.dwKeyMatrix, und dem Event-Handler OnKeyMatrixChange verwendet.
NULL	Wert für einen ungültigen Pointer bzw. ungültige Adresse.
O_RDONLY, ...	Flags zum Öffnen von Dateien. Verwendung in der Funktion file.open .
ofOpen	Flag zum Zeichnen eines *offenen* Polygons, z.B. bei display.dia.poly.draw
ofClosed	Flag zum Zeichnen eines *geschlossenen* Polygons, z.B. bei display.dia.poly.draw

ɔfFilled	Reserviert als Flag zum Zeichnen eines *gefüllten* Polygons
ɔfNoScroll	Flag zum Zeichnen eines Polygons, welches *nicht* zusammen mit den Kurven gescrollt wird. Wird bislang nur in der Funktion display.dia.poly.draw verwendet.
fVectorASC, ...	'String-Formate', z.B. bei Umwandlung von tCANmsg nach string.
mOff, smCell, smRow, ..	Selektions-Modus. Details in der Beschreibung des Display-Elements 'Tabelle'.
vmXYZ	'widget messages' or, sometimes, 'windows message'. Verwendung in Event-Handlern. Das Script kann damit bestimmte Touchscreen- und ähnliche Ereignisse abfangen.

3.2 4.3.2 Anwenderdefinierte Konstanten

Zusätzlich zu den fest (im Script-Compiler) eingebauten Konstanten können auch eigene Konstanten im Script definiert werden.
Dadurch kann in vielen Fällen die Lesbarkeit des Quelltextes erhöht werden, speziell wenn der Wert einer Konstanten an mehreren Stellen im Script benötigt wird, oder wenn sich der Zweck einer dezimalen (numerischen) Konstante nicht direkt aus dem Wert erschliesst.
Die Definition einer Liste von eigenen Konstanten beginnt mit dem Schlüsselwort 'const', und endet mit dem Schlüsselwort 'endconst'.

Eine einzelne Konstante wird wie folgt definiert :
<***constant_name***> = <***value***> ;
oder (mit Angabe des Datentyps, was nötig ist, wenn z.B. der Datentyp der Konstante sich nicht aus dem Wert ergibt, oder mehrdeutig sein könnte) :
<***data_type***> <constant_name> = <value> ;
oder (zur Definition eines Arrays aus Konstanten, Details dazu weiter Unten) :
<data_type> <constant_name> [***array_size***] = <value> ;

Beispiel (bitte beachten Sie den Quelltext-Stil - Einrücken zwischen const und endconst) :

```
const
  // Identifiers for various graphic control elements (buttons, etc)
  idBtnUp    = 1;  // button to scroll up    the text panel
  idBtnDown  = 2;  // button to scroll down  the text panel
  idBtnLeft  = 3;  // button to scroll left  the text panel
  idBtnRight= 4;  // button to scroll right the text panel
  idBtnClose= 5;  // button exit from 'text file' display page

  MAX_DIR_ENTRIES = 20;
endconst;  // end of a list of user-defined 'constants'
```

Selbstdefinierte Konstanten müssen im Quelltext *vor* deren Verwendung definiert werden (idealerweise stehen sie 'am Anfang' des Scriptes).

Hinweise:

- Verwenden Sie symbolische Konstanten anstatt 'magischer numerischer Werte' im Quelltext - niemand (außer vielleicht Ihnen selbst) wird wissen, was diese numerischen Werte bedeuten sollen, speziell in langen select..case - Listen, und als Control-Identifier in Message-Handlern.
- Als Control-Identifier verwendete Konstanten (für graphische Steuerelemente, englisch: 'Controls') sollten mit dem Namensvorsatz 'id' beginnen, um sie von anderen Konstanten zu unterscheiden.
  Beispiel: idBtnUp, idBtnDown, idBtnLeft, ... statt 'id1, id2, id3'. Siehe auch: Auswahl eines Control-Identifiers in der Definition eines Buttons per Doppelklick auf das Feld 'Control ID').
- Genau wie bei Namen von Variablen, Datentypen, und ähnlichen Elementen sind die Namen (Symbole) von Konstanten auf maximal 20 Zeichen begrenzt.
- Innerhalb eines Scripts können maximal 256 verschiedene const...endconst - Blöcke verwendet werden. Die Anzahl von einzelnen Konstanten (in diesen Blöcken) ist nur durch die Größe des Bytecode-Speichers begrenzt (geräteabhängig, i.A. mindestens 64 kByte für den *gesamten Bytecode*).
- Der UPT-*Display-Interpreter* kann ebenfalls auf Script-Konstanten zugreifen, ohne dass dem Konstantennamen dazu der Prefix "script." vorangestellt werden muss.
  Dies funktioniert allerdings nur, wenn der Name der Script-Konstanten nicht länger als die maximale Länge einer *Display-Variablen* (!), d.h. maximal 8 bis 16 Zeichen (Firmware-abhängig).
  Die Länge des Namens einer Script-Konstanten *im Script selbst* ist nahezu unbegrenzt.

3.3 4.3.3 'Berechnete' Konstanten (berechnet beim *Kompilieren*, nicht während der *Laufzeit*)

Um den Wert von einfachen numerischen Ausdrücken (als Konstante) bereits beim Übersetzen, statt erst zur Laufzeit des compilierten Programmes zu berechnen, kann mit Hilfe des Doppelkreuz-Zeichens ein (einfacher!) Ausdruck bereits vom Compiler berechnet werden.

Beispiel:

Der Ausdruck in der folgenden Zuweisung..

```
N := #(1+2+3)
```

wird bereits beim *Übersetzen* (durch den Compiler selbst, nicht durch das Laufzeitsystem) berechnet.
Im Gegensatz dazu wird der Ausdruck in der folgenden Zuweisung..

```
N := 1+2+3
```

erst *während der normalen Laufzeit* berechnet, was zu einer etwas geringeren Geschwindigkeit führen könnte.
Das oben gezeigte, bewusst einfach gehaltene Beispiel funktioniert statt mit numerischen Konstanten auch mit symbolischen Konstanten. Der Zugriff auf den Wert von Variablen ist im geklammerten Ausdruck (nach dem Doppelkreuz im ersten Beispiel) natürlich nicht möglich, denn der Wert ("Inhalt") einer Variablen ist dem Compiler nicht bekannt.

Bitte beachten: Der im Compiler integrierte Parser für Konstanten-Ausdrücke unterstützt nur sehr einfache Operationen, kennet keine Prioritäten (keine "Punktrechnung vor Strichrechnung", keine Klammer-Ebenen), und kann bei der Berechnung von Konstanten keine Funktionen aus der Laufzeitumgebung aufrufen (selbst dann nicht, wenn das *Funktionsargument* eine Konstante ist). Der einzige vom Compiler 'berechenbare' Datentyp ist Integer .

3.4 4.3.4 Konstanten-Arrays

Für einige Anwendungen ist es nötig, Konstanten in einem Array anzulegen ("Tabelle mit konstantem Inhalt"). In dieser Tabelle kann dann z.B. per Index N auf die N-te Konstante zugegriffen werden. Man könnte dazu auch ein Variablen-Array verwenden, welches aber *während der Laufzeit* 'mit Werten gefüllt' werden müsste. Wesentlich eleganter und effizienter ist der Einsatz eines Konstanten-Arrays, denn dies wird bereits *beim Übersetzen* des Scriptes mit Daten gefüllt.

Beispiel (ein Konstanten-Array mit Farbwerten, stammt aus dem 'quadblocks' - Demo) :

```
const
  int BlockColors[7] = // seven block colours : BlockColors[0...6]
!
    { clBlue, clRed, clCyan, clYellow, clGreen, clMagenta, clWhite
};
endconst;
```

Hinweis: Wie bei allen Arrays, beginnt die Indexzählung auch hier bei Null !
Das erste Element im obigen Beispiel ist daher BlockColors[0], das letzte BlockColors[7] !
Siehe auch : Konstanten (Übersicht), Variablen-Arrays, Anzeige von Tabellen (Anzeige-Element).

4 4.4 Feste und anwenderdefinierte Datentypen

Die Script-Sprache unterstützt bislang nur die folgenden 'elementaren' (fest eingebauten) Datentypen:

- **`int`** (alias "integer") : Vorzeichenbehaftete 32-bit Ganzzahl. Dies ist der bevorzugte ("schnellere") numerische Datentyp.
- **`float`** : 32-bit Fließkommazahl mit 'einfacher Präzision'. Besteht aus dem Exponenten (8 Bit), der Mantisse (23 Bit), und dem Vorzeichen (1 Bit).
  Variablen mit diesem Typ können auch Dezimalbrüche wie 0.12345 speichern. Bei älteren Geräten (z.B. mit ARM7TDMI und Cortex-M3) waren Berechnungen mit diesem Datentyp wesentlich langsamer als mit Integer-Zahlen. Bei neueren Geräten mit Cortex-M4F-CPU (z.B. MKT-View IV) sind Fließkomma- Operationen dank Hardware-Fliesskomma-Einheit fast so schnell wie Integer-Operationen.
  Wenn nötig, kann ein Fliesskommawert aus einzelnen Bytes (mit Exponent und Mantisse) 'zusammengesetzt' werden. Dazu dienen die Funktionen BytesToFloat() oder BinaryToFloat(). Mit der Funktion ftoa() ('float to ASCII') kann eine Fliesskommazahl mit einer definierbaren Anzahl von Vor- und Nachkommaziffern in eine Zeichenkette (string) umgewandelt werden.
- **`double`** : 64-bit Fließkommazahl mit 'doppelter Präzision'. Bietet auf Kosten der Geschwindigkeit eine wesentlich höhere Genauigkeit als der einfache (32-Bit-)Datentyp

'float'.
Dieser Typ sollte daher nur verwendet werden, wenn Fließkommazahlen mit *einfacher* Genauigkeit (float) nicht ausreichen, z.B. ...

- Zum Speichern von Datum **und** Uhrzeit mit extremer Auflösung als 'Unix-Zeit' (mit Nachkommastellen)
- Für Berechnungen mit Längen- und Breitengraden von präzisen DGPS-Empfängern (mit Auflösung im Zentimeterbereich)
- Zum Speichern des Rückgabewertes der Funktion BytesToDouble(), wenn die *maximale* Auflösung benötigt wird

- **`string`** : Zeichenkette. Die Zeichen selbst belegen (abhängig von der Länge) zusätzlichen Speicher. Bei der Berechnung von Struktur- und Arraygrößen belegt ein Objekt mit dem Datentyp 'string' scheinbar nur 4 Bytes, dies ist in der Tat aber nur ein *Pointer* (Zeiger) auf die eigentlichen Zeichen im dynamisch verwalteten Blockspeicher. Details zum Datentyp 'String' finden Sie hier).
- **`char`** : Ein einzelnes Zeichen (8 Bit). Dient als Array zur Speicherung von Strings mit *fester* Länge, z.B. bei der Definition von Strukturen (mehr dazu später).

Die oben aufgeführten Typnamen dienen auch als Funktionsnamen für explizite Typumwandlungen. Der Ausdruck

```
int( 100.0 * Math.cos(phi) )
```

liefert daher einen Integer-Wert, auch wenn die Berechnung Fliesskommazahlen verwendet.

Zusätzlich zu den elementaren Datentypen existieren für die Definition von Strukturen und Arrays noch die folgenden Typen, die für Berechnungen automatisch nach 'int' umgewandelt werden:

- **`byte`** : Vorzeichenlose 8-Bit-Ganzzahl. Wertebereich 0 bis 255 .
  Dieser Datentyp belegt in *Arrays* oder *Strukturen* genau ein Byte (als einzelne Variable, oder auf dem RPN-Stack aber genausoviel wie eine normaler Integer-Wert)
- **`word`** : Vorzeichenlose 16-Bit-Ganzzahl. Wertenbereich 0 bis 65535 .
  Dieser Datentyp belegt nur in *Arrays* oder *Strukturen* genau zwei Bytes (sonst mehr, wegen 4-Byte-Alignment für "einzelne Werte").
- **`dword`** : Vorzeichenlose 32-Bit-Ganzzahl (kann nicht 'verlustfrei' in 32-bit-Integer-Berechnungen verwendet werden, sondern dient nur zur 'Speicherung' !) .
  Belegt in *Arrays* oder *Strukturen* genau vier Bytes pro Eintrag.
- **`bool`** : Ähnlich wie Integer, allerdings zum Speichern der Werte TRUE (1) oder FALSE (0) optimiert. Wird bislang (2019-01) bei *Berechnungen* wie der Datentyp int behandelt (d.h. TRUE plus TRUE ergibt 2 (Zwei), was als boolscher Wert zwar Unfug darstellt, aber trotzdem 'funktioniert' weil bei der Abfrage (z.B. in einer if-Bedingung) lediglich zwischen 'Null' und 'ungleich Null' unterschieden wird. Beim Testen eines boolschen Wertes 'x' sollte daher nicht 'if x==TRUE ', sondern z.B. 'if x<>FALSE' oder einfach 'if (x)' verwendet werden. Nur so funktioniert wie Abfrage, wenn 'x' z.B. aus der Abfrage eines bestimmte Bits durch eine bitweise Und-Verknüpfung funktioniert:

  ```
  x := (some_bitcombination & 0x0100);
  ```

In zukünftigen Versionen des Script-Compilers könnte bei *Arrays vom Typ 'bool'* eine speicherplatzsparendes Verfahren (mit nur *einem Bit* pro Element) verwendet werden.

- **`tColor`** : Ebenfalls ein Integer-Wert, allerdings mit der Bedeutung 'Farbe'. Wird von manchen Funktionen mit variabler Parameteranzahl als optionaler Parameter verwendet, z.B. beim Zeichnen von Polygonen in Diagrammen. Farben können z.B. mit der Funktion rgb(red,green,blue) 'gemischt' werden.

Wenn nötig, werden die oben genannten Datentypen werden zur Laufzeit automatisch umgewandelt. Hier ein Beispiel mit automatischer Umwandlung von Integer nach Fliesskomma:

```
var
  int   i;  // Deklaration einer Integer-Variablen
  float f;  // Deklaration einer Fliesskomma-Variablen
endvar;

i := 1234;     // Zuweisung Integer-Wert an Integer-Variable
f := i;        // Zuweisung Integer-Wert an Fliesskomma-Variable
// (mit automatischer Konvertierung. In älteren Versionen musste dazu
//  die folgende explizite Konvertierung definiert werden:
//  f := float(i); // Umwandlung nach 'float', dann Zuweisung an 'f' )
```

Darüberhinaus verfügt die Script-Sprache über einige 'fest eingebaute' Datenstrukturen, z.B.:

- **`tScreenCell`** : Datentyp zur Beschreibung einer einzelnen 'Zelle' im Puffer für den emulierten Text-Bildschirm.
  Die Komponenten dieser Struktur sind:
  `.bChar` : 8-Bit-Zeichencode, i.A. ASCII oder "DOS", 0..255
  `.bFlags` : Bit 7 = Flag zum Erzwingen des Neu-Zeichnens *dieser* Zelle.
  `.bFontNr` : reserved for future use
  `.bZoom` : reserved for future use
  `.fg_color` : foreground colour (Vordergrundfarbe)
  `.bg_color` : background colour (Hintergrundfarbe)

- **`tCanvas`** : Datentyp für eine 'Leinwand' (engl. canvas), mit dem das Script beliebige Grafiken *zur Laufzeit* erzeugen kann.
  Die Komponenten dieser Struktur standen zum Zeitpunkt der Erstellung dieser Dokumentation (12/2017) noch nicht fest - beabsichtigt sind Ähnlichkeiten mit HTML5 <canvas> (width,height,data).
  Komponenten von tCanvas:
  `.width` : Breite (Anzahl Pixel)
  `.height` : Höhe (Anzahl Pixel)
  Siehe auch: Canvas-Funktionen und -Methoden zum 'Zeichnen'.

- **`tMessage`** : Datentyp für Message-Handling. Enthält die folgenden Komponenten:
  `.receiver` : Identifiziert den geplanten 'Empfänger' der Message (der Wert Null steht hier für 'Broadcast'-Meldungen, d.h. 'Nachricht an Alle')
  `.sender` : Identifiziert den Sender der Message. Ist der Sender kein 'sichtbares' Steuerelement (windowed control), könnte dieser Wert Null sein).

`.msg` : Message-Typ. Eine der mit 'wm' ('windows message') beginnenden Konstanten, eventuell auch benutzerdefiniert (i.V.).
`.param1` : Erster Message-Parameter. Die Bedeutung hängt vom Meldungstyp ab.
`.param2` : Zweiter Message-Parameter. Die Bedeutung hängt vom Meldungstyp ab.
`.param3` : Dritter Message-Parameter. Die Bedeutung hängt vom Meldungstyp ab.
Details zur Verwendung dieses Datentyps finden Sie im Kapitel 'Message-Handling' .

- **`tCANmsg`** : Datentyp für die Ablage eines einzelnen CAN-Bus-Telegrams im Speicher. Dient (als 'Pointer auf tCANmsg') für die Parameterübergabe an selbstdefinierte CAN-Empfangs-Handler, und kann (optional) als Typ des Arguments beim Aufruf des Kommandos can_transmit dienen.
  Komponenten des *Datentyps* **tCANmsg** sind:
  `.id` : Kombination aus Bus-Nummer (in Bits 31+30), Standard/Extended-Flag (in Bit 29), und dem 11- oder 29-Bit CAN-ID (in Bits 10 bis 0 bzw. 28 bis 0).
  `.tim` : Timestamp (enthält bei *empfangenen* Telegrammen eine präzise Zeitmarke vom CAN-Treiber).
  Zum Vergleich von Zeitmarken lesen Sie bitte unbedingt diesen Hinweis !
  `.len` : Länge der 'Nutzdaten' in Bytes. Bei CAN sind nur 0 bis 8 Bytes pro Telegramm erlaubt, bei CAN FD maximal 64.
  Zum Anfordern einer Übertragung kann die Länge mit dem Bitflag cCanRTR ('Remote Transmit Request') kombiniert werden.
  Bei CAN wird RTR selten verwendet. Ein LIN-Master kann einen Slave per cCanRTR zum Senden eines Datenfelds auffordern.
  `.b[0] .. b[7]` : Nutzdatenfeld als Byte-Array (acht mal 8 Bits)
  `.w[0] .. w[3]` : Nutzdatenfeld als Word-Array (vier mal 16 Bits)
  `.dw[0] .. dw[1]` : Nutzdatenfeld als Doubleword-Array (zwei mal 32 Bits)
  `.i32[0] .. i32[1]` : Nutzdatenfeld als Array mit zwei 32-Bit Integer-Werten, little endian ("Intel")
  `.f32[0] .. f32[1]` : Nutzdatenfeld als Array mit zwei 32-Bit Fliesskomma-Werten, little endian
  `.f64[0]` : Nutzdatenfeld als Array mit 64-Bit Fliesskomma-Werten (für **CAN**: nur *ein* Element)
  `.bitfield[` <Bit-Index des LSBs> `,` <Anzahl Datenbits>`]` : Bitfeld wie in den Variablen 'can_rx_msg' und 'can_tx_msg'

  Zur Erleichertung der Kommunikation per J1939 stehen zusätzlich folgende 'Aliase' für den 29-Bit-CAN-ID zur Verfügung:
  `.PRIO`: 'Message Priority'. Alias für CAN-ID Bits 28 bis 26.
  `.EDP`: 'Extended Data Page'. Alias für CAN-ID Bit 25.
  `.DP` : 'Data Page'. Alias für CAN-ID Bit 24.
  `.PF` : 'PDU Format'. Alias für CAN-ID Bits 23 bis 16.
  `.PS` : 'PDU Specific'. Alias für CAN-ID Bits 15 bis 8.
  `.SA` : 'Source Address'. Alias für CAN-ID Bits 7 bis 0.
  `.PGN`: 'Parameter Group Number' mit bis zu 18 Bit. Dies ist ein weiterer Alias für 'DP'+'PF'+'PS'.

Zur Erleichertung der Kommunikation per ISO-TP / ISO 15765-2 wurden weitere Aliase für den 29-bit CAN-ID implementiert.
Warnung (in Anbetracht der Vielfalt an Adressierungsarten in ISO 15765-2) : Die folgenden Aliase gelten nur für "Normal Fixed Addressing", und "Mixed Addressing mit 29-Bit CAN Identifiern", aber nicht für "Normal Addressing" !
`.ID28_26` : Bits 28 bis 26 im 29-Bit-CAN-ID. Bei ISO-TP mit "normal fixed addressing" steht dort meistens 0b110 (binär) .
`.ID25_24` : Bits 25 bis 24 im 29-Bit-CAN-ID. Bei ISO-TP mit "normal fixed addressing" steht dort fast immer 0b00 (binär) .
`.ID23_16` : Bits 23 bis 16 im 29-Bit-CAN-ID. Bei ISO-TP mit "normal fixed addressing, TAtype=physical" steht dort z.B. '218' (dezimal) .
`.ISO_TA` : ISO 15765-2 'Target Address'. Alias für CAN-ID Bits 15 bis 8, allerdings nicht bei allen Adressierungsarten !
`.ISO_SA` : ISO 15765-2 'Source Address'. Alias für CAN-ID Bits 7 bis 0. Man beachte die überraschende Kompatibilität mit J1939.

Wird ein Wert vom Typ tCANmsg, oder ein Pointer auf eine Variable vom Typ tCANmsg in einen String konvertiert, dann kann u.A. eine Zeichenkette im Vector-ASC-Format erzeugt werden, die sich z.B. zum Aufzeichnen als Log-Datei (per Script, auch bei Geräten ohne CAN-Logger) eignet.
Für Geräte mit CAN-FD-kompatiblem Controller existiert neben **tCANmsg** noch der Datentyp **tCAN_FD_msg**, der (im Gegensatz zum 'classic CAN') ein bis zu 64 Byte langes Datenfeld bietet.

- **`tTimer`** : Datentyp für einen programmierbaren Timer, mit folgenden Komponenten:
  `.period_ticks` : Periodendauer des Timers, umgerechnet in 'Ticks' des Zeitmarken-Generators
  `.expired` : >=1 (TRUE) wenn ein Timer dieses Typs abgelaufen ist, andernfalls 0 (FALSE)
  `.ts_next` : Stand des Zeitmarken-Generators (system.timestamp) beim *nächsten* Ablauf dieses Timers
  `.running` : Liefert den Wert 1 (TRUE) wenn dieser Timer zur Zeit 'läuft', anderfalls 0 (FALSE). Dient nicht zum Starten !
  Hinweis: Ohne Event-Handler, und ohne explizites Stoppen bleibt 'running' immer gesetzt !
  `.user` : Ein frei verwendbares Feld vom Typ anytype. Kann z.B. als Zähler oder Index (→ Timer-Event-Demo), aber auch als Pointer (Zeiger auf andere Objekte) verwendet werden.
  Dieser Datentyp wird für das Kommandos setTimer verwendet, und (als 'Pointer auf tTimer') beim periodischen Aufruf eines Timer-Event-Handlers.
  Hinweis: Ein Timer kann nicht per 'timerX.running := TRUE' *gestartet* werden.
  Um einen nicht-laufenden Timer zu *starten*, verwenden Sie den Befehl setTimer mit Zeitangabe.
  Grund: Eine Zuweisung an 'timerX.running' lädt den Zähler nicht nach.
- **`tTable`** : Datentyp für die graphische Anzeige einer Tabelle auf dem Bildschirm.
  Eine Variable des Typs 'tTable' enthält *nicht die Daten*, die als Tabelle auf dem Bildschirm

angezeigt werden sollen, sondern dient u.A. als Bindeglied zwischen der 'Datenquelle' (z.B. einem Array) und dem sichtbaren Anzeige-Element ("\table"). Details zu Tabellen (als Anzeige/Bedienelement) finden Sie in einem separaten Dokument (table_49.htm).

- **tDirEntry** : Datentyp zum Lesen des Inhaltsverzeichnisses eines Dateisystems (Speicherkarte, RAMDISK, etc).
  Details im Kapitel Lesen von Verzeichnissen.

Mit dem Schlüsselwort '**ptr**' oder alternativ '*' (als Zusatz zum Datentyp) können auch Pointer (Zeiger) in der Script-Sprache deklariert werden. Im Gegensatz zu Pascal (u.A.) ist 'ptr' kein eigener Datentyp, sondern muss immer mit anderen Typen kombiniert werden.

Einige Script-Funktionen können *unterschiedliche* Datentypen als Rückgabewert ('return value') liefern. Um *den Typ* des zurückgelieferten Wertes zu untersuchen, kann der Aufrufer den typeof-Operator wie im folgenden Beispiel in einer select..case - Anweisung einsetzen, um je nach Datentyp eine unterschiedliche Verarbeitung durchzuführen. In der case-Liste werden dazu symbolische Konstanten wie z.B. dtFloat, dtInteger, dtString verwendet; und die Variable (für den Return-Wert) als 'anytype' deklariert:

- **anytype** : 'Dummy'-Datentyp zur Deklaration einer Variablen, die (im Prinip) "jeden Typ" (any type) annehmen kann.
  Nach dem Zuweisen eines Wertes an eine als 'anytype' deklarierte Variable kann der Typ dieses Wertes mit dem typeof()-Operator abgefragt werden.
  Beispiel:

```
var
     anytype result;
endvar; // end of variable declarations
     ...
     result := cop.sdo(0x1234, 0x01);  // read something via CANopen
(Service Data Object)
     select( typeof(result) )  // what's the TYPE OF the returned value ?
        case dtInteger:  // the SDO transfer delivered an INTEGER
           ...
        case dtByte:     // the SDO transfer delivered a BYTE
           ...
        case dtString:   // the SDO transfer delivered a STRING
           ...
        case dtError:    // the SDO transfer returned an ERROR CODE
           ...
     endselect;
```

Zusätzlich zu den weiter oben aufgeführten 'fest eingebauten' Datentypen können in der Script-Sprache auch eigene Datentypen definiert werden. Beispiel:

```
typedef
  tHistoryBufEntry = // this is the name of a user-defined data type
     struct          // begin of a user-defined data structure
        int iRefNo;     // 1st member: an integer variable
        int iSender;    // 2nd member: another integer
```

```
        float fUnixTimestamp;  // 3rd: a floating point variable
        string sInfo;    // 4th member: a string
        int x,y,z;       // 5th to 7th: 3 members widh the same type
    endstruct;     // end of the user-defined data structure
end_typedef; // alias endtypedef, end of the data type definition
```

Hinweise:

- Ein einzelner 'typedef'-Block kann mehrere Datentyp-Definitionen enthalten. Das Semikolon trennt in dem Fall mehrere Definitionen.
- Datentypen müssen *definiert* sein, bevor sie in der *Deklaration* von Variablen verwendet werden können.
  Es empfiehlt sich daher, mindestens einen 'typedef'...'endtypedef'-Block am Anfang des Scriptes (vor den Variablen-Deklarationen) einzusetzen.
- Das kleine 't' als Prefix im Namen einer *Typ-Definitionen* ist nicht zwingend vorgeschrieben, wird aber im Interesse der Lesbarkeit des Quelltextes empfohlen.
- Komponenten innerhalb einer Typdefinition müssen per Semikolon getrennt werden (Doppelpunkt oder Zeilenende reicht hier nicht) .
- Das Schlüsselwort **typedef** leitet die Definition einer Liste von *Datentypen* ein, das Schlüsselwort **end_typedef** (oder **endtypedef**) beendet diese Liste.
- Das Schlüsselwort **struct** steht am Anfang einer *Struktur-Definition*, das Schlüsselwort **endstruct** beendet die Definition *einer* Struktur.
- Strukturen können innerhalb der Script-Sprache nur komponentenweise manipuliert werden. Das Kopieren einer 'kompletten' Struktur mit einer einzigen Anweisung ist (noch) nicht möglich.
- Da der Datentyp 'string' innerhalb einer Strukturdefinition (wie im oben angeführten Beispiel) lediglich ein Zeiger (pointer) auf die Zeichenkette in einem dynamisch allozierten Speicherblock ist, belegt der String innerhalb der Struktur wesentlich weniger Bytes, als dies die Länge des Strings erwarten liesse.

Im Moment können selbstdefinierte Strukturen nur als 'einfachen' Datentypen zusammengesetzt sein. Ineinander verschachtelte Strukturen, und Strukturen mit Arrays als Komponenten, waren zum Zeitpunkt der Erstellung dieses Dokuments nicht möglich (im Gegensatz zu Arrays, bei denen jedes Array-Element aus einer Struktur bestehen darf).

Siehe auch: var..endvar zur Definition von *globalen* Variablen.

4.1 4.4.4 Explizite Typenumwandlungen (typecasts)

In seltenen Situationen müssen Datentypen ***explizit*** umgewandelt werden.

Die Syntax des Typecast-Operators entspricht der Programmiersprache "C":

```
(<Datentyp>)<Zu konvertierender Wert>
```

Beispiel:

```
s1 := (string)n; // typecasting integer 'n' into string 's1'
```

Alternativ kann statt der obigen Typecast-Syntax auch der Name des Datentyps als Funktionsname verwendet, und der zu konvertierende Wert als Funktionsargument in Klammern übergeben werden:

```
    <Datentyp>( <Zu konvertierender Wert> )
```

Beispiel (aus der Applikation programs/script_demos/StringTest.cvt) :

```
    s1 := string(n); // default method to convert "anything into a string"
```

Die Umwandlungsfunktion (hier: string) liefert als Return-Wert dann den gleichnamigen Datentyp.

Vorsicht beim Einsatz von expliziten Typumwandlungen im Zusammenhang mit Pointern:
In einem 32-Bit-Integerwert kann zwar die 32-Bit-Adresse (aus einem Pointer) gespeichert werden, nicht aber der Typ des Objektes, auf das der Pointer zeigte. Nach der Umwandlung Pointer -> Integer -> Pointer hat das Laufzeitsystem daher keine verlässliche Information mehr, auf welchen Typ von Objekt der Pointer zeit. Anhand des Adresswert lässt sich zwar noch feststellen, ob ein Pointer auf Codespeicher, Stack, globale Variablen, oder dynamisch allozierte Objekte zeigt, aber nicht den Typ des Objektes, Array-Größen, usw.
Aus diesem Grund sollten Typenumwandlungen 'mit Pointern' vermieden werden.
Bei der Umwandlung bestimmter *fest in der Script-Sprache implementierter* Datentypen gelten folgende Regeln:

- int: wird in das auch von itoa() verwendete Format konvertiert (dezimal, Vorzeichen nur wenn negativ)
- tCANmsg: Für diesen Datentyp kann das String-Format durch Setzen von CAN.string_format definiert werden.
  Mit CAN.string_format := sfVectorASC wird eine CAN-Message per string() in eine Zeichenkette im Vector ASC Format konvertiert, *ohne Carriage Return und New Line*.
  Der Zeitstempel (erste Spalte in einer Vector-ASC-Datei) kann per CAN.timestamp_offset 'verschoben' werden.
  Ein Beispiel finden Sie in der Applikation programs/script_demos/CAN_ASC_Logger.cvt .

4.5 Variablen

Die Script-Sprache unterstützt lokale und globale Variablen (mehr dazu im nächsten Kapitel). Ferner kann das Script auch auf die in der Display-Applikation definierten Variablen des UPT-Interpreters ("Anzeige-Variablen") zugreifen.

Wie in den meisten Programmiersprachen *sollten* Variablen vor deren Verwendung deklariert werden.

Ähnlich wie in altem BASIC kann der Datentyp von *nicht deklarierten globalen Variablen* (die einfach "durch ihre Verwendung" existieren) durch einen speziellen Namensanhang ***(suffix)*** definiert werden. Von dieser Möglichkeit sollte nach Möglichkeit aber kein Gebrauch mehr gemacht werden (sie existiert nur noch aus Gründen der Abwärtskompatibilität). *Veraltet*: Nicht deklarierte Variablen, bei denen der Datentyp per *Suffix* definiert wurde, z.B. '%' für "integer", '&' für "long integer", '$' für "string", und '!' für Fliesskomma.

Zur Wiederholung: Die oben erwähnte Verwendung *nicht deklarierter* Variablen ist veraltet und sollte vermieden werden.
Stattdessem sollten **alle** Variablen vor der Verwendung *deklariert* werden, wie in den folgenden Kapiteln beschrieben. Beispiel:

```
#pragma strict // 'strict' compilation: ANY variable must be declared before
being used !

var // global variables (accessable from all subroutines) ...
    int nLines, nColumns; // a few integer variables
    string sInfo;         // a single string variable
    tTable MyTable;       // control object for a 'visual' table element
    string MyData[5][3];  // an array of strings which will be displayed in a
table [5 lines][3 columns]
endvar;
```

Mit der Direktive **#pragma strict** (in einer Zeile *vor* dem Deklarationsteil) kann der Compiler angewiesen werden, ***nur*** deklarierte Variablen zu verwenden. 'Automatisch' erzeugte globale Variablen sind dann nicht mehr möglich. Diese stellten sich in der Vergangenheit als tückische Fehlerquelle heraus.

Hinweis für Entwickler:

> Globale Variablen können während der Laufzeit im Programmiertool per Debugger (Symboltabelle mit aktuellen Werten) oder im Remote-Betrieb per Web-Browser auf der Seite 'script' inspiziert werden. Mit lokalen Variablen funktioniert dies nicht, da sie keine 'feste Adresse' haben, und mit dem gleichen Namen in unterschiedlichen Instanzen (auf dem Stack) existieren könnten. Das Gleiche gilt auch für die Inspektion von Variablen am programmierbaren Gerät.

5 4.5.1 Variablen-Deklarationen im Script

Wie bereits erwähnt, müssen 'normale' (einfache, globale) Variablen nicht unbedingt deklariert werden - im einfachsten Fall beginnt die Existenz einer Variablen bei deren erstmaliger

Verwendung (und zwar, wie in BASIC, als globale Variable). Es empfiehlt sich trotzdem (oder grade deswegen), jede Variable vor der ersten Verwendung zu deklarieren.

Hinweise:

Bei Verwendung der Option '#pragma strict' *müssen* Variablen vor der Verwendung deklariert sein.
Mit dem Prefix 'display.' kann das Script auch auf Anzeige-Variablen zugreifen, die im Programmiertool auf der Registerkarte 'Variablen' definiert wurden.

5.1 4.5.1.1 Globale Script-Variablen

Die (empfehlenswerte) Deklaration von globalen Variablen beginnt mit dem Schlüsselwort 'var', und endet mit dem Schlüsselwort 'endvar' (alias 'VAR' ... 'END_VAR' für die Freunde von IEC 61131 "Structured Text").
Hier ein Beispiel zur Deklaration einiger globaler Variablen. Beachten Sie wieder den empfohlenen Quelltext-Stil ... hier die Einrückung zwischen 'var' und 'endvar' :

```
#pragma strict // 'strict' compilation: ANY variable must be declared before
being used !

var // global variables (accessable from all subroutines) ...
   int nHistoryEntries;  // declares an integer variable
   tHistoryBufEntry History[100]; // declares an array of tHistoryBufferEntries
    // Note: the indices for an array with 100 entries run from 0 (ZERO) to 99 !

 logged: // The following variables may be recorded by the built-in CAN logger:
    // Signals from J1939 PGN 61443 = "Electronic Engine Controller 2" :
    int   AccelPedalKickdown;   // SPN 559  "Accelerator Pedal Kickdown Switch"
    int   AccelPedalPosition1;  // SPN 91   "Accelerator Pedal Position 1"

 private: // The following variables shall NOT be 'logged':
    int   iSomeInternalStuff;
    ...
endvar;
```

Zwischen den Schlüsselwörtern '**var**' und '**endvar**' (d.h. bei der Deklaration von *globalen* Variablen) können optional die folgenden Attribute angegeben werden, die sich auf die ***nachfolgenden*** Deklarationen beziehen:

private:

Die nachfolgenden Variablen dürfen nur innerhalb des Scripts verwendet werden;
sie sollen z.B. *nicht* in der Auswahlliste für die Definition von Anzeigeseiten erscheinen, und sind auch für den (in manchen Geräten integrierten) Logger "tabu".

public:

Die nachfolgenden Variablen sind auch ausserhalb des Scripts verwendbar;
und werden (vom Programmiertool) u.A. in die Auswahlliste bei der Definition von Anzeige-Elementen übernommen.

logged:

Die nachfolgenden Variablen *dürfen*(*) vom in manchen Geräten integrierten Logger aufgezeichnet werden.
Mit dem Attribut 'private:' (s.O.) kann das Ende der log-baren Variablen markiert werden.

(*) Das Attribut 'logged' bedeutet nicht, dass die nachfolgenden Variablen *immer* geloggt werden. In der CAN- Logger-Konfiguration muss zusätzlich die Option + **als 'logged' deklarierte Script-Variablen** gesetzt sein, um nicht nur CAN-Telegramme und Daten vom GPS-Empfänger, sondern auch Script-Variablen aufzuzeichnen.

noinit:
Mit diesem Attribut deklarierte Variablen werden *nach Möglichkeit* beim Nachladen anderer Applikationen per system.exec() ***nicht*** automatisch initialisiert.
Dies funktioniert nur mit mit einfachen Datentypen wie z.B. int, aber nicht mit dynamisch allozierten Typen wie z.B. string, da vor dem Compilieren des Scripts und dem Initialisieren der Script-Runtime alle dynamisch allozierten Speicherblöcke wieder freigegeben werden.

Globale Variablen können unabhängig von der Deklaration als 'private' oder 'public' zur Laufzeit im Programmiertool per Debugger (mit aktuellen Werten in der Symboltabelle) oder im Remote-Betrieb per Web-Server / Seite 'script' inspiziert werden.

Globale Variablen werden üblicherweise im *Initialisierungsteil* des Scripts mit Defaultwerten besetzt (falls diese von Null verschieden sein sollen). Es empfiehlt sich, erst *danach* durch Aufruf von 'init_done' die Event-Handler zu aktivieren. So wird vermieden, dass z.B. Event-Handler bereits aufgerufen werden, bevor das Script seine eigene Initialisierung (z.B. Laden von Konfigurationsdaten) abgeschlossen hat, und ungültige Werte verwendet werden.

5.2 4.5.1.2 Lokale Script-Variablen

Innerhalb Prozeduren oder Funtionen können, wie im folgenden Beispiel, ***lokale*** Variablen deklariert werden. Lokale Variablen verwenden den *Stack* als Speichermedium, und existieren daher nur innerhalb der Prozedur (oder Funktion), in der sie deklariert sind. Kehrt die Funktion / Prozedur zum Aufrufer zurück, endet die Existenz der lokalen Variablen. Dies hat den nicht zu unterschätzenden Vorteil, daß die Funktion weniger 'Nebeneffekte' (als bei der Verwendung globaler Variablen innerhalb der Funktion) hat.

Beispiel:

```
proc Test
   local int x,y,z; // define three local integer variables
      ....
    print( x,y,z );
endproc // local variables (x,y,z) cease to exist at this point
```

Hinweis: Im Gegensatz zu 'var'...'endvar' gibt es kein Schlüsselwort 'endlocal'. Das Schlüsselwort '**local**' bezieht sich auf die in der gleichen Zeile definierten Variablen.

Um zu vermeiden, daß der Stack-Speicher knapp wird, sollten in einer Funktion oder Prozedur nur die lokalen Variablen deklariert werden, die innerhalb der Funktion/Prozedur wirklich gebraucht

werden (der Compiler optimiert unnötige Deklarationen nicht 'weg'). Besonders beim gegenseitigen rekursiven Aufruf von Prozeduren konnte der Stack knapp werden, weil bei jedem Funktionsaufruf entsprechend viel Platz auf dem Stack benötigt wird. Erst bei der Rückkehr aus der aufgerufen Funktion wird dieser Platz wieder frei.

Siehe auch: Debugging ... Anzeige des Stack-Speichers

5.3 4.5.1.3 Pointer (Zeiger)

Ähnlich wie in der Programmiersprache "C" können Variablen für *Sonderzwecke* als 'Pointer' deklariert werden. Dabei ist allerdings (wie weiter Unten beschrieben) äußerste Vorsicht geboten, da das Laufzeitsystem die Gültigkeit eines Pointers nicht exakt prüfen kann.
Ähnlich wie in 'C' können Pointer-Variablen (oder Referenzen) mit der symbolischen Konstante NULL initialisiert werden, z.B. wenn der Zeiger absichtlich 'ungültig' gemacht werden soll.

Beispiel für die Deklaration einer Variablen vom Typ 'Pointer auf Integer':

```
int ptr myPointerToInteger; // declaration of a typed pointer, preferred
```

Um "C"-Entwicklern den Einstieg zu erleichtern, kann das Schlüsselwort 'ptr' auch durch das in "C" übliche Sternchen (*) zwischen dem elementaren Datentyp und dem Namen der deklarierten Variablen ersetzt werden:

```
int * myPointerToInteger;   // declaration of a typed pointer, "C"-style
```

Zweck von Pointern ist z.B. die Verarbeitung von binären Datenblöcken, die nicht in Form eines benutzerdefinierten Datentyps beschrieben werden können.
Bei der Verwendung von Pointern ist (wie in "C") extreme Vorsicht geboten, denn das Laufzeitsystem kann nicht in allen Fällen prüfen, ob ein Pointer auf eine 'gültige' Speicheradresse verweist, bzw. ob der Inhalt an dieser Adresse mit dem verwendeten Datentyp kompatibel ist.
Für die Verwendung von Pointern gilt...

- sicherstellen, dass der Pointer bei der Verwendung *noch gültig ist*
- beachten, dass die Adressen von lokalen Variablen *ungültig* werden, sobald die aufgerufene Funktion zum Aufrufer zurückkehrt
- darum (wenn überhaupt) nur Pointer auf globale Variablen setzen

In vielen Fällen können Pointer durch Arrays oder durch selbstdefinierte Datentypen ersetzt werden. Ein Vorteil von Pointern: Da ein Pointer (im Gegensatz zu Strukturen) lediglich vier Bytes im Speicher belegt, können Pointer (wie auch Referenzen) bei der Übergabe von Parametern effizienter als das Umkopieren einer kompletten Datenstruktur sein.
Für verkettete Listen, Bäume, Tabellen, und ähnliche Datenstrukturen sind Pointer (nahezu) unerlässlich.
Notfalls können Pointer per typecast auch als Integer-Wert gespeichert werden, z.B. im 32-Bit 'user'-Feld eines Timers.

5.3.1 Zuweisen der *Adresse* einer Variablen an einen Pointer

Um einer Pointer-Variablen ***die Adresse*** einer anderen Variablen (oder einer Funktion) zuzuweisen, verwenden Sie die Operator-Funktion **addr**, oder ein dem Variablennamen vorangestelltes **&**

(Ampersand als Adressoperator wie in der Programmiersprache "C"):

Der Ausdruck addr(<variable>) liefert die Adresse der Variablen in der Argumentenliste.

Beispiel für die Deklaration und Initialisierung eines Pointers auf eine 'einfache' Variable :

```
var // declare global variables...
  int myIntegerVar;   // declare an integer variable
  int ptr myPtrToInt; // declare a variable with a pointer to an
integer value
endvar;

myPtrToInt := addr( myIntegerVar ); // assign address of
'myIntegerVar' to pointer 'myPtrToInt'
```

Zum Dereferenzieren eines Pointers auf 'einfache' Datentypen (keine Strukturen) dient, im Gegensatz zu 'C' und Pascal, weder '*' noch '^', sondern ein formaler Array-Index, der in **eckigen Klammern** dem Namen der Pointer-Variablen *nachgestellt* wird. Index Null bedeutet 'das Element, auf welches der Pointer direkt zeigt', usw.
Im Gegensatz zu 'echten' Arrays kann das Laufzeitsystem bei Pointern aber keine Prüfung des Indexes auf Gültigkeit durchführen (da der Pointer ausser dem Datentyp keine Information enthält, 'auf was' er momentan zeigt). Der *Entwickler* ist (wie in 'C') dafür verantwortlich, dass sowohl Pointer als auch der Array-Index zulässig sind !
Der zum Pointer addierte Adress-Offset wird intern durch Multiplikation des Indexes mit der Größe des Datentyps (z.B. 4 Bytes beim 'Pointer auf Integer') berechnet. Bei typfreien Pointern ist nur Index Null zulässig.
Beispiele für die De-Referenzierung eines Pointers auf eine 'einfache' Variable:

```
  myPtrToInt[0] := 12345; // pointer access as a formal array,
here: index zero = first array element
  myPtrToInt[1] := 0; // here ILLEGAL, because in this example
myPtrToInt only points to ONE integer value !
```

Bei Pointern auf Strukturen (z.B. benutzerdefinierte Typen) wird beim Zugriff auf einzelne Strukturkomponenten automatisch der Pointer dereferenziert (im Gegensatz zu Sprache 'C', wo '.' zum Zugriff auf Struktur-Komponenten ohne Pointer, und '->' zum Zugriff mit Pointer verwendet wird).
Beispiel zum Zugriff auf die Komponenten einer Struktur *per Pointer*:

```
typedef // define data types and structs...
  tMyStruct = struct
    int iRefNo;
    string sName;
  endstruct;
end_typedef;
```

```
var // declare global variables...
  tMyStruct myStruct;   // declare a variable of type 'tMyStruct'
  tMyStruct ptr myPtr;  // declare a pointer to a 'tMyStruct'
endvar;

myPtr := addr( myStruct ); // take address of 'myStruct' and
assign it to pointer 'myPtr'
myPtr.iRefNo := 12345;     // actually sets myStruct.iRefNo
myPtr.sName := "Hase";     // (in 'C' this would be myPtr->sName)
```

5.3.2 Parameterübergabe per Pointer

Vorteilhaft sind Pointer besonders bei der Übergabe von Parametern an Unterprogramme (siehe Anwenderdefinierte Funktionen und Prozeduren), da bei der Übergabe eines Pointers (d.h. einer Adresse) an ein Unterprogramm nur vier Bytes kopiert werden, statt (z.B. bei der direkten Übergabe großer benutzerdefinierter Strukturen) einen kompletten Speicherblock zu duplizieren.

Beispiel: Siehe CAN-Empfangs-Handler (dort werden empfangene CAN-Telegramme immer per *Pointer* übergeben, d.h. call-by-reference, nicht call-by-value).

Ein weiteres Beispiel in dem ein Funktionsargument als Pointer übergeben werden *muss* sind die Event-Handler OnGetCellText, OnGetEditText, und OnSetEditText für das Anzeige-Element 'Tabelle'.
Speziell das **OnGetCellText**-Event kann beim Aktualisieren der Anzeige sehr häufig aufgerufen werden. Darum wird das Argument 'sText' (in einer Tabellenzelle anzuzeigender Text) als Pointer übergeben, statt durch Umkopieren eines großen Speicherblocks (bei der Übergabe als 'out'-Argument).

5.4 4.5.2 Zugriff auf Script-Variablen aus dem Anzeigeprogramm

Durch den Vorsatz "script." vor dem Namen einer globalen Script-Variablen kann das Anzeigeprogramm (d.h. der Display-Interpreter) auf 'einfache' *Script-Variablen* in der Display-Applikation (d.h. in den globalen UPT-Ereignis-Definitionen, oder für die Anzeige) zugreifen.

Dabei ist zu beachten, daß das Anzeigeprogramm und das Script quasi-parallel abgearbeitet werden. Prinzipiell läuft das Script im Hintergrund, je nach Betriebssystem in einem eigenen Thread bzw. in einem eigenen Task. Die Variable könnte daher im Script in dem Moment überschrieben werden, in dem sie vom Anzeigeprogramm ausgelesen wird, was speziell bei Zeichenketten zu Problemen führen könnte (da diese nicht mit einem einzigen, nicht unterbrechbaren Befehl kopiert werden können). Per se wird die Bearbeitung von Display (d.h. Anzeige der programmierbaren UPT-Seiten) und Script nicht synchronisiert. Das Script hat allerdings die Möglichkeit, das Aktualisieren der Anzeige vorübergehend zu unterbrechen (siehe display.pause).
Einige Beispielprogramme für die Script-Sprache nutzen diese Möglichkeit, um Script-Variablen auf dem LCD zu 'inspizieren', ohne Zutun des Script-Programms.

In bestimmten Sonderfällen kann auf den Vorsatz 'script.' beim Zugriff auf Script-Variablen, oder beim Aufruf von Script-Funktionen bzw -Prozeduren *aus dem Anzeigeprogramm* verzichtet werden. Dies sind z.Z.:

- Aufruf von Script-Prozeduren aus Button-Events
- Anzeige von Script-Variablen in Anzeige-Elementen (in der Definitionsspalte 'Var/Formel')

Siehe auch:

- Synchronisierung von Script und Anzeige mit dem Befehl 'display.pause'
- Weitere Details zum Thema Interaktion zwischen *Script* und *Display-Applikation* :
  - Zugriff auf ***Display-Variablen*** aus dem Script
  - Zugriff auf ***Script-Variablen*** aus dem Interpreter
  - Aufruf von ***Script-Prozeduren*** aus dem Display-Interpreter (z.B. aus Reaktionsmethoden)
  - Aufruf von ***Script-Funktionen*** aus Anzeigeseiten (per Backslash-Sequenz im Format-String)

4.5.3 Zugriff auf Anzeige-Variablen aus der Script-Sprache

In vielen Fällen werden Scripte dazu verwendet, um 'besondere' Werte für die Anzeige zu berechnen. Zu dem Zweck hat das Script während der Laufzeit sowohl Lese- als auch Schreibzugriff auf alle auf der Registerkarte 'Variablen' definierte *Display-Variablen*. Um dem Script-Compiler mitzuteilen, daß es sich um eine Display-Variable (und nicht um eine im Script deklarierte Script-Variable) handelt, wird der Prefix "**`display.`**" vor den Namen der Display-Variablen gestellt. Das Laufzeitsystem stellt dann sicher, daß der Wert dieser Variablen nicht in dem Moment modifiziert wird, in dem z.B. die Anzeige aktualisiert, oder die vom Display-Interpreter abgearbeiteten 'Anzeige-Events' abgearbeitet wird.
Beispiel (mit 'Oeldruck' als *Display-Variable*, die mit einem 'CANdb'-Signal verbunden ist, wodurch 'Oeldruck' auch zu einer *Netzwerk-Variablen* wurde) :

```
if ( ! display.Oeldruck.va ) then
   print( "Öldruck wird nicht übertragen !");
else if ( display.Oeldruck < 1.2345 ) then
   print( "Öldruck zu niedrig !");
endif;
```

Hinweis: Aus dem oben erklärten Grund könnte der Zugriff auf Display-Variablen die Abarbeitung des Script-Programms abbremsen (weil das Script eventuell auf die Aktualisierung der Anzeige "warten" muss).
Verwenden Sie Display-Variablen im Script daher nicht als Schleifenzähler und dergleichen, sondern nur für den einen Zweck: für die *'Anzeige'* !
Siehe auch:

- Steuerung der programmierbaren Anzeigeseiten per Script (Seitenumschaltung, etc etc.)
- Details zur Interaktion zwischen *Script* und *Display-Applikation* :
  - Zugriff auf ***Display-Variablen*** ("Werte") aus dem Script
  - display.GetVarDefinition (Zugriff auf *die Definition* einer Display-Variablen)

- Zugriff auf ***Anzeige-Elemente*** (auf der aktuellen Display-Seite) per Script
- Zugriff auf ***Script-Variablen*** aus dem Interpreter
- Aufruf von ***Script-Funktionen*** aus Backslash-Sequenzen in Display-Strings (zum Ersatz von Texten auf der Anzeige, per Format-String)
- Event-Handling in der Script-Sprache (als Ersatz für die alten 'Event-Definifionen' in den programmierbaren Anzeigeseiten)

- Schlüsselwörter
- Beispiele
- Übersicht (dieses Dokuments)

4.6 Arrays

Sowohl Variablen als auch Konstanten können als Arrays deklariert werden. Die Syntax, und die Bedeutung von Array-Indizes ist ähnlich wie in der Programmiersprache "C". Wie in "C" existiert kein Schlüsselwort namens 'array'. Wie in "C" erkennt man Arrays (und Array-ähnliche Strukturen) an **eckigen Klammern**. Diese werden in der Deklaration für die Array-Größen (Dimensionen) verwendet, und -später, beim Zugriff auf einzelne Elemente- für die Array-Indizes.

Gültige Array-Indizes liegen zwischen NULL (=erstes Element im Array) und <Arraygröße (.size) MINUS EINS> !

Beispiel für ein dreidimensionales Array, welches zum besseren Verständnis in "Seiten" (pages,z), "Zeilen" (lines,y), und "Spalten" (columns,x) organisiert sein möge:

```
var
  int ThreeDimArray[10][20][30]; // [z][y][x]
endvar
   ...
  z := 1; // "page" index, valid: 0..9
  y := 2; // "line of page", valid: 0..19
  x := 3; // "column of line", valid: 0..29
  ThreeDimArray[z][y][x] := 1234;
```

Hinweise und Tipps zur Verwendung von Arrays:

- Die maximale Anzahl von Dimensionen pro Array beträgt DREI.
  Vierdimensionale Arrays sind nicht möglich.
  Arrays, die Arrays als Elemente enthalten, sind ebenfalls nicht möglich.
  Arrays, die Strukturen als Elemente enthalten, sind allerdings möglich (und werden oft verwendet).
- Bestimmte Datentypen in Arrays (z.B. Strings) können problematisch sein. Ein String besteht intern lediglich aus einem 4 Byte großen Pointer auf eine Zeichenkette. Die Zeichenkette selbst wird in einem anderen Speicherbereich ("Blockspeicher") angelegt, allerdings erst wenn dem String eine 'nicht leere' Zeichenkette zugewiesen wird. Ein Array aus 100 Strings belegt daher scheinbar nur ca. 400 Bytes im Speicher, allerdings nur solange alle Zeichenketten "leer" sind (und daher keinen zusätzlichen Speicher belegen). Aus dem gleichen Grund kann ein komplettes Array auch nicht 'in einem Block' kopiert werden (denn dann würden nur die Pointer, d.h. die Adressen der Zeichenketten dupliziert, nicht aber die Zeichenketten selbst. **Ob der Speicher für eine bestimmte Anwendung ausreicht, stellt sich in diesem Fall daher nicht 'sofort' nach dem Compilieren des Scripts heraus, sondern erst nach einer ausreichenden Programmlaufzeit** (wenn alle Strings 'angefüllt' sind, indem sie z.B. im Initialisierungsteil des Scripts aus einer Datei gelesen wurden, oder mit einem geeigneten Protokoll über ein Netzwerk empfangen wurden) **!**
- Partielle Array-Referenzen (wie in C) sind im Script nicht möglich.
  Im obigen Beispiel (dreidimensionales Array) kann daher keine "ganze Seite" im Array kopiert werden:

```
ThreeDimArray[z] := ThreeDimArray[z+1]
```
ist (noch?) nicht möglich (Stand 2011-10-06).

- Der Inhalt eines kompletten Arrays kann während der Laufzeit im Programmiertool inspiziert werden.
  Geben Sie dazu den Namen des Arrays (als globale Script-Variable) ohne Array-Indizes in der Watch-Liste ein.
- Der Typ *eines einzelnen Elements* kann zur Laufzeit per <Array-Name>.type abgefragt werden. Diese Eigenschaft liefert z.B. den Wert dtByte, wenn es sich um ein Byte-Array handelt.
- Ein Array aus BYTES kann als Speicher für beliebige 'binäre' Datenblöcke verwendet werden. Das append()-Kommando dient dann zum Anhängen von Zeichenketten an das Byte-Array (bis auf das einen String abschliessende Null-Byte, welches beim Lesen einer Zeichenkette nicht als Teil derselben angesehen wird).
- Zum Anhängen von Array-Elementen **kann** eine explizite Indexvariable mit dem aus "C" bekannten '++' - Operator zum Inkrementieren verwendet werden, z.B.:
```
TxBuffer[TxByteIndex++] := 0x00; // append a ZERO BYTE to the array
```
- Zum Anhängen von Array-Elementen **ohne** Indexvariable verwenden Sie append(<Ziel-Array>, <Daten>), wodurch die Array-Eigenschaft .len ("aktuelle Länge") bis zur maximalen Kapazität (.size.size alias .cap) erhöht wird, z.B.:
```
append( TxBuffer, (byte)0x00); // append a ZERO BYTE to the array
```
- Zur effizienten Übergabe von Arrays an Unterprogramme *ohne Kopieren der Elemente* lesen Sie bitte die Hinweise in Kapitel 4.6.3, "Übergabe von Array-Referenzen".

Übersicht aller Array-Eigenschaften und -Methoden (Schlüsselwörter direkt nach dem Array-Namen)

```
.cap (Alias für .size)
.fifo_mode   .fifo_head   .fifo_tail
.len
.size .type  .t_sample    .unix_time
```

5.5 4.6.1 Maximale 'Größe' (size) und momentan verwendete 'Länge' (len) eines Arrays

Die *maximale* Anzahl von Array-Elementen (auch "Kapazität" genannt) kann mit der Member-Funktion 'size' abgefragt werden. Bei mehrdimensionalen Arrays wird dabei die Dimension in Klammern angegeben, z.B.:

```
  maxPages := ThreeDimArray.size(0); // how many "pages" (first array dimension) ?
  maxLines := ThreeDimArray.size(1); // how many "lines per page" (second array dimension) ?
  maxColumns:=ThreeDimArray.size(2); // how many "columns per line" (third array dimension) ?
```

Um die Mehrdeutigkeit von 'size' zu vermeiden, kann ähnlich wie in der Programmiersprache "Go" statt '.size' auch die Member-Funktion '.cap' ('Kapazität', gemessen in Array-Elementen, nicht Bytes) verwendet werden.

Die *momentan verwendete* Anzahl von Array-Elementen ("Länge") kann mit der Member-Funktion 'len' abgefragt werden:

```
  numPages := ThreeDimArray.len(0); // used number of "pages"
(first array dimension) ?
  numLines := ThreeDimArray.len(1); // used number of "lines per
page" (second array dimension) ?
  numColumns:=ThreeDimArray.len(2); // used number of "columns per
line" (third array dimension) ?
```

Fehlt die Angabe der Dimension direkt nach dem Schlüsselwort ".len" bzw. ".size" / ".cap", wird die Länge bzw. Größe in der *ersten* Dimension ermittelt. Bei eindimensionalen Arrays kann die geklammerte Angabe der Dimension daher entfallen, z.B.:

```
  currentLength := TxBuffer.len; // number of array elements
currently "in use"
```

Ähnlich wie in "Go" (bei Slices) wird auch hier zwischen der "Kapazität" (.cap) und "Länge" (.len) unterschieden. Solange keine Zuweisung an Elemente eines Arrays erfolgt ist, ist zwar eine "Kapazität" vorhanden, aber die "verwendete Länge" beträgt Null. Mit der Funktion append() können neue Elemente an das Array angefügt werden, wobei die "Länge" bis zur "Kapazität" (.size alias .cap) anwachsen kann.

Durch explizites 'Null-Setzen' der Länge (z.B. `TxBuffer.len := 0`) kann der alte Inhalt des Arrays (zumindest für 'append()' und Co) effizient 'entsorgt' werden, ohne die eigentlichen Array-Elemente zu modifizieren. Eine 'clear'-Methode für Arrays erübrigt sich dadurch.
Andere Funktionen (z.B. file.write) verwenden **.len**, wenn Daten aus dem Array verarbeitet werden sollen.
Im Gegensatz zu Slices in "Go" kann ein Array in der Script-Sprache nicht dynamisch vergrößert werden. Arrays haben eine 'feste' Maximalgröße, um den Speicherbedarf bereits beim Compilieren des Scriptes ermitteln zu können (statt erst zur Laufzeit im Target).

5.6 4.6.2 Weitere Elemente im Array-Header

Neben den im vorgehenden Kapitel vorgestellten Eigenschaften 'Größe' und 'momentan belegte Länge' besitzt jedes Array weitere Elemente, die in den folgenden Unterkapiteln vorgestellt werden. Diese Komponenten im Array-Header werden z.B. beim Auslesen von Abtastwerten aus der DAQ (Datenerfassungseinheit), und beim Zeichnen von Y(t)-Diagrammen mit Arrays als 'Quelle' verwendet.
Zum Zugriff auf diese Elemente wird dem Namen des Arrays ein Punkt und der Name des Elements angehängt, z.B.:

```
dblUnixTimeOfLastSample := fltFftin.unix_time;
dblSamplingRate_Hz  := 1.0 / fltFftin.t_sample;
```

5.6.1 4.6.2.1 Arrays als FIFO (Ringspeicher mit 'first in, first out'-Prinzip)

ToDo: Beschreibung von .fifo_mode, .fifo_head, .fifo_tail hier einfügen.

5.6.2 4.6.2.2 Abtastintervall und Zeitstempel für das *erste* Element im Array

`<array-name>.t_sample`

Abtastintervall *in Sekunden*. Wird *intern* als Fließkommazahl mit einfacher Präzision (float) gespeichert. Das Abtastintervall ist der Kehrwert der in der digitalen Signalverarbeitung gebräuchlicheren Abtastrate (sampling frequency) bezeichnet.
Nach dem Durchlaufen einer (Vorwärts-)FFT wird in dieser Komponente die FFT-Bin-Breite in Hertz abgelegt.

`<array-name>.unix_time`

Zeitstempel des *ersten ("ältesten")* im Array gespeicherten Abtastwerts (an Array-Index Null).
Als Einheit wird (wie bei system.unix_time) die Anzahl seit dem 1. Januar 1970, um 00:00:00.0 Uhr vergangener *Sekunden* verwendet. Da dieser Wert *intern* als 64-Bit-Integer in Mikrosekunden gespeichert wird, muss er ebenfalls ein Vielfaches von 1e-6 sein. Diese Auflösung sollte für alle mit dem MKT-View erzielbaren Abtastraten (die im Bereich einiger kHz liegen) ausreichen. Beim Auslesen eines Kanalspeichers per daq.read_channel() wird dieser Zeitstempel automatisch in den Array-Header übernommen.

Die zum Zeitpunkt der Erstellung dieser Beschreibung noch unvollendeten DSP-Funktionen (für das MKT-View IV) passen den o.g. Zeitstempel automatisch an, z.B. um die Gruppenlaufzeit eines Tiefpaßfilters für die später folgende mehrkanalige Anzeige als Y(t)-Diagramm auszugleichen.

Siehe auch (Zweck von Abtastintervall und Zeitstempeln im Array-Header):
Auslesen von Abtastwerten aus der Datenerfassungseinheit

5.7 4.6.3 Übergabe von Array-Referenzen (um Kopieren zu vermeiden)

Bei der Übergabe eines Arrays (oder eines Teils davon) an Unterprogramme (d.h. beim Aufruf benutzerdefinierter Prozeduren oder Funktionen) sollten Arrays aus Effizienzgründen per Referenz übergeben werden. Statt die Array- Elemente zu kopieren wird so nur ein Pointer (als Referenz) vom Aufrufer an die aufgerufene Funktion übergeben.
Um in der Funktionsdeklaration zu verdeutlichen, dass ein bestimmter Parameter eine Array-Referenz (*ohne Kopieren der Elemente*) sein soll, wird nach dem Typ des Arrays und dem formalen Parameternamen eine leere Array-Klammer angehängt (genaugenommen: ein leeres Klammernpaar pro Dimension). Der Compiler kennt daran den Typ des Arrays, was für die Syntax-Prüfung ausreicht. Innerhalb der aufgerufenen Funktion kann die so übergebene Referenz (im folgenden Beispiel "SrcBitmap**[]**") wie ein Array verwendet werden:

```
func DrawSprite( tCanvas ptr pDestCanvas, byte SrcBitmap[] )
     pDestCanvas.drawImage( SrcBitmap );
endfunc;
```

Da in diesem Beispiel keine Array-Elemente kopiert werden, sondern lediglich eine Referenz (u.A. mit der Adresse des Arrays) übergeben wird, wird dafür nur wenig Rechenleistung verbraucht. Zu

beachten ist allerdings (wie bei jeder Übergabe mit 'call-by-reference'), dass das Unterprogramm den Inhalt des übergebenen Arrays modifizieren kann.

5.8 4.6.4 Beispiele für die Verwendung von Arrays

Umfangreichere Beispiele zur Verwendung von Arrays finden Sie im Demoprogrammen "QuadBlocks" und "MacPan" .
Ein weiteres Beispiel, in dem ein eindimensionales Script-Array als Datenquelle für Diagramme verwendet wird, finden Sie in der Applikation programs/DAQ_Test.cvt. In der Applikation script_demos/diagrams.cvt werden Arrays als Speicher für Polygon-Koordinaten in Diagrammen verwendet.

Zum effizienten Umwandeln von Byte-Arrays in Zeichenketten bietet die 'Konstruktor'-Funktion string(<byte-array>, <start index>, <length>) die Möglichkeit, auch ohne slices einen 'Ausschnitt' des Arrays in eine Zeichenkette umzuwandeln. Dies wird z.B. benötigt, wenn Daten aus einem Empfangspuffer (deklariert als Byte-Array) als Zeichenkette weiterverarbeitet werden sollen.

6 4.7 Operatoren

In der Script-Sprache werden ähnliche numerische Operatoren wie im Display-Interpreter verwendet (wobei die interne Funktion allerdings stark abweicht, Details bei Interesse im Kapitel Bytecode). Die folgenden Operatoren sind in der Script-Sprache implementiert :

Operator	Alias	Priorität	Erläuterungen
^	POW	5 (höchste)	reserviert für 'A hoch B' (keine bitweises EXOR!)
*		4	Multiplikation
/		4	Division
%	MOD	4	Modulo (berechnet den Integer-Divisionsrest). Für Fliesskommawerte verwenden Sie Math.fmod .
+		3	Addition
-		3	Subtraktion
<<	SHL	3(?)	bitweise links schieben
>>	SHR	3(?)	bitweise rechts schieben
==	= (*)	2	Test auf 'Gleichheit'
!=	<>	2	Test auf 'Ungleichheit'
< , > , ..		2	andere Vergleichsoperatoren
||	or	1 (niedrigste)	Logisches (boolsches) ODER
&&	and	1	Logisches (boolsches) UND
|	bit_or	1	bitweise ODER-Verknüpfung
&	bit_and	1	bitweise UND-Verknüpfung (mit linkem Operanden)
exor		1	bitweises (!) EXKLUSIV-ODER
!	not	1	boolsche Negation
~	bit_not	1	bitweises NOT (Einerkomplement)
addr(variable)	& (vorgestellt)	1	Adresse einer Variablen ermitteln
(data type)		1	explizite Typumwandlung (typecast)
++ (Suffix)		1	Post-Increment
-- (Suffix)		1	Post-Decrement
:=			Zuweisung, z.B. A := B; // kopiere 'B' nach 'A' (*)

(*) Verwenden Sie "=" weder als **Vergleichsoperator**, noch als **Zuweisungsoperator**.
Im Interesse einer eindeutigen Unterscheidung von VERGLEICH und ZUWEISUNG :
Verwenden Sie ':=' als Zuweisung des Wertes (rechts vom Operator) an eine Variable (links vom Operator), wie in PASCAL.
Verwenden Sie '==' als Test auf Gleichkeit, wie in der Programmiersprache "C" (und, viele Jahre später, Java).
Ohne diese Notation müsste der Compiler an manchen Stellen raten, ob '=' eine Zuweisung oder 'Test auf Gleichheit' sein soll.

Ferner:

Verwenden Sie Klammern, wenn Ihnen die Priorität der Operatoren unklar ist.
Überlassen Sie nichts dem Zufall !
Speziell bei den bitweisen und boolschen "AND" und "ODER" - Operatoren gibt es (in Gegensatz zu "C") noch keine klar definierte Prioritäten. Beispiel zur Verwendung von Klammern (um die Reihenfolge bei der Verknüpfung eindeutig zu klären) :

```
Warning :=  EngineRunning and (  ( WaterTemp < 5 ) or ( WaterTemp > 95 )  ) ;
```

(In diesem Beispiel wird das Flag 'Warning' gesetzt, wenn die Wassertemperatur unter 5 Grad, oder über 95 Grad liegt, aber nur wenn auch der Motor läuft)

Siehe auch: Schlüsselwörter, Übersicht .

6.1 4.7.1 Der 'Address-Operator' (Prefix &)

Der '&'-Operator vor dem Namen einer Variablen liefert (ähnlich wie in 'C') deren *Adresse*.
Ähnliches kann (für Einsteiger besser lesbar) auch mit dem 'addr'-Operator bewirkt werden, dessen Syntax eher an den Aufruf einer Funktion erinnert (obwohl der erzeugte Bytecode in beiden Fällen identisch ist).
Ist z.B. 'MyVariable' der Name einer lokalen oder globalen Script-Variablen, dann liefert der Ausdruck **&MyVariable** wie auch **addr(MyVariable)** die Speicheradresse der Variablen 'MyVariable'.
Dieser Operator wird gelegentlich bei Funktionsaufrufen mit Parameterübergabe 'per Referenz' (call-by-reference), statt wie meistens 'per Wert' (call-by-value) verwendet.
Beispiele für die Parameterübergabe 'per Referenz' finden Sie in den Funktionen inet.recv() und file.read() .

Wir empfehlen für den Einsteiger die Verwendung des intuitiv 'begreifbaren' Operators 'addr()'.
Fortgeschrittene Anwender bevorzugen das "C"-kompatible, dem Variablennamen vorangestellte Ampersand (**&**).

Hinweis:

Bei der Übergabe von *Arrays* als Funktionsargument wird kein Address-Operator benötigt, denn Arrays werden *grundsätzlich* nur als Referenz übergeben, um keine Rechenzeit für das Kopieren des Arrays zu verschwenden.

6.2 4.7.2 Inkrement- und Dekrement-Operator ('++', '--')

Der '++'-Operator wird *rechts* neben dem Namen einer Variablen (innerhalb eines numerischen Ausdrucks) verwendet, um den Wert der Variablen um Eins zu erhöhen, *nachdem* der Wert der Variablen zur Berechnung des Ausdrucks gelesen wurde.
Die Syntax entspricht dem 'Post-Inkrement-Operator' in der Programmiersprache "C".
Entsprechend dient '--' direkt nach dem Variablennamen zum **Dekrementieren** ('Post-Dekrement').
Beispiel:

```
j := i++;  // copy 'i' to 'j', then increment 'i' by one
```

Der obige Code enspricht weitgehen dem Folgenden, ist aber kürzer und schneller:

```
j := i;   // copy 'i' to 'j'
i := i+1; // increment 'i' by one
```

Der '++'-Operator wird gerne zum 'elementweisen' Füllen von Arrays verwendet. Da die Index-Zählung bei Null(!) beginnt, wird die Index-Variable am Anfang auf Null gesetzt, und wie im folgenden Beispiel *nach* dem Zugriff um Eins erhöht (darum 'Post-Inkrement').
Ein komplettes Beispiel finden Sie in der Vorstellung des append()-Kommandos.

```
TxBuffer[TxByteIndex++] := 0x00;   // append another ZERO BYTE
```

In diesem Beispiel ist 'TxByteIndex' der Array-Index zum sequenziellen Füllen des Arrays 'TxBuffer'. Bei jedem neu angehängten Element (hier: Byte) wird 'TxByteIndex' um Eins erhöht. Da beim 'Post-Inkrement'-Operator der Wert von 'TxByteIndex' erst *nach* der Indizierung erhöht wird, wird das erste Array-Element korrekterweise an Index 0 (TxByteIndex=Null) eingetragen.

Zurück zur Übersicht der Operatoren .

7 4.8 Anwenderdefinierte Funktionen und Prozeduren

Anwenderdefinierte Funktionen und Prozeduren ersetzen (seit Oktober 2010) die aus BASIC bekannten Befehle 'gosub-return'. Damit ist eine wesentlich einfachere und überschaubarere Programmstruktur realisierbar, mit Zerlegung des Programmes in kleinere Einheiten (Funktionen und Prozeduren). Diese können sich gegenseitig, mit einer wohldefinierten Parameterübergabe (Parameterliste) aufrufen. Auch rekursive Algorithmen sind, dank lokaler Variablen auf dem Stack, möglich. Für die effiziente Übergabe von *Arrays* an Funktionen oder Prozeduren (ohne Kopieren bzw. Duplizieren) gelten die in Kapitel 4.6 aufgeführten Regeln.
Im Gegensatz zu Funktionen liefern Prozeduren keinen Wert (als "Funktionsergebnis") zurück, und können daher nicht als Teil in numerischen Ausdrücken verwendet werden.

7.1 4.8.1 Anwenderdefinierte Prozeduren

Hier ein einfaches Beispiel für eine anwenderdefinierte, *rekursive* (*) Prozedur, mit der eine Dezimalzahl auf dem Bildschirm ausgegeben wird. Quelle: Beispielprogramm TScreenTest).
Beachten Sie die Einrückung zwischen den Schlüsselwörtern 'proc' und 'endproc'. Das Einrücken ist für den Compiler ohne Bedeutung, gehört aber zum guten Stil im Script-Quelltext.

```
//------------------------------------------------------
proc PrintDecimal( int i )
  // Einfache REKURSIVE Prozedur zum Ausgeben einer Dezimalzahl .
  if( i>10 ) then
      PrintDecimal( i / 10 );    // höherwertige Ziffern ausgeben,
rekursiv
  endif;
  print( chr( 48 + (i % 10) ) ); // niederwertige Ziffer ("Einer")
ausgeben
endproc; // end PrintDecimal()
```

Beispiel zum Aufruf der oben definierten Prozedur (Details zur Rekursion folgen später) :

```
N := 123456;
PrintDecimal( N );  // call user-defined procedure
```

Intern wird der Wert von 'N' kurz vor dem Prozeduraufruf auf dem Stack abgelegt ("Push"). Die Prozedur verwendet den Wert (i) dann wie eine lokale Variable (d.h. 'i' existiert nur innerhalb der Prozedur). Jeder weitere (rekursive) Aufruf der Prozedur alloziert weiteren Speicher auf dem Stack, der erst wieder freigegeben wird, wenn die Prozedur zum Aufrufer zurückkehrt.
Bei Interesse an den Details: Die virtuelle Maschine, die den Bytecode abarbeitet, verwendet das 'BP' - Register (base pointer) um auf Funktions- bzw Prozedurparameter und auf lokale Variablen zuzugreifen. Der aktuelle Inhalt von BP wird beim Aufruf ebenfalls auf dem Stack gerettet, so daß der tatsächliche Stackbedarf bei jeder Rekursion noch etwas höher ist, als es die obige (vereinfachte) Beschreibung erwarten lässt.

Entgegen der Regel '*genau EINE Quelltextzeile pro Anweisung*' kann sich der the Kopf eine Prozedur oder Funktion (wie ein Prototyp in "C") über mehr als eine Quelltext-Zeile erstrecken. Beispiel (aus dem Demoprogramm 'TimeTest') :

```
//------------------------------------------------------
proc SplitUnixSeconds(
```

```
        in int unix_seconds, // one input, six outputs...
        out int year, out int month, out int day,
        out int hour, out int min, out int sec )
  // Procedure to split 'Unix Seconds' into
  // year (1970..2038), month (1..12), day-of-month (1..31),
  // hour (0..23 ! ), minute (0..59), and second (0..59) .
```

7.2 4.8.2 Anwenderdefinierte Funktionen

Anwenderdefinierte Funktionen ähneln den Prozeduren. Der wesentliche Unterschied ist, daß eine Funktion einen direkten "Rückgabewert" (Funktionsergebnis, aka 'return value') zum Aufrufer zurückgibt.

Einfaches Beispiel: Anwenderdefinierte *Funktion* zur Addition von zwei Integer-Werte

```
//-------------------------------------------------------
func Sum2( int a, int b) // adds two integers, returns the sum
   return a+b;
endfunc;

Summe := Sum2( 1, 2 );   // invoke the user-defined function
```

Beachten Sie, daß -im Gegensatz z.B. zu "C" oder Pascal- der Typ des Rückgabewertes nicht im Funktionskopf deklariert wird !
Grund: Script-Funktionen können, ähnlich wie in JavaScript (nicht Java!) *verschiedene* Datentypen als Funktionsergebnis zurückgeben. Der Aufrufer kann -wenn nötig- den *Typ* des Rückgabewertes mit Hilfe des typeof()-Operators ermitteln.

In benutzerdefinierten Funktionen dient der Befehl '**`return`**', gefolgt zum zurückzugebenden Wert, um die Funktion an dieser Stelle zu beenden und (mit dem "Wert") zum Aufrufer zurückzukehren. In Event-Handlern (die in Form von benutzerdefinierten Funktionen definiert werden *können*) signalisiert der Return-Wert, ob das Ereignis im Handler 'komplett' bearbeitet wurde (`return TRUE`) oder auch dem Default-Handler zugeführt werden soll (`return FALSE`).

Erreicht der Programmzähler das Ende der Funktion ('endfunc') ohne vorher auf eine 'return'-Anweisung gestoßen zu sein, dann liefert die Funktion den Wert 0 (Null) mit dem Datentyp Ganzzahl (int) zurück.

Funktionen mit bestimmten 'speziellen Namen' können vom System auch als Event-Handler aufgerufen werden. In dem speziellen Fall wird der Programmablauf durch den Aufruf des Event-Handlers **unterbrochen (interrupt)**, und der Rückgabewert der Funktion bestimmt, ob das Ereignis 'wie gewohnt' vom System verarbeitet werden soll oder nicht.

Sie finden zahlreiche Beispiele für benutzerdefinierte Prozeduren, Funktionen, und Event-Handler in den Beispielen für die Script-Sprache.

7.3 4.8.3 Aufruf von Script-Funktionen per Backslash-Sequenz aus den programmierbaren Anzeigeseiten

Seit August 2011 können benutzerdefinierte Funktionen (d.h. im Script implementierte) Funktionen auch direkt aus den programmierbaren Anzeigeseiten aufgerufen werden, um z.B. den Text im

Format-String eines Anzeige-Elements während der Laufzeit zu *ersetzen*. Für den Aufruf aus dem Display-Interpreter muss der Name der Script-Funktion, gefolgt von der Argumentenliste, in einer Backslash-Sequenz im Format-String für die Anzeige eingebettet sein. Damit der Display-Interpreter erkennt, daß aus dem Format-String das Script aufgerufen werden soll, muss vor dem Namen der Script-Funktion die Sequenz '**\script**' vorangestellt werden.

Syntax (im *Format-String* einer Display-Zeilen-Definition):

\script.<*function_name*>**(**<*arguments*>**)**

Darin ist <function_name> der Name einer anwenderdefinierten Script-Funktion; und <arguments> sind die Argumente (an die Funktion übergebene Parameter oder Variablen-Werte) - siehe folgendes Beispiel.

Beispiel (im Format-String der Definition eines Elementes auf einer programmierbaren Anzeigeseite):

\script.GetText(123)

Darin ist GetText der Name einer benutzerdefinierten Funktion (implementiert in der Script-Sprache), die als Funktionsargument einen Integer-Wert erwartet (in diesem Beispiel eine 'Text-Referenz-Nummer', und die als Rückgabewert immer einen String liefert. Die maximale Länge des auf diese Weise vom Script in den Format-String kopierbaren Strings beträgt 1024 Zeichen (begrenzt durch die statischen String-Typen im Display-Interpreter; hat nichts mit der Script-Sprache zu tun). Dieses Beispiel stammt aus dem 'Multi-Language-Demo' (script_demos/MultiLanguageTest.cvt), und diente als Beispiel zum Einsatz der Script-Sprache zur 'Internationalisierung' der Display-Applikation, indem alle statischen Texte aus dem Anzeigeprogramm in das Script verlagert wurden (oder in eine per Script geladene Text-Datei) :

```
//-------------------------------------------------------------
func GetText( int ref_nr ) // User defined function .
  // Returns strings in different languages .
  // Input: ref_nr = reference number for a certain text,
  //        may run from zero to 9999 .
  // Currently selected language in variable 'language',
  //    which may be 10000(=LANG_ENG) or 20000(=LANG_GER),
etc.
  select( ref_nr + language )
....
    case #(123 + LANG_ENG): return "onehundredandtwentythree";
    case #(123 + LANG_GER): return "Einhundertdreiundzwanzig";
....
    else: return "missing translation, ref_nr="+itoa(ref_nr);
  endselect;
endfunc;
```

Hinweis: Wie bei Event-Handlern muss auch die wie oben beschrieben aus dem 'Format-String' aufgerufene Script-Funktion 'so schnell wie möglich' zum Aufrufer zurückkehren. Endlosschleifen sind daher tabu; denn andernfalls wäre das Gerät nicht mehr bedienbar (bzw. aus Sicht des Bedieners 'abgestürzt'). Das Laufzeitsystem ('watchdog') terminiert die aufgerufene Funktion, wenn

diese nicht schnell genug 'freiwillig' wieder zum Aufrufer zurückkehrt ! Im normalen Script-Kontext besteht diese Einschränkung nicht, denn dort sorgt das Laufzeitsystem dafür, dass selbst bei einer Endlosschleife in Script (ohne "wait") das Gerät bedienbar bleibt.
Sollte die im Kapitel Event-Handling spezifizierte Zeit für die aufgerufene Funktion nicht ausreichen, kann *im äußersten Notfall* der 'watchdog' auch im Script nachgeladen werden - was aber die bereits erwähnten Konsequenzen auf die Bedienbarkeit haben kann.

Siehe auch: Details zur Interaktion zwischen *Script* und *Display-Applikation* ...

- Zugriff auf ***Display-Variablen*** aus dem Script
- Zugriff auf ***Script-Variablen*** aus dem Display-Interpreter
- Aufruf von ***Script-Prozeduren*** aus dem Display-Interpreter

4.8.4 Aufruf eigener Script-Prozeduren aus dem *Display-Interpreter*

Um die Interaktion zwischen Display (programmierbarer Anzeigeseite) und Script zu vereinfachen, kann eine in der Script-Sprache geschriebene Prozedur (oder Funktion) direkt per Kommandozeile aus dem (UPT-)Display-Interpreter aufgerufen werden. Im folgenden Beispiel ruft die "Reaktionsmethode" eines Buttons bei der Betätigung desselben eine in der Script-Sprache geschriebene Prozedur auf:

Definition des Buttons (in der Display-Applikation) :

```
\btn($2,"German",60,script.SetLanguage(LANG_GER))
```

Darin ist **script.**SetLanguage der Aufruf einer Script-Prozedur (in diesem Beispiel zum 'Setzen der Sprache'), die bei der Definition des Buttons eingefügt bzw. aus einer Liste ausgewählt wurde.

Der Präfix '**`script.`**' teilt dem Display-Interpreter (!) mit, daß die danach folgende Prozedur in der Script-Sprache aufgerufen werden soll (d.h. Aufruf des kompilierten Scripts aus der interpretierten Display-Definition). Die an die Prozedur als Parameter übergebenen Werte (in diesem Beispiel 'LANG_GER') werden dazu auf dem Stack abgelegt. Der Aufruf selbst könnte das im Hintergrund laufende Script an beliebiger Stelle unterbrechen.

Beim Aufruf von Prozeduren (und Funktionen) aus dem Display-Interpreter gelten ähnliche Einschränkungen wie hier (für Event-Handler) beschrieben.
Die wichtigste Einschränkung: Im Gegensatz zum 'normalen' Script dürfen die aus dem Display-Interpreter aufgerufenen Prozeduren keinerlei Warteschleifen enthalten, und keine (Script-)Befehle aufrufen die den Aufrufer für mehr als ein paar Millisekunden blockieren würden. Grund: Im Gegensatz zum normalen Script-Kontext kann der Display-Interpreter nicht 'beliebig lange warten' - das System würde dadurch scheinbar abstürzen. Weitere Details zu dieser Problematik siehe system.feed_watchdog.

Ein komplettes Beispiel finden Sie in der Applikation 'Multi-Language-Test'.

Siehe auch: Weitere Informationen zur Interaktion zwischen *Script* und *Display-Applikation* :

- Zugriff auf ***Display-Variablen*** aus dem Script
- Zugriff auf ***Script-Variablen*** aus dem Interpreter

- Aufruf von ***Script-Funktionen*** aus Backslash-Sequenzen in Display-Strings (zum Ersatz von Texten auf der Anzeige, per Format-String)
- Event-Handling in der Script-Sprache (als Ersatz für die alten 'Event-Definifionen' in den programmierbaren Anzeigeseiten)

4.8.5 Input- und output- Argumente in der Parameterliste

Per default sind alle Parameter in der Argumentenliste einer Prozedur (oder Funktion) lediglich "Eingänge" (inputs), d.h. die Prozedur kann deren Wert lesen, aber (im Variablenraum des Aufrufers) nicht ändern. Nur Argumente mit dem Schlüsselwort 'out' (in der Parameterliste) können als "Ausgang" (output) die Variable des Aufrufers modifizieren. Argumente mit der Deklaration 'in' (input), bzw. ganz ohne 'in' oder 'out', können die Variablen des Aufrufers nicht beeinflussen. (Für Experten: Der Grund für diese Einschränkung ist, daß die Script-Sprache bislang weder 'Pointer' wie in "C", noch 'Call-By-Reference' wie in PASCAL unterstützt).

Beispiel :

```
proc AddInteger( in int A, int B, out int Result )
  Result := A + B;
endproc
...
var
  int N;
endvar;
AddInteger( 1,2, N ); // CALL of the procedure defined above. Output copied to 'N' when returning .
```

Das Schlüsselwort 'in' wurde lediglich als Pendant zu 'out' implementiert, um zu verdeutlichen daß das danach folgende Argument kein 'output' sondern (nur) ein 'input' ist (d.h. read-only aus Sicht der aufgerufenen Funktion oder Prozedur) .

Die Deklaration eines Arguments mit dem Schlüsselwort 'out' hat weder mit von von "C" bekannten Pointern, noch mit der von PASCAL bekannten Übergabe 'Call-by-Reference' zu tun ! In der Tat werden die als 'output' deklarierten Argumenter erst nach der **Rückkehr** *durch den Aufrufer* in die entsprechenden Variablen *zurückgeschrieben* ! Dies hat den Nebeneffekt, daß alle 'Outputs' erst "wirksam" werden (im Variablenraum des Aufrufers), wenn die aufgerufene Funktion zum Aufrufer zurückkehrt. Da dieses 'Zurückschreiben' in einer *nicht unterbrechbaren* Sequenz erfolgt, ist die Konsistenz aller 'Outputs' gewährleistet, falls die Funktion bzw Prozedur mehrere 'Outputs' in der Parameterliste enthält.

Siehe auch: Parameterübergabe per Pointer (empfehlenswert bei der Übergabe von 'großen' Strukturen, da für die Übergabe eines Pointers nur 4 Bytes kopiert werden müssen, statt Dutzende von Bytes im Speicher vom Aufrufer an die aufgerufene Funktion zu kopieren).

7.4 4.8.6 Rekursive Aufrufe

In diesem Zusammenhang bedeutet *Rekursion*, daß eine Prozedur (oder Funktion) sich selbst aufrufen darf -- solange der Stack dazu ausreicht. Bei jedem Aufruf ("Instanz") wird ein neuer Satz von lokalen Variablen (zu denen auch die Werte in der Argumentenliste zählen) auf dem Stack

alloziert, der erst dann wieder freigegeben wird, wenn die Instanz durch Rückkehr zum Aufrufer ihre Lebenszeit beendet.

Zur Verständnis von rekursiven Aufrufen betrachten wir noch einmal die bereits bekannte Prozedur PrintDecimal (Beispiel aus dem Kapitel "Anwenderdefinierte Prozeduren"):

```
proc PrintDecimal( int i )
  if( i>10 )
    then PrintDecimal( i / 10 );
  endif;
  print( chr( 48 + (i % 10) ) );
endproc;
```

Beim Aufruf PrintDecimal(123456) erhält die erste Instanz den Parameter i = 123456 (als lokale Variable auf dem Stack). Da 'i' in diesem Fall größer als Zehn ist, ruft sich die Funktion erneut selbst auf (= Rekursion !), diesmal mit dem Wert i = 12345 . Bei diesem zweiten Aufruf wird eine neue Instanz erzeugt, die weiteren Speicher im Stack belegt. Da 'i' weiterhin größer als Zehn ist, erfolgen weitere rekursive Aufrufe. Bei der letzten Instanz ist 'i' dann kleiner als Zehn (siehe Unten, mit i = 1 ist dann die höchstwertige Ziffer erreicht). Beginnend mit der höchstwertigen Ziffer werden dann die einzelnen Ziffern per "print" auf dem Bildschirm aufgerufen, wie in der folgenden Aufruf-Liste zu erkennen
(-> bedeutet "ruft auf", <- bedeutet "kehrt zum Aufrufer zurück") :

```
PrintDecimal( 123456 )
  -> PrintDecimal( 12345 )
      -> PrintDecimal( 1234 )
          -> PrintDecimal( 123 )
              -> PrintDecimal( 12 )
                  -> PrintDecimal( 1 )
                       (no further recursion; prints "1")
                  <- (procedure returns to the caller)
                 (caller now prints 12 modulo 10 = "2")
              <-
             (caller now prints 123 modulo 10 = "3")
          <-
         (caller now prints 1234 modulo 10 = "4")
      <-
    (caller now prints 12345 modulo 10 = "5")
  <-
 (first instance finally prints 123456 modulo 10 = "6")
<-
```

Rekursive Aufrufe sind nicht auf eine einelne Prozedur (oder Funktion) beschränkt. Funktionen und Prozeduren dürfen sich auch gegenseitig rekursiv aufrufen.
Algorithmen können mit Hilfe von Rekursion zwar elegant und einfach implementiert werden (wie im Beispiel 'Ausgabe einer Dezimalzahl'), der dabei benötigte Stack-Speicher ist allerdings schwer voraussagbar. Zu tiefe Rekursion kann daher zu Laufzeitfehlern, d.h. i.A. zum Abbruch des Scripts, führen.
Darum sind in vielen Fällen die im folgenden Kapitel vorgestellten Schleifen-Konstrukte die robustere Alternative.

8 4.9 Ablaufsteuerung in der Script-Sprache (program flow control)

Zur Steuerung des Programmflusses bietet die Script-Sprache die folgenden Konstrukte:

- if..then..else..endif
- for..to.. [step] .. next
- while .. endwile (prüft die Schleifen*wiederholungs*bedingung vor der Schleife, der Schleifenrumpf wird möglicherweise *nicht* abgearbeitet)
- repeat .. until (prüft die Schleifen*abbruch*bedingung nach dem Schleifenrumpf, diese Schleife wird daher *mindestens einmal* abgearbeitet)
- select .. case .. else .. endselect
- ~~goto (jumps to a label within the same function ... please forget about line numbers!)~~
- ~~gosub .. return (depreceated, use procedures wherever possible)~~
- stop : Beendet die Ausführung des "Main-Threads" im Script. Danach laufen nur noch die Event-Handler weiter.
- end_script : Optionale Textmarke. Erzeugt im Bytecode ein gleichnamiges Token, welches zur Laufzeit verhindert, dass der Programmzähler aus dem Initialisierungsteil des Scripts in das erste Unterprogramm 'durchläuft'. Fehlt diese Textmarke, dann wird der entsprechende Bytecode *automatisch* (vor der Implementierung der ersten Funktion oder Prozedur) vom Compiler eingefügt.

Hinweis: Wie in BASIC und IEC 61131 "Structured Text", und im Gegensatz zu Sprachen wie "C" und Java, wird bei fast allen Schlüsselwörtern nicht zwischen Groß- und Kleinschreibung unterschieden. Beachten Sie aber die Empfehlungen zum Quelltext-Stil (Einrücken, Zweck von Groß/Kleinschreibung, etc.).

Im weiteren Sinne dienen auch anwenderdefinierte Funktionen und Prozeduren zur Steuerung des Programmablaufes. Seit deren Einführung sollten Sie die Kommandos 'goto' und 'gosub / return' in neu erstellten Scripten nicht mehr verwenden. Diese veralteten (und aus der Basic-Steinzeit stammenden) Befehle existieren aber weiterhin, um die Kompatibilität mit alten Script-Programmen zu gewährleisten.

8.1 4.9.1 if-then-else-endif

"Wenn" <Bedingung>. "dann" <Anweisungen> "sonst" <andere Anweisungen> "Ende".

Einfaches Beispiel:

```
if A<100 then
    tue_irgendwas_wenn_A_kleiner_als_einhundert_ist ;
     ...
else
    tue_etwas_anderes_mit_A_groesser_oder_gleich_einhundert ;
     ...
endif;
```

Im Gegensatz zur Empfehlung '*immer genau ein Kommando pro Quelltext-Zeile*' kann/darf sich die Bedingung zwischen **if** und **then** über mehrere Zeilen erstrecken. Dies ist besonders bei komplexen

Bedingungen hilfreich, z.B. wenn mehrere Ausdrücke mit 'und' (and) und 'oder' (or) miteinander verknüpft werden.

Zur Vereinfachung von Mehrfach-Abfragen dient der Befehl 'elif' (else-if), mit dem wie im folgenden Beispiel weitere verschachtelte 'else', 'if', und 'endif'-Kommandos eingespart werden können:

```
if ( A < 0 ) then
    print( "Negativ !");
elif ( A==0 ) then
    print( "Null");
elif ( A==1 ) then
    print( "Eins");
elif (A >= 2) and ( A <= 3 ) then
    print( "Zwei bis Drei");
elif (A == 5) or (A == 7) then
    print( "Fünf oder Sieben");
else
    print( "Irgendein anderer Wert (",A,")" );
endif;
```

8.2 4.9.2 for-to-(step-)next

Implementiert eine Schleife mit einer Zähl-Variablen, die vom angegebenen Startwert bis zum angegebenen Endwert läuft. Optional kann dabei auch die Schrittweite (step) angegeben werden. Das Schlüsselwort 'next' markiert das Ende der in der Schleife wiederholt abgearbeiteten Anweisungen.

Einfaches Beispiel (mit Pseudo-Code):

```
for Loop:=1 to 100
  tue_irgendwas_einhundert_mal;
next Loop;
```

Die Angabe der Zählvariablen (im obigen Beispiel: 'Loop') nach dem Schlüsselwort 'next' ist nicht zwingend. Es wird aber empfohlen, die Zählvariable trotzdem *auch bei 'next'* anzugeben, da so die Lesbarkeit verbessert wird (speziell bei mehreren ineinander geschachtelten Schleifen), und der Compiler überprüfen kann ob zu jedem 'for' auch das vom Entwickler beabsichtigte 'next' existiert. Beim Compilieren mit der Option '#pragma strict' erfolgt eine Warnung, wenn nach dem Schlüsselwort 'next' der Name der Zählvariablen fehlt.

Optional kann die Schrittweite der Zählvariablen (hier: "I") angegeben werden. Dazu dient das Schlüsselwort STEP :

```
for Loop:=0 to 200 step 2
  do_something_a_couple_of_times
next Loop;
```

Soll die Zählvariable abwärts statt aufwärts zählen, muss der STEP-Wert negativ sein (der Compiler erkennt nicht automatisch die Schrittrichtung, weil Start- und Endwert zum Zeitpunkt der Übersetzung nicht bekannt sein könnten, denn Start- und Endwert könnten Variablen sein, statt (wie

in den einfachen Beispielen) Konstanten.
Weitere Beispiele finden Sie im Beispielprogramm namens 'LoopTest' .

8.3 4.9.3 while..endwhile

Syntax:

```
while <condition>
  <statements>;
endwhile;
```

Diese Schleife wird wiederholt, solange die am ***Anfang*** der Schleife geprüfte Bedingung (englisch 'condition') erfüllt ist (TRUE, bzw. ungleich Null).

Beispiel:

```
I:=0; // make sure 'I' starts at some defined value
while I<100
  I := I+1 ;
  some_other_statements ;
endwhile;  // .. or END_WHILE for IEC61131-similarity
```

Hinweis: Die Anweisungen innerhalb der while...endwhile - Schleife werden möglicherweise KEIN EINZIGES MAL ausgeführt (im Gegensatz zur repeat..until - Schleife).

Wird als Bedingung der Wert Eins (1) eingesetzt, dann bildet while(1) .. endwhile eine Endlosschleife. Dies wird wie im Beispiel aus der Einleitung für die 'Hauptschleife' im Script verwendet, wobei die Schleifengeschwindigkeit durch Aufruf von wait_ms(50) auf 50 Millisekunden pro Durchlauf begrenzt wird. Dadurch wird erreicht, dass das Script nicht einen Großteil der CPU-Leistung 'verbraucht'.

8.4 4.9.4 repeat..until

Syntax:

```
repeat
  <statements>;
until < stop_condition >;
```

Schleife mit einer Abbruch-Bedingung(!), die erst am Ende der Schleife getestet wird (nach dem Schlüsselwort "until").

Beispiel:

```
I:=0; // make sure 'I' starts at some defined value
repeat
  I := I+1;
  some_statements
until I>=100;
```

Hinweis: Unabhängig von der Bedingung werden die Anweisungen innerhalb der repeat-until-Schleife *mindestens einmal* ausgeführt !

8.5 4.9.5 goto

Unbedingter Sprung. Sollte unbedingt vermieden werden. Die Verwendung des 'goto'-Kommandos (als Sprungbefehl im Script) führt schnell zu einem unlesbaren, kaum noch zu pflegenden 'Spaghetti-Code'. Oder, wie es Niklaus Wirth formulierte, "GOTO Considered Harmful" . Um den Einsatz von 'goto' zu verübeln, funktioniert 'goto' nicht an beliebigen Stellen *innerhalb* von Funktionen und Prozeduren (und erst recht nicht, um Funktionen oder Prozeduren aus dem Hauptprogramm aufzurufen). Würde das Programm per 'goto' wild zwischen verschiedenen Funktionen oder Funktionen hin- und herschalten, entstünde nicht nur im Quelltext, sondern auch auf dem Stack ein großes Chaos.

Einfaches, und 'grade noch verzeihbares' Beispiel für das 'goto'-Kommando:

```
if divisor==0
  then GOTO ErrorHandler;
  else quot := dividend / divisor;
endif;
```

... some other code ***in the same subroutine*** ....

```
ErrorHandler: // we shouldn't get here...
  Info := "Something went wrong";
  stop;
```

8.6 4.9.6 gosub..return

Steinzeitlicher ("BASIC"-kompatibler) Unterprogramm-Aufruf ohne Parameterübergabe. Veraltet (s.U.), kann und soll durch benutzerdefinierte Prozeduren ersetzt werden.
"Gosub" funktioniert ähnlich wie das ähnlich grausame "goto", vorher wird allerdings die Adresse der nächsten auszuführenden Instruktion (durch den Aufrufer) auf dem Stack. gerettet. "Return" springt dann, mit Hilfe der vom Stack zurückgeholten Adresse, wieder zum Aufrufer zurück, und die Programmausführung wird dort fortgesetzt.

Hinweis: Im Gegensatz zu anwenderdefinierten Prozeduren haben per 'gosub' / 'return' implementierte Unterprogramme keinen eigenen Stack-Frame, und können daher keine lokalen Variablen verwenden. Auch aus diesem Grund sollte 'gosub' in neuen Script-Programmen nicht mehr verwendet werden. Die Kombination gosub/return existiert nur noch aus Kompatibilitätsgründen.

8.7 4.9.7 select..case..else..endselect

Mit dieser Konstruktion kann eine lange, verschachtelte if-then-else-Kombination ersetzt werden, allerdings nur wenn Integer- oder String-Konstanten mit *einer* Eingangsvariablen verglichen werden sollen. Jede (normale, einfache) "case"-Marke entspricht dem Vergleich der nach 'select' definierten Variablen mit einer Konstanten.
Mit "**case** <Wert1> **to** <Wert2>" kann auch ein kompletter *Wertebereich* definiert werden. Der nach dem Case folgende Code wird abgearbeitet, wenn der Wert der select-Variablen zwischen 'Wert1' und 'Wert2' liegt (inklusive 'Wert1' und 'Wert2'. Durch "case 4 to 6:" im folgenden Beispiel wird daher geprüft, ob der Wert 'X' 4, 5, oder 6 beträgt. Diese erweiterte case-Syntax ist seit September 2013 verfügbar.

Einfaches Beispiel (mit Pseudo-Code, "do_something" = "mach' irgendwas") :

```
X := can_rx_msg.id;
select X
   case 1 :      do_something_if_X_equals_one();
   case 2 :      do_something_if_X_equals_two();
   case 3 :      do_something_if_X_equals_three();
   case 4 to 6 : do_something_if_X_is_between_four_and_six();
   else   :      do_something_if_X_is_none_of_the_above();
endselect;
```

Bitte beachten Sie den *Doppelpunkt* am Ende der Case-Marke (aka "Label"). Andere Konstrukte werden wie üblich mit einem *Semikolon* abgeschlossen (hier z.B. nach dem endselect).
Statt 'else:' kann ähnlich wie in der Programmiersprache "C" oder "Go" seit 10/2018 auch das Schlüsselwort "default:" verwendet werden.
Im Gegensatz zu "C" ist hier kein "break"-Statement am Ende jedes Case-Blocks nötig. Das Script-Programm wird, abgesehen von den folgenden Ausnahmen, niemals von einem 'case' zum nächsten 'durchfallen' (kein "fall through" ohne 'break'). Die Ausnahmen, in denen das Programm von einer Case-Marke in die Anweisungen nach der nächsten Case-Marke springen kann, sind:

- Folgen zwei oder mehr Case-Marken direkt aufeinander, ***ohne 'ausführbare' Anweisungen dazwischen***, dann werden alle "leeren" case-Marken bis zum nächsten Anweisung (statement) übersprungen. Im folgenden Beispiel mit case 1, case 2, case 3 wird dies deutlich:

```
select X
  case 1 : // same handler for cases 1, 2 and 3 ...
  case 2 : // with no code between cases, 'fall through' as in
"C".
  case 3 : // since 2013, we could use "case 1 to 3" instead
     print("X = One,Two,or Three");
  case 4 :
     print("X = Four");
  else   :
     print("X is less than one, or larger than four");
endselect;
```

- Mit dem Schlüsselwort 'enter_next_case' wird das aus der Programmiersprache "C" bekannte "fall through" erzwungen, z.B:

```
for A:=0 to 6
   print("A=",A,": ");
   select A
      case 4 :    // "fall through" from one case to the next ..
      case 5 :    // .. only if there is nothing in between
         print("four or five.");
      case 3 :
         print("larger than two, ");
         enter_next_case; // aka 'fall through' to the code after the next
case label, as in "C"
     case 2 :
         print("larger than one, ");
         enter_next_case;
```

```
        case 0 to 1 :
            print("non-negative.");
        else    :
            print("negative, or larger than five.");
    endselect;
    print("\r\n"); // carriage return + new line (-> nächste Zeile)
next A;
```

Ausgabe:
A=0: non-negative.
A=1: non-negative.
A=2: larger than one, non-negative.
A=3: larger than two, larger than one, non-negative.
A=4: four or five.
A=5: four or five.
A=6: negative, or larger than five.

Soll eine case-Marke (mit leerem Anweisungsblock) 'nichts' tun, und auch nicht in den Anweisungsblock der *nachfolgenden* Case-Marke springen, kann die aus Programmiersprachen wie "C" bekannte break-Anweisung verwendet werden. Bei der Verwendung innerhalb eines select-endselect-Blocks springt **break** immer zum dazugehörigen **endselect**. Beispiel:

```
select X
  case 1 :
     break; // do nothing (do NOT enter the next case after
this empty block)
  case 2 :
     break; // do nothing as well
  case 3 :
     print("X = Three");
  case 4 :
     print("X = Four");
  else   :
     print("X is less than one, or larger than four");
endselect;
```

Für 'case'-Marken sind nicht nur Integer- sondern auch String-Konstanten erlaubt. Im Beispielprogramm zum Einlesen von INI-Dateien wird diese Möglichkeit genutzt, um nach den zeilenweise aus der Datei gelesenen 'Sektionen' und Schlüsselnamen zu verzweigen.

Weitere Beispiele für select-case finden Sie in den Testprogrammen 'ScriptTest1.cvt' und 'ScriptTest3.cvt'.
Beide sind im Archiv des Programmiertools enthalten (nach der Installation des Tools in programs/script_demos).

8.8 4.9.8 wait_ms .. wait_resume

Wartet für eine bestimmte Zeit (in Millisekunden), oder bis "irgendwas passiert".
Während dieser Wartezeit steht die CPU für andere Aufgaben zur Verfügung (z.B. Aktualisieren des Displays).

wait_ms(N)

Pausiert die 'normale' Abarbeitung des Scripts für die angegebene Anzahl von Millisekunden (N >= 1) .
In diesem Zustand läuft die Anzeige etwas schneller (d.h. höhere Page-Update-Rate), weil die gesamte verfügbare CPU-Leistung für das Display verwendet werden kann.
Ohne Aufruf einer Warte-Funktion aus der Hauptschleife (Endlosschleife) würden Script und UPT-Anzeige zwar auch 'parallel' nebeneinander laufen; die Script-Hauptschleife würde dann aber unnötig viel Rechenzeit verbrauchen. Empfohlene Wartezeiten (für die Hauptschleife) liegen bei 10 bis 50 Millisekunden.
VERWENDEN SIE 'wait_ms()' NIEMALS IN EVENT-HANDLERN und vergleichbaren 'Interrupt-ähnlichen' Funktionen !
Siehe auch: system.timestamp.

wait_ms(0) : Sonderfall zum Abarbeiten 'aller anstehenden Ereignisse'

In diesem Fall (mit "Null Millisekunden Wartezeit") werden lediglich die Event-Handler für alle momentan anstehenden Timer-Events aufgerufen, aber keine Aktualisierung des Displays (die je nach Komplexität der Anzeigeseite einige Dutzend Millisekunden dauern könnte).
Der Befehl wait_ms(0) dient ähnlich wie sched_yield() in Linux, oder Sleep(0) in der Windows-API, zum 'kooperativem Multitasking': Er gibt die CPU 'freiwillig' für andere Aufgaben ab.
Er wurde für den Aufruf aus 'lange dauernden' Script-Schleifen implementiert, um die Latenzzeit für Timer-Events zu verringern. Beispiel:

```
  while( ! file.eof(fh) )    // Wiederholen bis Dateiende...
     temp := file.read_line(fh); // nächste Zeile einlesen
     ImportFromCSV( temp ); // Zeile aus CSV-Datei verarbeiten (das kann
dauern..)
     wait_ms( 0 );          // Event-Handler laufen lassen (kooperatives
Multitasking)
  endwhile;
```

Hinweis: wait_ms(0) wartet nicht auf das *Aktualisieren des Bildschirms* (was etliche Millisekunden dauern könnte).

Wenn momentan *kein Timer-Event* zur Bearbeitung ansteht, und seit dem vorherigen Aufruf von wait_ms(0) weniger als 10 ms vergangen sind, dann macht wait_ms(0) *nichts*, sondern kehrt sofort zum Aufrufer zurück.

wait_resume

Beendet die per wait_ms im Hauptprogramm eingeleitete Pause, selbst wenn das dort spezifizierte Zeitintervall noch nicht abgelaufen ist.
Dieses Kommando wird typischerweise in Event-Handlern verwendet, um das in einer Endlosschleife wartende Hauptprogramm (main thread im Script) wieder 'aufzuwecken'.
In anderen Programmiersprachen existieren vergleichbare Funktionen, z.B. notify() in Java.

Siehe auch: system.timestamp, Inhalt, Schlüsselwörter, Kurzreferenz .

9 4.10 Weitere Funktionen und Kommandos

Zusätzlich zu den im vorhergehenen Kapitel beschriebenen Befehlen zur Ablaufsteuerung verfügt die Script-Sprache über spezielle, z.T. hardwareabhängige 'Sonderfunktionen'.
Die meisten dieser Sonderfunktionen werden in diesem Kapitel vorgestellt, einige weitere (z.B. übliche Funktionen wie random) sind dagegen nur in der Liste der Schlüsselwörter aufgeführt.

Siehe auch:

Inhalt , String-Funktionen, Datei-Ein-/Ausgabe-Funktionen, CAN/LIN-Bus-Funktionen, Zugriff auf Anzeige-Elemente (display.xyz), Steuern von Diagrammen (display.dia.xyz), Mehrzeilige Textfelder und simulierte Textbildschirme (print & Co), System-Funktionen, Funktionen zur Konvertierung von Datum und Uhrzeit, anwenderdefinierte Funktionen und Prozeduren .

9.1 4.10.1 Numerische Funktionen, "Mathematik", Digitale Signalverarbeitung

Die in diesem Abschnitt beschriebenen Funktionen und Prozeduren sind i.A. auf die Verarbeitung von Integer- und Fliesskommawerten beschränkt.
Datentypen wie 'byte', 'word', 'dword' werden vor der Verarbeitung nach Integer (d.h. Ganzzahl) konvertiert.
Siehe auch: Zahlen und numerische Ausdrücke .

9.1.1 4.10.1.1 Einfache numerische Funktionen

`limit(variable,min_value,max_value)`

Begrenzt den in <variable> gespeicherten Wert auf den angegebenen Bereich. Beispiel:

```
limit( variable, min_value, max_value ); // 'variable' begrenzen
```

Dies ist kürzer und effizienter als die folgende äquivalente Sequenz:

```
if ( variable <  min_value ) then
     variable := min_value;
endif;
if ( variable >  max_value ) then
     variable := max_value;
endif;
```

`random`

Liefert eine Pseudo-Zufallszahl zwischen Null und **<N> minus Eins**.

`Math.abs(x); Math.abs(x,y)`

Liefert den Absolutwert einer Zahl, bzw (mit zwei Argumenten) die Länge eines Vektors, nach der Formel abs(x,y) := sqrt(x^2 + y^2) .
Zusammen mit Math.atan2 dient Math.abs zur Umrechnung von kartesischen Koordinaten in Polarkoordinaten.

Math.atan2(y,x)

Berechnet den Arkustangens für alle vier Quadranten. Ergebnis im Bogenmaß (Einheit 'Radiant').
Im Gegensatz zum einfachen Arkustangens (mit y/x als Argument) ist die Funktion atan2 ohne umständliche 'Fallunterscheidungen' direkt für die Umrechnung von kartesischen Koordinaten in Polarkoordinaten geeignet.

Math.fmod(x,y)

Berechnet den Divisionsrest von x/y mit Fliesskomma-Werten (float oder double).
Für Ganzzahlen (int) verwenden Sie stattdessem den Modulo-Operator (%).

Math.log(x)

Berechnet den natürlichen Logarithmus von 'x', d.h. Logarithmus zur Basis 'e'.

Math.log10(x)

Berechnet den dekadischen Logarithmus von 'x', d.h. Logarithmus zur Basis 10.

Math.pow(base,exponent)

Berechnet 'base' hoch 'exponent'.
Beide Argumente und das Ergebnis verwenden den Datentyp float.
Beispiel (aus script_demos/MathTest.cvt): f := pow(2.0, 0.5); // actually the square root of two

Math.sqrt(x)

Liefert die Quadratwurzel von 'x'.
'x' muss eine nicht-negative Fliesskommazahl sein.

Math.sin(x)

Liefert den Sinus des angegebenen Arguments (Winkel 'x' im Bogenmaß)'.

Math.cos(x)

Liefert den Cosinus des angegebenen Arguments (Winkel 'x' im Bogenmaß)'.

9.1.2 4.10.1.2 Numerische Funktionen für die digitale Signalverarbeitung

Die in diesem Kapitel aufgeführten Funktionen sind nur bei Geräten mit Hardware-Fließkomma-Einheit vorhanden (z.B. MKT-View IV) bzw. nur dort wegen der für die digitale Signalverarbeitung erforderlichen Geschwindigkeit 'sinnvoll nutzbar'. Eine Übersicht über die Verfügbarkeit dieser Funktionen finden Sie in der Feature-Matrix (Spalte "Specials", "DSP functions").

Literaturempfehlung zum Thema FFT (und DSP) :
The Scientist and Engineer's Guide to Digital Signal Processing von Stephen W. Smith.
Darin finden Sie u.A. auch die Begründung, warum eine *reelle* FFT mit 1024 Abtastwerten als Eingang ein *komplexes* Spektrum mit 1025 (!) 'Frequenz-Eimern' (bins) erzeugt.

Math.cfft(input,output)

Schnelle Fourier-Transformation mit **komplexem** Ein- und Ausgang.
Nur für Geräte mit Hardware-Fließkomma-Einheit (z.B. MKT-View IV).
Ein- und Ausgang sind Arrays vom Datentyp 'float' mit 2^N Elementen (z.B. 256, 512, 1024, ...).
Im Gegensatz zur 'reellen FFT' müssen beide Arrays die gleiche Größe haben ! Beispiel:

```
   float  fltFFTin[1024];  // FFT input (complex time domain samples, aka
I/Q signal)
   float  fltFFTout[1024]; // FFT output (complex frequency bins)
     ...
   Math.cfft( fltFFTin, fltFFTout ); // komplexe FFT, vorwärts
```

Die komplexen Wertepaare werden wie folgt in Array vom Typ 'float' abgelegt:
Realanteil ersten Wertes, Imaginäranteil des ersten Wertes,
Realanteil zweiten Wertes, Imaginäranteil des zweiten Wertes, ...
Werden, wie im obigen Beispiel, Arrays mit je 1024 Elementen vom Typ float verwendet, dann transformiert die **komplexe** FFT 512 (!) Punkte *aus dem Zeitbereich* in den *Frequenzbereich*.
Wurde das Eingangssignal z.B. mit 10000 Samples/Sekunde (10 kHz) abgetastet, dann liefert die komplexe FFT ein Spektrum mit 512 komplexen 'Frequenz-Eimern' (FFT bins). Jeder 'Eimer' repräsentiert dann einen Frequenzbereich mit 10000 Hz / (2 * 512) = ca. 9.8 Hz Breite. Der 'Eimer' mit der höchsten darstellbaren Frequenz 'speichert' dann die komplexe Amlitude für ca. 512 * 9.765 Hz = 5000 Hz, was bei einer Abtastrate von 10000 Hz der Nyquist-Frequenz entspricht. Die 'Eimerbreite' wird im Header des Zielarrays (im o.g. Beispiel: fltFFTout.t_sample) abgespeichert, und ersetzt dort das Abtastintervall. Aus einer *Zeit in Sekunden* wird also eine *Frequenz in Hertz*, was u.A. auch bei der horizontalen Skalierung in "Y(t)" -Diagrammen berücksichtigt werden muss (die dann eigentlich "Y(f)"-Diagramme heissen müssten).
Siehe auch: Math.rfft (FFT mit reellem Eingang), Math.ComplexToMagnitudes, Datenerfassungseinheit (DAQ) .

Math.rfft(input,output)

Schnelle Fourier-Transformation mit **reellem** Eingang und **komplexem** Ausgang.
Nur für Geräte mit Hardware-Fließkomma-Einheit (z.B. MKT-View IV).
Als Eingang (input) kann ein Array von z.B. 1024 Abtastwerten im Zeitbereich verwendet werden.
Die Array-Größe muss eine Zweierpotenz sein (z.B. 256, 512, 1024 oder 2048 Punkte).
Der Ausgang (output) muss ein Array mit *mindestens der gleichen Kapazität* sein, denn auch eine 'reelle FFT' liefert ein *komplexes* Ergebnis (z.B. Spektrum), z.B.:

```
   float  fltFFTin[1024];    // FFT-Eingang (1024 Abtastwerte)
   float  fltFFTout[1024+2]; // FFT-Ausgang mit 513 komplexen "Frequenz-
Eimern"
     ...
   daq.read_channel( 7, fltFFTin );  // Eingang für die FFT als Array aus
der DAQ lesen
   Math.rfft( fltFFTin, fltFFTout ); // Vorwärts-FFT mit reellem Ein- und
komplexem Ausgang
```

Auch die 'relle' Vorwärts-FFT berechnet aus dem Abtastintervall (hier: fltFFTin.t_sample in *Sekunden*) und der FFT-Größe die FFT-Bin-Breite in *Hertz*, die mathematisch nicht ganz korrekt als 't_sample' im Header des Ziel-Arrays (hier: fltFFTout.t_sample in *Hz*) gespeichert wird.
Zu Anzeigezwecken kann das komplexe Spektrum mit der Funktion Math.ComplexToMagnitudes() logarithmiert werden, d.h. in Dezibel mit beliebigem Referenzwert konvertiert.
Siehe auch: Literaturempfehlung zum Thema FFT (und DSP) am Anfang dieses Kaptitels.

`Math.ComplexToMagnitudes(input,output,reference)`

Wandelt ein Array aus komplexen Wertepaaren (typischerweise der Ausgang einer FFT, d.h. komplexes Spektrum) in "Dezibel" um. Durch geeignete Wahl des Bezugswertes (reference) kann z.B. der Wertebereich vom A/D-Wandler, der Verstärkungsfaktor des Sensors, und der von der FFT-Länge abhängige 'Gewinn' kompensiert werden.
In der mitgelieferten Applikation 'DAQ-Test' (programs/DAQ_Test.cvt) wird damit ein in dBfs (*) skaliertes, logarithmisches Spektrum für die Anzeige als Hintergrundbild in einem Diagramm aufbereitet:

```
   ...
   Math.rfft( fltFFTin, fltFFTout ); // Vorwärts-FFT mit reellem Ein- und
komplexem Ausgang
   ref := 32767.0 * C_FFT_SIZE;      // Referenz für 0 dBfs (*) vom 16-Bit-
A/D-Wandler
   Math.ComplexToMagnitudes( fltFFTout, fltLogSpectrum, ref ); // Array
logarithmieren (-> dB)
```

(*) dBfs : "decibel over full scale"
0 dBfs entspricht in diesem Beispiel die Amplitude eines reinen Sinus-Tons, dessen Spitzenwert *grade eben* die Übersteuerungsgrenze eines 16-Bit-A/D-Wandlers erreicht. Die 16-Bit-Werte lagen im Wertebereich +/- 32767 - daher der erste Teil des Referenzwertes.
Eine Sinusfunktion mit der Amplitude A, mit <C_FFT_SIZE> Abtastwerten im Zeitbereich, ergibt im Frequenzbereich (FFT) ein Maximum von A * C_FFT_SIZE (!) - daher der zweite Teil des Referenzwertes.

Die Prozedur Math.ComplexToMagnitudes() führt für jedes komplexe Wertepaar (re,im) die folgende Operation durch :
`output[i] := 20 * log10( sqrt( re^2 + im^2 ) / reference );` //
Prinzip - nicht die Implementierung

Die oben aufgezählten Array-manipulierenden Funktionen setzen nicht nur *Array-Daten*, sondern modifizieren auch einige Komponenten im *Array-Header* des Ziel-Arrays:

- <Ziel-Array>.unix_time wird vom Quell-Array kopiert,
- <Ziel-Array>.t_sample wird bei Fourier-Transformationen auf die *Bin-Breite in Hertz* gesetzt,

- <Ziel-Array>.t_sample wird bei allen anderen Array-Manipulations-Befehlen vom Quell-Array übernommen.
- <Ziel-Array>.len wird auf die Anzahl *tatsächlich verwendeter* Elemente gesetzt.
  (bei der komplexen Fouriertransformation i.A. eine Zweierpotenz)

9.2 4.10.2 Timer und Stoppuhren (in der Script-Sprache)

9.2.1 4.10.2.1 Event- oder Intervall-Timer

Mit dem Script-Kommando **setTimer** wird ein Timer gestartet, z.B.:

```
      var
        tTimer timer1;  // Deklaration einer Timer-Instanz als globale Variable
des Typs 'tTimer'
      endvar;
         ...
      setTimer( timer1, 200 ); // Starte 'timer1' mit einer Periodendauer von
200 Millisekunden,
                               // hier noch ohne Timer-Event-Handler
         ...
```

Der erste Parameter beim Aufruf von setTimer ist strenggenommen *die Adresse* einer *globalen* Variablen vom Typ tTimer. Es gelten die gleichen Regeln (bezüglich des Address-Ausdrucks) wie bei der Funktion addr.

Um zu testen ob ein Timer abgelaufen ist (d.h. "die programmierte Zeit ist mindestens einmal um") kann das Script das 'expired'-Flag in der tTimer-Struktur abfragen. Beispiel:

```
      if ( timer1.expired ) then
        timer1.expired := FALSE; // 'expired'-Flag quittieren; wird nach Ablauf
des Timers wieder gesetzt
        system.beep( 1000, 1 ); // kurzer Piepton (1000 Hz, 1 mal 100 ms Dauer)
      endif;
```

Hinweis: Solange der Timer nicht explizit gestoppt wird (z.B. per timer1.running := FALSE), wird das 'expired'-Flag immer wieder (periodisch) gesetzt.
Unabhängig vom Zeitpunkt an dem das 'expired'-Flag wie im obigen Beispiel zurückgesetzt wird, läuft der Timer im Hintergrund nahezu synchron weiter.

Optional kann (als dritter Parameter) beim Aufruf von setTimer *der Name* (als String) oder *die Adresse* (per addr) eines Timer-Event-Handlers übergeben werden. Dieser Event-Handler (in Form einer selbstdefinierten Script-Funktion) wird von da an periodisch aufgerufen. Details und ein Beispiel für einen Timer-Event-Handler folgen im Kapitel 'Timer-Events'.

Siehe auch: wait_ms(), system.ti_ms, system.unix_time, StartStopwatch(), ReadStopwatch_ms() .

9.2.2 4.10.2.2 StartStopwatch / ReadStopwatch_ms ("Stoppuhr" zur Zeitmessung in Millisekunden)

In einigen Fällen kann der im vorherigen Kapitel beschriebene Timer (tTimer) durch eine einfachere Funktion ersetzt werden, die weniger Resourcen benötigt.
Soll z.B. lediglich die Zeit zwischen zwei Ereignissen (oder Programmzeilen im Scipt) *gemessen* werden, ohne weitere Aktionen (Events) auszulösen, eignet sich auch die im Folgenden beschriebene Alternative.
Sie arbeitet ähnlich wie eine 'Stoppuhr' bei der Streckenzeitmessung:

- passiert der Läufer den Startpunkt für die Zeitmessung, wird per StartStopwatch() die Stoppuhr gestartet;
- passiert der Läufer den Zielpunkt für die Zeitmessung, wird per ReadStopwatch_ms() die verstrichene Zeit in Millisekunden 'abgelesen'.
  (die Uhr wird dazu nicht "gestoppt" sondern wirklich nur abgelesen - Details weiter unten)

Die hier beschriebene Stoppuhr wurde möglichst resourcenschonend angelegt, die "tickende Uhr" verbraucht keine CPU-Zeit :

- durch die Anweisung StartStopwatch(< Integer-Variable >) wird lediglich der aktuelle Wert des Zeitmarken-Generators (system.timestamp) in die angegebene Variable umkopiert
- beim Aufruf von ReadStopwatch_ms(< Integer-Variable >) wird die Differenz zwischen Zeitmarken-Generator und ("minus") dem in der Integer-Variablen gepeicherten 'Startwert' gebildet, und in Millisekunden umgerechnet.

Da, wie oben beschrieben, nicht der Wert in der Integer-Variablen inkrementiert wird, sondern lediglich beim 'Ablesen' der Uhr die verstrichene Zeit in Millisekunden berechnet wird, kann -im Gegensatz zu setTimer(tTimer)- eine nahezu unbegrenzte Anzahl dieser 'Stoppuhren' gleichzeitig laufen, um per Script verschiedene Intervalle zu messen. Ferner kann eine einmal *gestartete* Stoppuhr beliebig oft 'abgelesen' werden, ohne sie zu stoppen.

Beispiel:

```
var
  int MeineStoppuhr;  // Deklaration einer einfachen Integer-Variablen
endvar;
   ...
  StartStopwatch( &MeineStoppuhr );
  Tue_Irgendwas();
  if( ReadStopwatch_ms( &MeineStoppuhr ) > 500 ) then
      print("Überraschung: 'Tue_Irgendwas()' hat über 500 ms gedauert !");
  endif;
   ...
```

9.3 4.10.3 Befehle für mehrzeilige Textfelder (Text-Panels): print, gotoxy, cls & Co

Die folgenden Kommandos dienen zur Anzeige von mehrzeiligen Texten, die als "\panel" - Element in der *Definition von programmierbaren Anzeigeseiten* verwendet werden können. Ein 'Text-Panel' kann den ganzen, oder nur einen Teil des Grafik-Bildschirms belegen. Das Text-Panel bietet eine einfache Möglichkeit, Listen, scrollbare Textfenster, Popups, und ähnliches zu realisieren. Ein

Text-Panel stellt quasi ein "Sichtfenster" auf einen größeren simulierten Textbildschirm dar, dessen Inhalt mit den folgenden Befehlen manipuliert werden kann.

`cls` : "clear screen"

Löscht den simulierten "Text-Bildschirm" (in diesem Fall wird damit lediglich ein Puffer mit Leerzeichen gefüllt, und die Vorder- und Hintergrundfarben in allen Zellen auf die aktuellen Farben gesetzt (siehe setcolor). Der Cursor für eventuell folgende Ausgaben (per 'print') wird in die linke obere Ecke des emulierten Textbildschirms gesetzt.

`clreol` : "clear to end of line"

Löscht den Rest der aktuellen Textzeile, beginnend an der aktuellen 'Cursor'-Position (siehe gotoxy). Die Cursorposition selbst wird von diesem Befehl ***nicht*** beeinflusst.

`setcolor(foreground,background)` : Setzt die Zeichenfarben für alle nachfolgenden Ausgaben in den Puffer des simulierten Textbildschirms.

Diese Farben werden bei den nachfolgenden Aufrufen der Kommandos print, cls, und clreol verwendet. Als Farbwerte sollten keine numerischen Werte verwendet werden (da das Farbmodell hardware-abhängig sein könnte). Verwenden Sie stattdessem die hier aufgezählten symbolischen Farbwerte (clBlack, clWhite, clRed, clGreen, clBlue, usw).
Soll eine der beiden Farben *nicht* geändert werden, so kann als Farbwert eine negative Zahl als Argument übergeben werden (d.h. -1 = "nicht ändern").

`rgb`(Rot, Grün, Blau) : Funktion zum Zusammenstellen einer Farbe aus den drei Farbkomponenten.

Neben den symbolischen Farbkonstanten (clBlack, clWhite, etc) ist dies die einzige 'empfohlene' Methode, im Script-Programm Farben zu definieren.
Der Wertebereich jeder Komponente reicht von 0 bis 255, unabhängig davon wie viele verschiedene Farben auf dem Display (LCD, TFT) wirklich dargestellt werden können (abhänging von "Bits pro Pixel" im Framebuffer). Aus dem Grund sind nicht alle der 2 ^ 24 theoretisch möglichen Farbwerte exakt realisierbar ! Die Firmware (bzw. der LCD-Treiber) verwendet immer die 'bestmögliche' Farbe, d.h. die Farbe, die dem spezifizierten "RGB-Gemisch" möglichst nahe kommt.
Der von der Funktion 'rgb' zurückgegebene Wert vom Typ tColor ist in hohem Maße hardware-abhängig ! Stellen Sie keine Vermutung darüber an, wie dieser Integer-Wert aufgebaut sein könnte... der Aufbau könnte sich von Gerät zu Gerät unterscheiden !
Das Demoprogramm 'Loop Test' verwendet die rgb()-Funktion, um ein Text-Panel mit einem Regenbogen-ähnlichen Prüfmuster zu füllen.

`gotoxy(x,y)` : Setzt den Cursor für die Textausgabe in Spalte 'x', Zeile 'y' im emulierten Textbildschirm.

Dabei muss 'x' zwischen 0 und 79, und y zwischen 0 and 24 liegen (der simulierte Textbildschirm entspricht dem Bildschirm eines alten 'DOS'-PCs mit 25 Zeilen und 80 Zeichen pro Zeile, auch wenn im Display nur ein Teil davon sichtbar ist).
Beispiel: gotoxy(0,0) setzt den Ausgabecursor für das nächste Zeichen in die obere linke Ecke des Text-Panels.

`print` : "Druckt" die in der Parameterliste angegebenen Werte (Strings oder/und Zahlen) auf den emulierten Textbildschirm.

Der Ausgabe-Cursor wird dabei pro Zeichen um eine Position weiterbewegt. Erreicht er den rechten Rand des emulierten Text-Bildschirms (d.h. 80 Zeichen pro Zeile), springt der Cursor an den Anfang der nächsten Zeile.
Um mehr als einen Parameter pro print-Aufruf zu 'drucken', können mehrere Werte in der Parameterliste (durch Komma getrennt) übergeben werden. Beispiel:

```
print("\nResults A,B,C = ",A," ",B," ",C)
```

(Zur Erinnerung: Backslash-n bedeutet 'new line' in einer String-Konstanten).
Bei der Ausgabe numerischer Parameter (int oder float) verwendet print *keine* führenden Nullen. Soll die Ausgabe eine bestimmte (feste) Breite bzw. führende Nullen enhalten, verwenden Sie z.B. die Funktion itoa (integer-to-ascii) um die Zahl in einen String mit einer genau definierten Anzahl von Ziffern umzuwandeln.
Beispiel (aus der Applikation 'TimeTest.cvt', zeigt ein Kalenderdatum im ISO 8601-Format an, z.B. 2011-12-31) :

```
print( itoa(year,4), "-", itoa(month,2), "-", itoa(day,2) );
```

tscreen : Objekt mit allen Eigenschaften eines 'Textbildschirms' (hier: Puffer für den simulierten Text-Bildschirm, der als 'Panel' angezeigt werden kann).
Eigenschaften:

tscreen.cell[Y][X]
Ermöglicht den Zugriff auf die Zeichen-Zelle in Zeile Y, Spalte X (*). Jede Zelle im Text-Puffer ist als Struktur vom Typ tScreenCell definiert. Darin enthalten sind u.A. der Zeichencode, Vorder- und Hintergrundfarbe sowie Flags zum erzwungenen Neu-Zeichnen und andere Effekte. Da die meisten zur Anzeige des Text-Panels verwendeten Zeichensätze einen DOS-kompatiblen Zeichensatz ('codepage 437') verwenden, können mit Hilfe dieses zweidimensionalen Arrays auch begrenzt 'grafikfähige' Anzeigen realisiert werden - siehe Beispielprogramm 'TScreenTest' .
(*) Wie bei allen Arrays in der Script-Sprache üblich, beginnt auch hier die Index-Zählung bei Null.
Zeichencode, Vorder- und Hintergrundfarbe des 'Zeichens in der oberen linken Ecke' des Text-Puffers stehen daher in **tscreen.cell[0][0]** !

tscreen.cell_width
Breite einer Zeichen-Zelle in Pixel, abhängig von der Darstellung auf der aktuellen Seite.

tscreen.cell_height
Höhe einer Zeichen-Zelle in Pixel, abhängig von der Darstellung auf der aktuellen Seite.
Wurde die Anzeige-Seite beim Laden automatisch an die Bildschirmauflösung angepasst,
dann können cell_width und cell_height vom ursprünglich 'designten' Wert abweichen.

tscreen.cx
Liefert die aktuelle horizontale Position des Text-Ausgabe-Cursors ('x'-Koordinate, Zählung beginnt bei Null am linken Rand)

tscreen.cy
Liefert die aktuelle vertikale Position des Text-Ausgabe-Cursors ('y'-Koordinate, Zählung beginnt bei *Null* in der *ersten* Zeile)

tscreen.cx und cy sind nur lesbar. Um die Position des Text-Cursor zu ändern verwenden Sie den Befehl gotoxy(<Spalte>,<Zeile>) .

tscreen.cs

Cursor Shape / Cursor Style. Bestimmt das Erscheinungsbild des Text-Cursors. Diese kann per Script als Bit-Kombination aus den folgenden Konstanten definiert werden:
csOff (Cursor aus), csUnderscore ("Tiefstrich"), csSolidBlock ("gefüllter Block"), csBlinking (blinkend).
Im Beispielprogramm "VT100" wird dieses Feature zur Emulation des blinkenden Cursors in einem VT100- oder VT52-Terminal verwendet.

tscreen.xmax

Liefert die maximal zulässige 'x'-Koordinate des Text-Cursors (d.h. die maximale Spaltennummer, i.e. 79).

tscreen.ymax

Liefert die maximal zulässige 'y'-Koordinate des Text-Cursors (d.h. die maximale Zeilennummer).
In älteren Firmware-Versionen war tscreen.xmax auf 79, und tscreen.ymax auf 39 festgelegt, d.h. 80 Zeilen mit je 40 Zeichen pro Zeile (beachten Sie auch hier die Index-Zählung *ab Null*). Seit 2013-03-20 kann die 'Geometrie' des simulierten Textbildschirms aber per Zuweisung an tscreen.xmax und tscreen.ymax nach den Erfordernissen des Scripts angepasst werden.
Bei den meisten Geräten (mit Firmware-Compilationsdatum ab 2013-03-20) gilt dabei:

- tscreen.xmax darf nicht über 99 liegen (d.h. bis zu **100** Zeichen pro Zeile)
- Das Produkt aus (tscreen.xmax+1) * (tscreen.ymax+1) darf 8000 nicht überschreiten
  (denn der Puffer ist auf 8000 Elemente vom Typ tScreenCell begrenzt)

Beim Ändern der 'Geometrie' sollte erst der kleinere Wert gesetzt werden (meistens tscreen.xmax), damit das Produkt (Ergebnis der Multiplikation) niemals das Maximum überschreitet. Beispiel:

```
  tscreen.xmax := 39; // wir brauchen nur 40 Zeichen pro
Zeile (Indizes 0..39), aber..
  tscreen.ymax := 99; // 100 Zeilen (0..99) im simulierten
Textbildschirm ("Text Panel") !
```

Hinweis: Im Gegensatz zu den weiter unten erwähnten Funktionen hängen tscreen.xmax und tscreen.ymax ***nicht*** von der tatsächlich sichtbaren Größe des Text-Panels auf der aktuellen UPT-Anzeigeseite ab !

tscreen.auto_scroll

Dieses Flag kann von der Display-Applikation gesetzt werden, um das automatische Scrollen des Text-Puffers zu aktivieren. Mit tscreen.auto_scroll = TRUE wird der komplette Inhalt des Text-Puffers um eine (Text-)Zeile nach oben gescrollt, wann immer **tscreen.cy** den Wert von **tscreen.ymax** überschreitet.
Per Default ist das automatische Scrollen *abgeschaltet* (tscreen.auto_scroll := FALSE), was dazu führt das 'nicht mehr in den Textpuffer passende Zeilen' (per 'print')

verlorengehen bzw. nicht "gedruckt" werden.
Ein Beispiel für ein automatisch scrollendes Text-Panel mit Scroll-Indikator finden Sie u.A. im 'Internet-Demo' (InetDemo.cvt) .

tscreen.vis_width

Liefert die Breite des momentan sichbaren Text-Panels (gemessen in Zeichen-Zellen, nicht in Pixeln). Diese 'sichtbare' Netto-Breite hängt von der Größe der optionalen Umrahnumg, des für die Anzeige verwendeten Zeichensatzes, und von der Größe des ersten sichtbaren Text-Panels auf der ***aktuellen UPT-Anzeigeseite*** ab. Beispiel: Ein 320 Pixel breites Text-Panel, ohne Rahmen, angezeigt mit einem Zeichensatz mit 8 Pixel Breite (pro Zeichen) bietet eine nutzbare (sichtbare) Breite von 40 Zeichen.

tscreen.vis_height

Liefert die Höhe des momentan sichbaren Text-Panels (gemessen in Text-Zeilen), ähnlich wie tscreen.vis_width. Ein 240 Pixel hohes Text-Panel hat, bei einem 16 Pixel hohen Zeichensatz, eine sichtbare Höhe von 15 Zeilen (typischer Wert für QVGA-Displays).
Im Demoprogramm 'QuadBlocks' dient diese Funktion zur automatischen Anpassung der Größe des 'Spielfelds', wenn die ursprünglich für 320*240-Pixel-Displays entworfene Applikation in ein Gerät mit einer höheren Auflösung (z.B. 480 * 272 Pixel) geladen wird.

tscreen.scroll_pos_x,
tscreen.scroll_pos_y

Aktuelle Scroll-Position bei der Anzeige des 'virtuellen' Textbildschirms auf dem für die Anzeige verwendeten Text-Panel.
Mit tscreen.scroll_pos_x=0 und tscreen.scroll_pos_y=0 wird in der oberen linken Ecke des Textpanels das Zeichen in der ersten Spalte (x=0), und ersten Zeile (y=0) des virtuellen Textbildschrims angezeigt. Ein Beispiel zum Einsatz der Scroll-Position finden Sie im Demo 'ScriptTest3.cvt', Funktion 'ScrollIntoView'. Durch den Aufruf dieser Funktion wird die zuletzt per 'print' ausgegebene Zeile sichtbar gemacht, indem die Scroll-Position für das Textpanel entsprechend angepasst wird. Mit Hilfe eines entsprechend konfigurierten Balkendiagramms kann dies als 'vertikaler Scroller' bei Geräten mit Touchscreen interaktiv (zum manuellen Scrollen) eingesetzt werden.

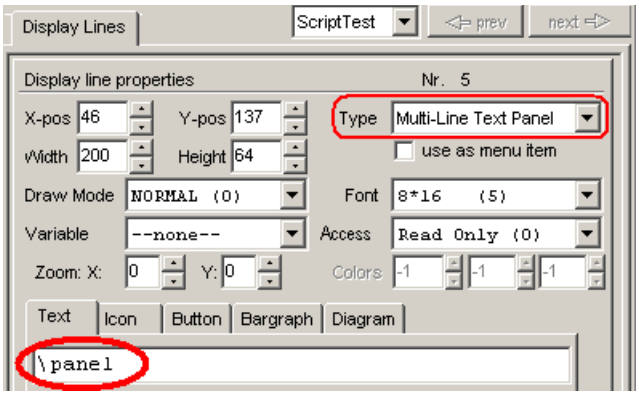
tscreen.modified

Liefert den Wert TRUE (1) wenn der Text-Puffer modifiziert, aber noch nicht auf dem Grafikbildschirm (TFT, LCD) aktualisiert wurde.
Das Script kann dieses Flag auch setzen (tscreen.modified := TRUE) um das System zum Neu-Zeichnen des (Text-)Bildschirm zu veranlassen. Dies ist z.B. dann nötig, wenn der Inhalt des Text-Puffers nicht per 'print', sondern durch direkten Zugriff per tscreen.cell modifiziert wurde. Das Flag wird von der UPT-Firmware automatisch zurückgesetzt, sobald die Aktualisierung 'erledigt' ist.
Diese Funktion wird im Beispielprogramm 'TScreenTest' verwendet, um das Neu-Zeichnen des Textbildschirms zu veranlassen, und um abzuwarten bis diese Aktualisierung 'komplett' ist.

Hinweis:

Die mit dem Kommando 'print' erzeugte Ausgabe erfolgt zunächst in den Puffer für den emulierten Text-Bildschirm. Ein Aussschnitt dieses emulierten Bildschirms kann in Form eines Text-Panels in der UPT-Anzeige-Seite eingebaut werden. Falls momentan kein derartiges Text-Panel auf der UPT-Anzeige-Seite sichtbar ist, bleibt die mit 'print' erzeugte Ausgabe zunächst unsichtbar. Wird danach auf eine UPT-Seite umgeschaltet, auf der sich ein Text-Panel befindet, dann wird auch der 'schon vorher gedruckte' Text komplett sichtbar.
Im Programmiertool kann der Inhalt des emulierten Textbildschirms in der rechten Hälfte der Registerkarte *Script* angezeigt werden. Dazu in der Combo-Box statt **'Debug-Anzeigen verbergen'** den Eintrag **'Puffer für Text-Panels'** auswählen.

Um ein Text-Panel auf einer UPT-Anzeige-Seite zu erzeugen, geben Sie die Backslash-Sequenz **`\panel`** im Format-String einer Anzeige-"Zeile" im Programmiertool ein (s.U.), oder ändern Sie den Typ eines bereits existierenden Anzeige-Elements von 'Text' (=normaler, einzeiliger Text) nach 'Multi-Line Text Panel'. Dadurch wird aus dem 'normalen' UPT-Anzeige-Element ein mehrzeiliges Textpanel. Stellen Sie dann noch eine passende Breite (width) und Höhe (height) auf der Registerkarte 'Display Lines' ein. Beispiel:



Definition eines Text-Panels

Das Text-Panel sollte für die zu erwartenden Ausgaben groß genug sein. Im oben gezeigten Beispiel ist das Text-Panel 200 Pixel breit und 64 Pixel hoch. Für die Anzeige wird hier ein Zeichensatz mit 8 * 16 Pixeln (x*y) verwendet. Auf dem Panel sind daher maximal 25 (=200/8) Zeichen pro Zeile, und vier Zeilen (=64/16) sichtbar. Unabhängig davon können im Zeichenpuffer (für den emulierten Textbildschirm) *bis zu* 40 Zeilen mit je 80 Zeichen pro Zeile gespeichert werden (hardware- und speicherabhängig, das Minimum bei "kleinen" Geräten sind 25 Zeilen mit 40 Zeichen pro Zeile). Da diese Parameter hardware- und firmwareabhängig sind, kann die Dimension des Text-Puffers durch die Funktionen tscreen.ymax und tscreen.xmax ermittelt werden. Per Script kann der auf dem Text-Panel sichtbare Ausschnitt gescrollt werden (tscreen.scroll_pos_x/y), ohne den Inhalt des Text-Puffers zu ändern.

Die Textfarben innerhalb des Text-Panels werden allein durch das Script gesteuert (nicht durch die Farben in der Definition des Text-Panels). Die Farben in der Definition des Text-Panels wirken sich nur auf den optionalen Rand (= farbiger Rahmen um das Textpanel) aus. Jede Zelle im Text-Puffer kann eine eigene (individuelle) Text- und Hintergrundfarbe haben. Im Beispielprogramm 'LoopTest' wird diese Möglichkeit verwendet, um ein reichlich buntes Test-Muster innerhalb des Text-Panels zu erzeugen. In den Beispielen 'TScreenTest' und 'QuadBlocks' dient das Text-Array

als 'Spielfeld', wobei die Möglichkeit des Schreib-/Lesezugriffs auf den Text-Puffer ausgenutzt wird.

Siehe auch: Mehrzeilige Text-Panels in der Definition von UPT-Anzeigeseiten (externer Link, funktioniert nur in HTML, aber nicht im PDF-Dokument).

9.4 4.10.4 Canvas-Funktionen (zum Zeichnen auf einer im Script deklarierten 'Leinwand')

Bei der Erstellung dieser Beschreibung (2017-12-14) existierten nur rudimentäre Funktionen zum Zeichnen *per Script* - eine detailierte Beschreibung folgt später (2018).
Weitere Details (aktueller als im Hilfesystem des Programmiertools) finden Sie dann hier .

Von der im nächsten Absatz beschriebenen Ausnahme abgesehen, müssen Objekte vom Typ tCanvas im Script als globale Variable(n) deklariert sein, d.h. im Abschnitt zwischen var und endvar. Nur dann werden sie vom Programmiertool in Auswahllisten zur Verfügung gestellt, z.B. als Hintergrundbild von Diagrammen und ähnlichen Anzeige-Elementen. Alternativ können Canvas-Objekte auch mit der Backslash-Sequenz \canvas(<name>) sichtbar gemacht werden. Lediglich beim 'direkten' Zeichnen in den Framebuffer (in OnPageUpdate()) wird keine eigene Canvas-Instanz benötigt.

Beispiel aus der Applikation 'DAQ-Test.cvt' :

```
var
   tCanvas spectrogram; // graphic time / frequency representation of spectra
("waterfall")
     ...
endvar;
```

In vielen Fällen werden 'Canvas'-Objekte mit sichtbaren Anzeige-Elementen verbunden. Sobald das Anzeige-Element das erste Mal auf einer Anzeigeseite *sichtbar* wird, wird die Größe (width, height) automatisch an das Anzeige-Element angepasst, falls das Script die passende Größe nicht *bereits vorher* eingestellt hat.
Da die Größe eines Anzeige-Elements (und damit auch die Größe der damit eventuell verknüpften Canvas) während *der Laufzeit* automatisch skaliert werden könnte, sollte das Script sich nicht auf eine bestimmte, feste Größe der Zeichenfläche verlassen, sondern diese ebenfalls erst *zur Laufzeit* abfragen (d.h. jedesmal wenn in die Canvas gezeichnet werden soll). Dadurch würde z.B. eine ursprünglich für ein MKT-View III mit 480 * 272 Pixeln entworfene Applikation auch problemlos in einem MKT-View IV mit 800 * 480 Pixeln funktionieren.

9.4.1 4.10.4.1 Zugriff auf einzelne Pixel einer Canvas

Mit der Methode **pixel[Y][X]** kann pixelweise auf die gesamte Zeichenfläche (tCanvas) zugegriffen werden. Dies ist zwar die einfachste, aber auch die langsamste Möglichkeit zur Manipulation von Bilddaten. Beispiel :

```
for y:=0 to spectrogram.height-1  // spectrogram: Variable des Typs tCanvas
   for x:=0 to spectrogram.width-1
       spectrogram.pixel[y][x] := rgb(x,y,x+y);
   next x;
next y;
```

Wie auch bei den später folgenden Methoden zur Manipulation der Pixeldaten erscheinen die geänderten 'Pixel' nicht sofort auf dem Bildschirm, denn ein Canvas-Object existiert 'per se' nur im Speicher ("off-screen"). Die Änderung wird erst sichtbar, wenn das Objekt (tCanvas) als Teil eines *sichtbaren Anzeige-Elements* auf dem Bildschirm erscheint.

9.4.2 4.10.4.2 Füllen eines Recktecks (Methode von tCanvas)

Mit der Methode **rect(x1,y1,x2,y2,c)** wird das zwischen x1/y1 und x2/y2 aufgespannte Rechteck mit der Farbe 'c' gefüllt.

9.4.3 4.10.4.2 Horizontales Scrollen (Methode von tCanvas)

Mit der Methode **hscroll(x1,y1,x2,y2,n)** wird das zwischen x1/y1 und x2/y2 aufgespannte Rechteck um n Pixel nach links (n<0) oder rechts (n>0) gescrollt.

9.4.4 4.10.4.3 Vertikales Scrollen (Methode von tCanvas)

Mit der Methode **vscroll(x1,y1,x2,y2,n)** wird das zwischen x1/y1 und x2/y2 aufgespannte Rechteck um n Pixel nach oben (n<0) oder unten (n>0) gescrollt.
Beispiel aus der Application 'DAQ_Test.cvt' :

```
xmax := spectrogram.width-1; // spectrogram: Variable des Typs tCanvas
ymax := spectrogram.height-1;
spectrogram.vscroll( 0,0, xmax, ymax, 1/*scroll DOWN by one pixel*/ );
```

9.4.5 4.10.4.4 Zeichnen einer Bitmap-Grafik (Methode von tCanvas)

Ähnlich wie in HTML5 kann diese Methode mit einer unterschiedlichen Anzahl von Parametern aufgerufen werden, z.B. mit **myCanvas** = Objekt vom Typ tCanvas:

myCanvas.drawImage(image, x, y);
Zeichnen ohne Strecken oder Stauchen, die Quell-Grafik (image) wird 1:1 kopiert.

myCanvas.drawImage(image, x, y, width, height);
Zeichnen mit Strecken oder Stauchen (width, height = Breite und Höhe des *Ziels*, in Pixeln).

myCanvas.drawImage(image, sx, sy, swidth, sheight, dx, dy, dwidth, dheight);
darin sind:
sx = source-X (Quell-Koordinate, linker Rand)
sy = source-Y (Quell-Koordinate, Oberkante)
swidth = Breite des zu kopierenden Quell-Rechtecks in Pixeln
sheight = Höhe des zu kopierenden Quell-Rechtecks in Pixeln
dx = destination-X (Ziel-Koordinate, linker Rand)
dy = destination-Y (Ziel-Koordinate, Oberkante)
dwidth = Breite des *Ziel*-Rechtecks in Pixeln
dheight = Höhe des *Ziel*-Rechtecks in Pixeln

Als Quelle (erster Parameter, im obigen Beispiel: 'image') können die folgenden Objekte bzw. Datentypen verwendet werden:

tCanvas :

Ein weiteres Objekt vom Typ tCanvas kann auch als 'Quelle' für die zu zeichnende Grafik dienen.

Byte-Array :

Bitmaps im 'MKT-Format' können als Byte-Arrays direkt im Script enthalten sein.
In der Beispiel-Applikation MacPan wird dies für die 'Sprite'-Grafiken (Spieler, Geister, etc) verwendet, z.B.:

```
byte pac_left[] = { // Pacman-Sprite : bitmap header followed by bitmap
data ("pixels")
           1,        // bitmap type: MKT "simple", no compression
           4,        // bits per pixel (here: four bits -> 16 different
colours, one hex digit per pixel)
           (word)16, // width in pixels (16 bit aka "word")
           (word)16, // height in pixels (" ")
           1,        // transparency flags
           0,        // "transparent" colour index
           (word)0,  // size of the colour palette (16 bits), 0 = no extra
colour palette
   //          0123456789ABCDEF
   /* 0 */  0x0000000000000000,
   /* 1 */  0x0000002222000000,
   /* 2 */  0x0000222222220000,
   /* 3 */  0x0002222222222000,
   /* 4 */  0x0020EE2222222200,
   /* 5 */  0x0022E22222222200,
   /* 6 */  0x0F02222222222220,
   /* 7 */  0x000F022222222220,
   /* 8 */  0x00000F0022222220,
   /* 9 */  0x000000F222222220,
   /*10 */  0x0000F02222222200,
   /*11 */  0x00F0222222222200,
   /*12 */  0x0002222222222000,
   /*13 */  0x0000222222220000,
   /*14 */  0x0000002222000000,
   /*15 */  0x0000000000000000
    };
```

9.5 4.10.5 Funktionen zum Zugriff auf Dateien (und 'Datei-ähnliche Geräte')

... existieren nur in Geräten mit geeigneter Hardware. Dies muss nicht unbedingt ein Speicherkarten-Leser sein, denn die in diesem Kapitel beschriebenen Funktionen werden z.T. auch zum Zugriff auf andere Medien verwendet. Dazu zählen die in einigen Geräten vorhandene RAMDISK, oder der Teil eines als Datenspeicher nutzbaren internen FLASH-Speichers, aber auch die manchmal vorhandene(n) serielle(n) Schnittstelle(n)).
Alle *Datei*-Zugriffs-Funktionen beginnen mit dem Schlüsselwort 'file.' (zur Vermeidung von 'namespace pollution'). Funktionen zum Zugriff auf *Verzeichnisse* beginnen mit dem Schlüsselwort 'directory.' .
Die Datei-E/A-Funktionen müssen eventuell vor der Nutzung erst freigeschaltet werden (zumindest bei universell einsetzbaren Geräten wie dem MKT-View II/III/IV).

Übersicht über die Datei-Ein-/Ausgabe-Funtionen :

- file.create(name, max_size_in_bytes) : erzeugt eine neue, beschreibbare Datei mit dem angegebenen Namen.
- file.open(name, o_flags) : öffnet eine existierende Datei (i.A. nur zum Lesen)
- file.write(handle, data) : schreibt in eine Datei
- file.read(handle, destination_variable) : liest aus einer Datei (mit optionalem Parser für mehrere Datenfelder)
- file.read_line(handle) : liest die nächste Textzeile aus einer Textdatei, und liefert diese als Zeichenkette (string) zurück
- file.seek(handle, offset, fromwhere) : Setzt den Dateizeiger
- file.eof(handle) : testet ob das Ende der Datei erreicht wurde (eof = end-of-file)
- file.close(handle) : schliest eine Datei, und gibt deren *Datei-Handle* wieder frei
- file.size(handle) : liefert die Größe einer *geöffneten* Datei, gemessen in Bytes.
- file.delete(name_or_pattern) : Löscht Datei(en), z.B. file.delete("ramdisk/*.*").
- directory.open(path_and_mask, options) : Öffnet ein Directory (Verzeichnis) zum Lesenory to read it. Liefert im Erfolgsfall ein *Handle*, sonst Null.
- directory.read(handle, dir_entry) : Liest den nächsten Dateieintrag aus dem Verzeichnis. Liefert TRUE im Erfolgsfall.
- directory.close(handle) : Schliesst das Verzeichnis nach dem Lesen. Nicht vergessen !

Lesen Sie bitte auch die Hinweise zum Pseudo-Datei-System mit Einschränkungen beim Schreiben und Löschen von Dateien (speziell im internen FLASH), und wie das Pseudo-Datei-System im Programmiertool simuliert werden kann.
Siehe auch: Test/Demoprogramm 'File Test' (demonstriert die Verwendung der obigen Datei-I/O-Funktionen).

4.10.5.1 Verzeichnisse im Pseudo-Datei-System der programmierbaren Terminals

Die meisten programmierbaren Anzeige-Geräte unterstützen bislang keine 'echten' Unterverzeichnisse (auf dem Datenträger). Viele dieser Geräte haben nicht einmal einen integrierten Speicherkartenleser. Trotzdem kann i.A. ein Teil des internen Speichers (FLASH oder/und RAM)

'wie ein Datenträger' genutzt werden. Um gezielt auf einen bestimmten Datenträger zuzugreifen, wird in der Script-Sprache das gleiche System von "Pseudo-Laufwerken" verwendet wie für das File Transfer Utility und den Web-Server.
Per Script-Sprache erzeugte Dateien können auch per File-Transfer-Utility aus dem Gerät aus gelesen werden, oder mit Hilfe des File-Transfer-Utilities (oder Web-Server) im Gerät abgespeicherte Dateien per Script gelesen werden. Der Name des "Pseudo-Verzeichnisses" wird, per Schrägstrich getrennt, vor den eigentlichen Dateinamen gesetzt (siehe Beispiel zu file.open) .

Bislang implementierte Pseudo-Verzeichnisse zum Zugriff bestimmte Speichermedien sind:

- "**font_flash**"
  Dies ist ein Teil des onboard-FLASH-Speichers (keine Speicherkarte!), in dem normalerweise die benutzerdefinierten Zeichensätze ("fonts", *.fnt) abgelegt werden. Je nach der Anzahl und Größe der benutzerdefinierten Zeichensätze bleibt auf diesem Speichermedium noch Platz für weitere Dateien (z.B. Übersetzungen für die Internationalsierung einer Anwendung, etc). Beim Erzeugen einer neuen Datei in diesem Medium muss (in zweiten Parameter von file.create) die erwartete maximale Größe der Datei angegeben werden. Beim Schließen der Datei werden die verbleibenden (ungenutzten) Bytes dann wieder freigegeben. Die gleiche Besonderheit gilt auch für die anderen Datenträger im internen FLASH (= alle Pseudo-Verzeichnisse, deren Namen mit "_flash" enden).
- "**audio_flash**"
  Dies ist ein weiterer (optionaler) Onboard-FLASH-Speicherchip (keine Speicherkarte), in dem vorzugsweise digitalisierte Audiodateien (*.wav) abgelegt werden. Bleibt neben den Audio-Dateien noch genügend Platz auf diesem Speichermedium, kann es ebenfalls vom Script für eigene Zwecke genutzt werden.
  Hinweis: Einige Geräte, z.B. MKT-View II, bieten zwar die Möglichkeit zur analogen Audio-Ausgabe, enthalten aber keinen speziellen "großen" FLASH-Baustein für digitalisierte Audio-Dateien. In diesem Fall sind die Pseudo-Verzeichnisse "font_flash" und "audio_flash" miteinander identisch, d.h. eine im Verzeichnis "font_flash" abgespeicherte Datei erscheint auch im Verzeichnis "audio_flash" ! Im Verzeichnis "audio_flash" abgelegte Dateien können mit dem Interpreter-Kommando 'audio.play' abgespielt werden.
- "**data_flash**"
  Dieser Teil des internen FLASH-Speichers wird vom System für die Ablage interner Daten verwendet (Anzeigeseiten, importierte Bitmaps, Script-Quelltext, etc).
  Abgesehen vom *Hochladen* einer einzelnen *.upt- bzw *.cvt-Datei per YMODEM-Protocol ist dieses Verzeichnis für den Anwender nicht nutzbar.
- "**memory_card**"
  Dieses Pseudo-Verzeichnis dient zum gezielten Ansprechen der FLASH-Speicherkarte. Fehlt dieser Eintrag in der Wurzel des Pseudo-Dateisystems (sichtbar u.A. im File-Transfer-Utility), dann enthält die Firmware keine Funktionen zum Ansteuern einer Speicherkarte. Existiert zwar ein Pseudo-Verzeichnis namens "memory_card", allerdings mit leerem Inhalt, dann ist entweder keine Speicherkarte eingesetzt, oder das Dateisystem auf der Speicherkarte wird von der Firmware nicht unterstützt.
  Im Programmiertool wird das Gerät 'memory_card' per Default durch ein Unterverzeichnis

namens 'sim_mc' (simulated memory card) auf der Festplatte simuliert. Der Pfad für die Simulation ist konfigurierbar.

- "**ramdisk**"
  Dieses Pseudo-Verzeichnis kann zum Speichern temporärer Dateien verwendet werden, z.B. für Bitmap-Dateien die nicht dauerhaft im FLASH-Speicher oder auf der Speicherkarte gesichert werden müssen. Ähnlich wie beim Zugriff auf interne FLASH-Speicher muss auch hier im zweiten Parameter von file.create() die erwartete maximale Größe der zu schreibenden Datei angegeben werden. Nach dem Schließen der so erzeugten Datei werden die davon 'nicht verbrauchten' Bytes wieder frei.
  Das Demoprogramm "FileTest" verwendet dieses Pseudo-Laufwerk zum Erzeugen einer Textdatei, in die einige Zeilen geschrieben, und anschließend wieder ausgelesen werden.
  Beachten Sie dass der Inhalt der 'RAMDISK' beim Abschalten der Versorgungsspannung, aber auch beim Umschalten in den Power-Down-Modus, verloren geht !
  (Grund: das dynamische RAM wird im Power-Down nicht mehr refresht, zumindest nicht beim MKT-View II/III/IV).
  Bei der *Simulation* eines Gerätes (mit RAMDISK) im Programmiertool wird die Ramdisk nur programmintern nachgebildet. Sie ist *nicht* in das Dateisystem des Hosts (Windows-PC) eingebunden ! Um trotzdem den Inhalt der RAMDISK während der Simulation zu untersuchen, wählen Sie im Hauptmenü des Programmiertools die Funktion
  Ansicht .. Simuliertes Dateisystem .. RAMDISK .

Zusätzlich zu den oben aufgeführen 'echten' ***Speichermedien*** dienen die Datei-Ein-/Ausgabe-Funktionen auch zum Zugriff auf die folgenden ***Geräte*** (die, ähnlich wie bei 'großen' Betriebssystemen, wie Dateien angesprochen werden können) :

- "**serial1**"
  Ermöglicht den Zugriff auf die erste serielle Schnittstelle (RS-232) des Gerätes, zumindest für die elementaren Schreib/Lese-Funktionen.
  In seltenen Fällen ist es nötig, beim Öffnen auch die Schnittstellenparameter einzustellen. Dies ist (zumindest für die Baudrate) möglich, indem eine Zeichenkette mit den Parametern wie im folgenden Beispiel an den Namen des Gerätes angehängt wird:
  `hSerial1 := file.open("serial1/9600");` // try to open 1st serial port, and configure it for 9600 bits/second .
  Weitere Beispiele zum Zugriff auf die serielle Schnittstelle per Script finden sie hier (im "GPS-Simulator").
  Wie alle *erweiterten* Script-Funktionen muss auch der Zugriff auf die seriellen Schnittstellen (per Datei-API) erst freigeschaltet werden !
  Wird eine serielle Schnittstelle als LIN-Bus-Interface verwendet, dann kann darauf (per Script) nicht 'wie eine Datei' zugegriffen werden. Zum Senden und Empfangen von LIN-Frames verwenden Sie stattdessem die CAN-Bus-API (Application Interface). Details dazu in der LIN-Bus-Dokumentation.
- "**serial2**"
  Ermöglicht den Zugriff auf die zweite serielle Schnittstelle (falls diese existiert, anderfalls liefert file.open("serial2") den Wert Null, d.h. "illegales Handle").
  Ein Beispiel zur Nutzung dieser Schnittstelle in der Script-Sprache finden Sie hier ("GPS Simulator").
  Im 'MKT-View II' dient die zweite serielle Schnittstelle zum Anschluss eines GPS-

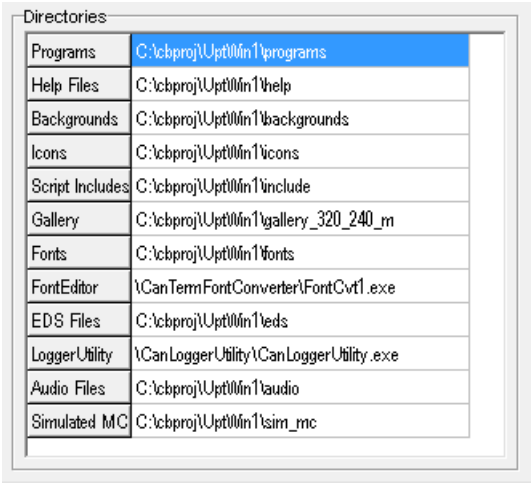
Empfängers.
Dies ist kein Standard-RS-232-Port ! Versuchen Sie niemals, mit einem 'normalen' 9-poligen Schnittstellenkabel diesen Port mit dem PC zu verbinden !
Sie könnten damit Ihren PC beschädigen, denn einige Pins des 9-poligen "GPS"-Steckers dienen als Ausgang für die Versorgungsspannung (vom MKT-View II zum GPS-Empfänger) ! Details finden Sie im Hardware-Handbuch des Gerätes, oder hier (Pinbelegung diverser "serieller Port"-Stecker und -Buchsen).

Hinweise zum Pseudo-File-System (PFS)

Das PFS ist kein 'normales' Dateisystem (weder FAT, noch NTFS, ext2,3,...) . Mit der Ausnahme des Gerätes "memory_card" wird jede Datei als ***einzelner, zusammenhängender, d.h. nicht fragmentierter Block*** auf dem Datenträger gespeichert. Dadurch kann er (innerhalb der Firmware) *direkt* per Zeiger (pointer) durch die CPU auf dem Speichermedium addressiert werden - ohne die Notwendigkeit, den Inhalt der Datei vom Datenträger in den Hauptspeicher umzukopieren.
Aus dem Grund sind einige Datei-Operationen, die Sie möglicherweise vom PC mit einem 'normalen' Dateisystem kennen, hier nicht möglich:

- Bei Dateien im FLASH oder in der RAMDISK muß bereits *beim Anlegen der Datei* die erwartete maximale Größe angegeben werden.
  Später, beim Schließen der Datei, werden die 'nicht benötigen' Bytes wieder frei (d.h. die Größe der Datei wird nachträglich wieder verkleinert), eine Datei kann während des Schreibens aber nicht über die vorher festgelegte 'Maximalgröße' hinauswachsen.
- Das Schreiben (write) ist bei Dateien im FLASH-Speicher nur möglich, nachdem die Datei neu erzeugt wurde (!!), aber nicht wenn eine bereits existierende Datei 'erneut' geöffnet wurde.
- Das Löschen einer einzelnen Datei mag zwar möglich sein (ohne den gesamten Datenträger zu formatieren), wegen der sehr großen FLASH-Sektoren wird dabei aber kein Speicher auf dem Datenträger frei (!!). Grund: Innerhalb eines (64 oder 128 kByte großen) FLASH-Sektors wird nicht nur *eine*, sondern i.A. *mehrere Dateien pro Sektor* gespeichert (im Gegensatz zu normalen Dateisystemem, bei denen ein Sektor nur 512 Bytes umfasst, so daß das gezielte (sektorweise) Löschen einer Datei problemlos möglich ist).
- Da die Dateien in einer RAMDISK nicht verschiebbar sind, kann beim Löschen einer Datei in der RAMDISK deren Speicher nur dann (ohne 'Formatieren') freigegeben werden, solange keine weitere Datei (nach der zu löschenden Datei) auf der RAMDISK kreiert wurde.
  Aus diesem Grund sollten 'temporäre' Dateien, die Ihr Script auf der RAMDISK ablegt, gelöscht werden sobald sie nicht mehr benötigt werden.
- Beachten Sie, daß der Inhalt der RAMDISK trotz angelegter Betriebsspannung auch im Betriebszustand 'Power-Down' (wie bei 'Power-Off') verloren geht. Grund: Ohne CPU-Takt kann das dynamische RAM nicht refresht werden.

Zur Simulation der diversen Speichermedien werden im Programmiertool 'echte' Dateien (auf der Festplatte) verwendet. Die Verzeichnisse, in denen der Inhalt dieser 'simulierten FLASH-Bausteine' abgelegt wird, kann im Programmiertool ***auf der Registerkarte 'Settings'*** konfiguriert werden. Per Doppelklick in die Tabelle wird ein Directory-Auswahl-Dialog gestartet, mit dem der komplette Pfad zum Simulationsverzeichnis definiert werden kann:



Verzeichnisse für die Simulation im Programmiertool

Beispiel: Für die Simulation des Speichermediums "audio_flash" kopieren Sie alle benötigten (Wave-Audio-)Dateien in das entsprechende Verzeichnis, oder legen den Pfad des Simulationsverzeichnisses auf das projektspezifische Verzeichnis, in dem sich die benötigten Dateien auf Ihrer Festplatte (oder Netzlaufwerk) befinden. Aus dem gleichen Verzeichnis können die benötigten Dateien dann später (per File-Transfer-Utility) in das 'echte' Gerät übertragen werden.
Während der Entwicklung der Applikation sollten Sie eine ***Liste aller später (im Target) benötigten Dateien*** anlegen. Zu diesen Dateien könnten folgende Typen (je nach Script auch noch mehr) gehören:

- Die Display-Applikation selbst (*.cvt oder *.upt),
- benutzerdefinierte **Zeichensätze** (*.fnt),
- **Bitmap-Grafiken** (*.bmp), die Ihre Applikation möglicherweise erst während der Laufzeit von einem geeigneten Datenträger lädt. (Im Gegensatz zu den 'normalen' Bitmaps, die beim Entwickeln der Display-Applikation mit dem Programmiertool importiert wurden. Diese Bitmaps ("Icons") sind in der *.cvt bzw. *.upt - Datei enthalten... und tragen i.A. wesentlich zu deren Größe bei.),
- **Audio-Dateien** (*.wav), die während der Laufzeit z.B. als gesprochene Anweisung für den Bediener abgespielt werden,
- **Text-Dateien** (*.txt) in verschiedenen Sprachen, mit deren Hilfe (und eines Scriptes wie im Beispiel 'MultiLanguageTest') eine Anwendung internationalisiert werden kann, ohne daß dazu die Anzeigeseiten dupliziert werden müssen.

Vergessen Sie nicht, von diesen Dateien Sicherungskopien anzulegen, und *ALLE* Dateien zum Programmieren der Geräte an die Fertigung bzw. an den Endanwender / Kunden weiterzugeben !

4.10.5.2 Erzeugen oder Öffnen von Dateien

file.create(name, max_size_in_bytes) : Erzeugt eine neue, schreibbare Datei mit dem angegebenen Namen.

Existiert die Datei bereits, wird sie überschrieben (bzw. gelöscht und dann neu angelegt).
Im Erfolgsfall liefert diese Funktion einen positiven 'Datei-Handle' zurück (= Integer-Wert, durch den die Datei für alle weiteren Zugriffe eindeutig identifiziert werden kann).
Andernfalls liefert diese Funktion einen negativen Fehlercode.
Der Dateiname kann optional ein Pseudo-Verzeichnis enthalten, mit dem das Speichermedium spezifiziert werden kann.
Der Parameter 'max_size_in_bytes' (maximal zu erwartende Dateigröße in Byte) muss bei manchen Speichermedien bereits beim Öffnen spezifiziert werden, um Fragmentierung auf dem Datenträger zu vermeiden (speziell bei der RAMDISK).
Beispiel zum Erzeugen und Schreiben einer Datei in der RAMDISK, mit einem Minimum an Fehlerabfrage :

```
var
  int fh; // file handle
endvar;
fh := file.create("ramdisk/test.txt",4096); // max 4096
bytes
if( fh>0 ) then   // successfully created the file ?
  file.write(fh,"First line in the test file.\r\n");
  file.close(fh); // never forget to close files !
endif;
```

file.open(name [, o_flags]) : öffnet eine *existierende Datei* oder ein *Gerät*, i.A. zum Lesen.
Im Erfolgsfall liefert diese Funktion einen positiven 'Datei-Handle' zurück (= Integer-Wert, durch den die Datei für alle weiteren Zugriffe eindeutig identifiziert werden kann).
Andernfalls liefert diese Funktion einen negativen Fehlercode.
Je nach Speichermedium werden einige (aber nicht alle) der folgenden 'open flags' unterstützt (bitweise kombinierbar für den Parameter 'o_flags'):

- O_RDONLY : Nur zum Lesen öffnen, d.h. die Datei kann gelesen, aber nicht geschrieben werden. Funktioniert auf allen Speichermedien.
- O_WRONLY : Nur zum Schreiben, aber nicht zum Lesen öffnen.
- O_RDWR : Datei sowohl zum Lesen als auch zum Schreiben öffnen. Dies funktioniert z.Z. nur mit 'echten' Dateien auf einer Speicherkarte !
- O_TEXT : Datei im TEXT-MODUS öffnen, und anhand der BOM (byte order mark) die darin verwendete Zeichen-Kodierung ermitteln.
  Alle Strings, die danach (zeilenweise) aus dieser Datei gelesen werden, erben die Zeichen-Kodierung aus der BOM.
  Fehlt eine BOM in der Datei, geht die Funktion davon aus, daß es sich um eine normale "ASCII-Datei" mit 8 Bit pro Zeichen handelt.
- O_CREATE : Wenn die zu öffnende Datei noch nicht existiert, wird sie automatisch neu erzeugt. UNIX-Puristen dürfen statt 'O_CREATE' auch die glorreiche Abkürzung 'O_CREAT' verwenden.

Unabhängig vom Eröffnungs-Modus (O_RDONLY, O_WRONLY, O_RDWR, O_TEXT, O_CREATE) wird der Dateizeiger immer auf den *Anfang* der Datei gesetzt. Das Verhalten der Funktion file.open entspricht daher weitgehend den in der Programmiersprache 'C' bekannten Runtime-Library-Funktion '_rtl_open'. Um eine bereits existierende Datei *am Dateiende* 'weiterzuschreiben', muss der Dateizeiger ggf. an das ENDE der Datei gesetzt

werden. Dazu dient die Funktion file.seek .

Beispiel zum Öffnen einer Textdatei, mit zeilenweisem Lesen, und Ausgabe der eingelesenen Zeilen auf dem Bildschirm:

```
var             // declare global variables:
  int fh;       // an integer for the file handle
  string temp; // a string named 'temp'
endvar;
fh := file.open("ramdisk/test.txt",O_RDONLY | O_TEXT);
if( fh>0 ) then // successfully created the file ?
  while( ! file.eof(fh) )
    temp := file.read_line(fh);
    print( "\r\n ", temp ); // dump the line to the screen
  endwhile;
  file.close(fh);
endif;
```

4.10.5.3 Schreiben und Lesen von *Dateien*

Nach dem erfolgreichen Erzeugen oder Öffnen einer Datei wird deren *Handle* (engl. "Griff", hier: ein Integer-Wert) als erstes Argument für die folgenden Funktionen verwendet, um die Datei zu lesen oder zu schreiben.

file.write(handle, data): Schreibt in eine Datei

In vielen Fällen ist 'data' eine Zeichenkette, ein oder mehrere String-Variablen, oder String-Ausdrücke. ***Geplant***: Die Datentypen 'int' und 'float' werden im Binärformat geschrieben. Das Schreiben von 'Binärdaten' wird nicht empfohlen,weder um den damit verbundenen Ärger mit inkompatiblen Datentypen, High-Low-Byte-Probleme ("endianness" aka "byte order"), etc zu vermeiden. Der ursprüngliche Zweck der Script-Sprache war die Verarbeitung von Zeichenketten, d.h. Strings.
Wird als zweites Argument (data) der Name einer Array-Variablen übergeben, dann schreibt file.write nur soviele Elemente in die Datei, wie momentan in .len enthalten sind (d.h. es wird nicht die Maximalgröße, sondern nur die im Array tatsächlich vorhandenen, 'gültigen' Elemente geschrieben).
Beispiele zur Verwendung von file.write finden Sie unter file.create() und append().

file.read(handle, &dest1, separator1, &dest2, ...) : liest strukturierte Daten aus einer Datei

Liest ein oder mehrere Datenfelder aus einer Datei. Diese Funktion ist (im Vergleich mit file.read_line) wesentlich komplexer, aber vielseitiger. Im Erfolgsfall liefert file.read die Anzahl gelesener Datenbytes zurück. Das Format der einzulesenden Daten wird durch die Argumentenliste beim Funktionsaufruf definiert. Pro Aufruf können seit 06/2016 bis zu 16 Argumente übergeben werden (inklusive Datei-Handle und Feld-Trennern).
Jedes zu lesende Datum (Singular von Daten) sollte als Referenz übergeben werden (vorzugsweise mit dem Address-Operator **&** als Prefix vor dem Namen der zu lesenden Variablen), z.B.:

```
    nBytesRead := file.read(handle, &sName, ",",
&sAdresse,",", &sInfo,"\r\n" );
```

Ursprünglich konnte die Script-Sprache nur *Textdateien* per file.read() einlesen. Hinweise

zum Einlesen von 'binären' Daten per file.read() finden Sie am Ende dieses Kapitels.

Die Zeichenkonstanten (Strings) zwischen den Namen der zu lesenden Variablen definieren die 'Trennzeichen' zwischen den Feldern.
Alternativ kann zum Einlesen von Dateien mit festen Feldbreiten auch die Breite angegeben werden, und zwar *nach* dem Namen der Variablen, in der das eingelesene Feld abgelegt werden soll. Beispiel:

```
    nBytesRead := file.read(handle, &sName,20, &sAdresse,40,
&sInfo,"\r\n" );
```

Im obigen Beispiel werden die String-Variablen 'sName' mit je 20 Zeichen, und 'sAdresse' mit je 40 Zeichen eingelesen. Der Rest der Zeile (d.h. bis zum Separator "\r\n", Carriage Return + New Line) wird in der Variablen 'sInfo' abgelegt.
Bei String-Variablen ist es in einigen Fällen nötig, den zulässigen ***Zeichenvorrat*** einzuschränken. Innerhalb der Argumentenliste von file.read erfolgt dies durch die folgenden Zusätze direkt *nach* dem Namen der einzulesenden String-Variablen:

- :NAME
  Zulässige Zeichen sind A..Z, a..z, '_' (Unterstrich), und -abgesehen vom ersten Zeichen im Namen- die Ziffern '0' bis '9'.
  Leerzeichen und Steuerzeichen wie Carriage Return (\r) und New Line (\n) sind *nicht* erlaubt.
- :NUMBER
  Zulässige Zeichen sind nur die Ziffern '0' bis '9' und der *Dezimalpunkt* ('.'), und die Zeichen 'e' oder 'E' vor dem optionalen Exponenten.

Vereinfachter Auszug aus dem Beispiel zum Einlesen von INI-Dateien, Testdatei IniDemo1.ini :

```
        if( file.read( handle, "[", sSection:NAME, "]", "\r\n" ) > 0 ) then
           // entered a new SECTION in the INI file :
           // ..
        elif( file.read( handle, sKey:NAME, "=", sValue, "\r\n" ) > 0 )
then
           // successfully read a key=value pair from the INI file :
           select (sSection)
              case "FileInfo" :     // ..
              case "SensorConfig" : // ..
              case "CAN1Setup":     // ..
              // ...
           endselect;
        elif( file.read( handle, sGarbage, "\r\n" ) > 0 ) then
           // successfully read a line with "something else" (possibly an
EMPTY line) :
           // ..
        else // didn't read anything so guess we reached the end of the
file:
           file.close( handle );
           return TRUE;
        endif;
```

Erläuterung des obigen Beispiels:

Mit den drei Aufrufen von file.read() werden (der Reihe nach) alle Möglichkeiten 'ausprobiert', den Inhalt der INI-Datei zeilenweise einzulesen.
Der dritte Aufruf (**`file.read( handle, sGarbage, "\r\n" )`**) dient zum 'Überlesen' aller Zeilen, die weder in das Schema [Sektionsname] noch <Schlüsselname>=<Wert> passen.
Da der Zeichenvorrat beim Einlesen der String-Variablen 'sKey' auf 'NAME' begrenzt ist, kann die Variable 'sKey' im obigen Beispiel keine mehrzeiligen Texte, z.B. die Kommentarzeilen am Anfang der Ini-Datei enthalten. Grund: Kommentarzeilen beginnen (zumindest in INI-Dateien) mit einem Semikolon, d.h. kein gültiges Zeichen eines 'Namens'. Ohne diese Vorkehrung würde 'sKey' u.U. mehrere Zeilen enthalten, bis zum ersten "=" (Trennzeichen zwischen 'key' und 'value').

Pro Aufruf von file.read können maximal 2048 Bytes aus der Datei gelesen werden. Jeder Aufruf von file.read ist entweder *komplett erfolgreich* oder liest *kein einziges Byte* aus der Datei. Dies wird im obigen Beispiel ausgenutzt, um *Kommentarzeilen*, *Sektions-Header-Zeilen*, und *Datenzeilen* zu unterscheiden.

Seit 2018 können per file.read() notfalls auch *binäre* Dateien eingelesen, bzw. binäre Streams von einer seriellen Schnittstelle verarbeitet werden. Die Datenübergabe erfolgt dann vorzugsweise mit Hilfe eines Byte-Arrays wie im folgenden Beispiel:

```
var
  byte b40RxData[40]; // byte-array for received data
  int  nBytesRcvd;    // number of bytes received
  int  hSerial1;      // handle to the 1st serial port
    ...
endvar;
    ...
hSerial1 := file.open("serial1/9600"); // open 1st serial port with
9600 bits/s
    ...
while(1)  // endless loop to read and process received data...
   nBytesRcvd = file.read(hSerial1, b40RxData);
   if( nBytesRcvd > 0 ) then  // something received ?
      ProcessRcvdBytes( b40RxData, nBytesRcvd );
   endif;
   wait_ms(50);  // wait to let the display program work
endwhile;
    ...
proc ProcessRcvdBytes( byte ptr pbRxData, int nBytesRcvd )
  local int i;
  local byte b;

  // Process all bytes, from the array passed in as function argument
  for i:=0 to nBytesRcvd-1
     b := pbRxData[i]; // next received byte
    ...
  next i;
endproc;
```

Komplette Beispiele zur Verwendung der Funktion file.read finden Sie in der 'VT100-Emulation' und in der Demonstation zum Einlesen von INI-Dateien.

file.read_line(handle) : Liest die nächste Zeile aus einer Textdatei, und liefert deren Inhalt als String zurück.

Ein Beispiel dazu finden Sie unter file.open .
Um den Inhalt einer Textdatei zeilenweise einzulesen, sollte die Datei mit dem Flag O_TEXT in der Funktion file.open geöffnet werden,
denn nur dann erhält der zurückgelieferte String automatisch zum Dateityp passende Zeichenkodierung (ceDOS, ceANSI, or ceUnicode).

file.eof(handle) : Testet ob das Dateiende erreicht wurde (end-of-file) .

Liefert den Wert FALSE (0) solange das Datei-Ende **noch nicht** erreicht wurde,
bzw TRUE (1) wenn beim letzten Leseversuch (z.B. per file.read_line) das Datei-Ende erreicht wurde.
(Hinweis: Wenn file.read_line() eine leere Zeile einliest, dann bedeutet das 'noch lange nicht', dass das Ende der Datei erreicht wurde !)
Ein Beispiel zur Verwendung von 'file.eof' finden Sie unter file.open .

file.seek(handle, offset, fromwhere) : Setzt den Dateizeiger.

'offset' ist die neue 'absolute' oder 'relative' Dateiposition, gemessen in Bytes.
'fromwhere' (oder 'origin') definiert die Bedeutung von 'offset'. Dieser Parameter darf einer der folgenden Integer-Konstanten sein:

- **SEEK_SET** Positionierung relativ zum Dateianfang (offset 0 = erstes Byte in der Datei)
- **SEEK_CUR** Positionierung relativ zur aktuellen Position (offset 0 = 'keine Bewegung')
- **SEEK_END** Positionierung relativ zum Dateiende (offset 0 wäre das Ende der Datei)

Der Rückgabewert von file.seek gibt die neue, absolute Position innerhalb der Datei an, gemessen vom Anfang der Datei.
Bei 'SEEK_END' darf der Offset nur Null (d.h. Sprung an's Dateiende) oder *negativ* sein ! Ein positiver Offset im Kombination mit 'SEEK_END' wäre eine (illegale) Position jenseits des Datei-Endes.
Ein Beispiel zur Verwendung von file.seek finden Sie in programs/script_demos/FileTest.cvt .

file.close(handle) : Schließt die angegebene Datei, und gibt deren *Datei-Handle* wieder frei.

Ein Beispiel zur Verwendung dieses Kommandos finden Sie unter file.create .
Vergessen Sie niemals, Dateien nach deren 'Gebrauch' wieder zu schließen ! Besonders dann, wenn neue Daten in die Datei geschrieben wurden, könnte noch ein Teil der Daten in einem internen Puffer stehen, der erst dann in die Datei zurückgeschrieben werden kann, wenn ein 'Sektor' des Dateisystems komplett ist. Erst beim Schließen der Datei werden die Puffer in die Datei kopiert, und (bei "echten" Dateien) der Eintrag im Directory und in der FAT (file allocation table) aktualisiert. Sind beim Abschalten des Gerätes noch Dateien öffen, könnte der Inhalt des Speichermediums beschädigt werden !

Per Default gehen die Funktionen file.open und file.read_line davon aus, daß Textdateien 'reinen Text' (plain text) mit 8 Bit pro Zeichen enthalten. Esoterische Formate wie '*.doc' oder '*.docx' werden nicht (niemals!) unterstützt. Ist das Flag O_TEXT im zweiten Parameter beim Aufruf von file.open gesetzt, dann untersucht die Funktion den Inhalt der Datei (genauer: den Anfang der Datei), und versucht dort die sogenannte Byte Order Mark (BOM) zu finden. Ist eine BOM vorhanden, so wird diese automatisch überlesen (damit sie nicht als 'Müll' beim ersten Aufruf von file.read_line am Anfang der gelesenen Zeichenkette auftaucht), und anhand der BOM der Encoding-Typ ("Dateityp der Textdatei") erkannt. Unterstützt werden die folgenden Textformate:

- UTF-16 mit Big-Endian byte order (BOM = 0xFE, 0xFF)
- UTF-16 mit little-endian byte order (BOM = 0xFF, 0xFE)
- UTF-8 (Pseudo-BOM = 0xEF, 0xBB, 0xBF)

Enthält die Textdatei Unicode-Text (in einem der oben aufgelisteten Formate), dann werden die per file.read_line eingelesenen Strings (hier: "Zeilen") automatisch in UTF-8 umgesetzt, da UTF-8 der einzige, von der String-Library unterstützte 'Unicode-fähige' String-Typ ist.
Benötigen Sie weitere Informationen zu den Unicode-fähigen Textdateien, lesen Sie bitte den Wikipedia-Artikel zum Thema Byte order mark .

4.10.5.4 Lesen von Verzeichnissen

Zum Lesen des Inhaltsverzeichnisses einer Speicherkarte (oder eines ähnlichen Speichermediums) dienen die folgenden Funktionen:

directory.open(string path_and_mask)
Öffnet ein *Directory* (Verzeichnis) zum Lesen. Liefert im Erfolgsfall einen positiven Integer-Wert ("Handle"), der für die nachfolgenden Aufrufe von directory.read() als Parameter übergeben werden muss.

directory.read(int handle, tDirEntry *dir_entry)
Liest den nächsten Verzeichniseintrag (Dateiname, Größe, Datum, Uhrzeit, Attribute, ...). Das zweite Argument muss ein *Pointer auf eine Variable vom Typ tDirEntry* sein (siehe folgendes Beispiel).
Liefert im Erfolgsfall TRUE, andernfalls FALSE (d.h. Ende des Verzeichnisses erreicht).

directory.close(int handle)
Beendet das Lesen des Verzeichnisses, und gibt per directory.open belegte Resourcen wieder frei. Nicht vergessen !

Beim Lesen von Verzeichniseinträgen wird bei jedem Aufruf von directory.read() die Dateibeschreibung in einer Struktur mit dem Typ 'tDirEntry' ("Typ für einen Verzeichniseintrag") abgelegt.
Eine Variable des Typs **tDirEntry** enthält die folgenden Komponenten:

name	Dateiname im klassischen 'DOS'-Format (8+3 Buchstaben)
attributes	'DOS'-kompatible Dateiattribute. Bitweise Kombination aus cFileAttr-Konstanten.

year	Jahr der letzten Änderung
month	Monat (1..12) der letzten Änderung...
mday	Tag des Monats (1..31)
hour	Stunde (0..23)
minute	Minute (0..59)
sec	Sekunde (0..59)
size	Dateigröße, gemessen in **Bytes**
medium	Speichermedium. Nur für Testzwecke.

Ein Beispiel zum Einsatz von directory.open / read / close finden Sie im kommentierten Quelltext der Funktion ReadDir() im Beispielprogramm 'App-Selektor'.

Siehe auch: Funktionen zur Datei-Ein-/Ausgabe (Übersicht), Pseudo File System, Zugriff auf Dateien per Web-Server,
Verarbeitung von Zeichenketten, Schlüsselwörter, Inhalt .

9.6 4.10.6 Empfang und Senden von CAN-Telegrammen (per Script)

Hinweis: Nicht alle programmierbaren Geräte erlauben den 'direkten' Empfang von CAN- bzw. LIN-Bus-Telegrammen per Script-Sprache, wie er in diesem Kapitel vorgestellt wird.
Geräte mit CANopen-Protokoll unterstützen die im Folgenden erläuterten Funktionen (mit der Ausnahme von CAN.status) ***nicht !***

Darüberhinaus können die hier beschriebenen CAN-Funktionen erst nach **Freischaltung** verwendet werden (zumindest bei Geräten wie MKT-View II,III,IV). Entsprechendes gilt auch für Geräte mit LIN-Bus-Unterstützung.

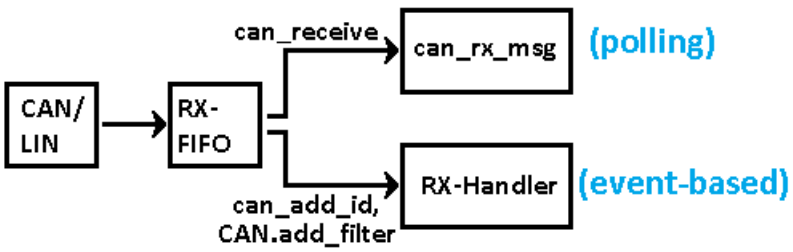
Übersicht der wichtigsten CAN-Funktionen : can_add_id, can_receive, can_transmit .

Mit dem Kommando can_add_id oder CAN.add_filter registrieren Sie alle CAN- bzw. LIN-Bus-Identifier, die Sie mit der Funktion can_receive in der Script-Sprache, oder durch den Aufruf eines Empfangs-Handlers verarbeiten wollen.
Für LIN (statt CAN) ist dabei die Maske cCanIdBit_LIN zu setzen. Im weiteren Verlauf dieser Beschreibung kann 'CAN-Telegramm' sowohl für 'CAN-Message' wie auch 'LIN-Frame' bedeuten. Wie in der LIN-Spezifikation wird auch hier die Bezeichnung 'LIN-Message' vermieden, denn eine 'Message' wird bei LIN für die Transportschicht ('transport layer messages') verwendet.

Nach der o.g. 'Registrierung' eines oder mehrerer CAN/LIN-Identifier kann mit der Funktion can_receive getestet werden, ob CAN- bzw. LIN-Telegramme im Empfangspuffer (FIFO) bereit stehen, und ggf. das nächste Telegramm auslesen. Das per 'can_receive' aus dem FIFO gelesene Telegramm steht danach in der globalen Variablen (Struktur) can_rx_msg zur Verarbeitung in der Script-Sprache bereit.

Solange das Script nicht erneut die Funktion can_receive (zum Empfang des nächsten Telegramms) aufruft, ändert sich der Inhalt von can_rx_msg nicht. Das Umkopieren von can_rx_msg erübrigt sich dadurch. Der Inhalt des empfangenen Datenfelds (in can_rx_msg) kann auf verschiedene Arten adressiert werden (z.B. als Byte-Array, Bitfeld, etc).

Zur Übergabe von CAN-Telegrammen beim Senden und Empfangen (auch per Handler, s.U.) dient der Datentyp tCANmsg .



CAN-Empfang per Script durch Polling oder Events

Um besonders schnell auf bestimmte CAN-Telegramme zu reagieren, kann für bestimmte CAN-Identifier ein CAN-Empfangs-Handler im Script implementiert werden. Dadurch können hoch priorisierte Telegramme (im Handler) verarbeitet werden, bevor niedriger priorisierte Telegramme (per Polling) aus dem Empfangs-FIFO verarbeitet werden.

In normalen *Display*-Applikationen wird das Empfangen und Senden von CAN-Telegrammen

durch eine Datenbank gesteuert, die im Programmiertool auf der Registerkarte 'CANdb' aus einer Datei (*.dbc) importiert werden kann. Das Script kann direkt auf die daraus erzeugten Anzeige-Variablen zugreifen (per display.<Signal- bzw. Variablenname>). Darum braucht sich das Script in den meisten Fällen (wenn der Aufbau der CAN-Telegramme per DBC-Datei beschrieben werden kann) nicht um den Aufbau der CAN-Telegramme zu kümmern. In einigen Fällen (z.B. in "Gateway"-ähnlichen Anwendungen, und für Rest-Bus-Simulationen mit speziellen CAN-Protokollen) kann das Script CAN-Telegramme ("Messages") basierend auf den Informationen aus der CAN-Datenbank selbst zusammenstellen (-> CAN.EncodeMessage) oder decodieren (-> CAN.DecodeMessage).
Ist keine passende DBC-Datei vorhanden, kann das Script notfalls auch über die Bitfeld-Komponente auf das Datenfeld eines empfangenen oder zu sendenden CAN-Telegramms zugreifen, und die Umrechnung zwischen physikalischem und per CAN übertragenen Wert ggf. "selbst" durchführen.

9.7 4.10.6.1 can_add_id(<CAN-ID>) : Registrieren eines CAN-Message-Identifiers für den Empfang

Fügt den angegebenen CAN-Message-Identifier zu einer internen Liste hinzu, in der alle *im Script* zu empfangenden Telegramme registriert sind. Dadurch werden ab dem Aufruf von `can_add_id( < CAN-ID >)` alle CAN-Telegramme mit passendem Message-Identifier *auch* per Script empfangen. Ein interner FIFO mit bis zu 2047 Einträgen dient dabei als Zwischenspeicher zwischen CAN-Treiber (Software) und Script (Display-Applikation).
Das Script-Programm empfängt *nur* CAN-Messages, deren Identifier per can_add_id oder CAN.add_filter registriert wurden. Alle CAN-Messages werden weiterhin wie gewohnt auch in anderen Firmware-Modulen verarbeitet (z.B. im "CANdb-Signal-Decoder" für die Anzeige, im internen CAN-Logger, oder im CANopen-Protokoll-Stack). Der Empfang von CAN-Telegrammen im Script hat daher *keine* Auswirkung auf die 'normale' Verarbeitung von empfangenen CAN-Messages.
Es können maximal 32 verschiedene CAN-Message-Identifier für den (zusätzlichen) Empfang im Script registriert werden (Stand: 2011-05-05).

In den höherwertigen Bits (31..29) des Parameters 'CAN-ID' werden Bus-Nummer und 11/29-Bit-Flag ("Extended") codiert. Dazu können die Script-Konstanten cCanIdBit_Bus2, cCanIdBit_Bus3, cCanIdBit_Extd bitweise mit dem eigentlichen Message-ID (in Bit 10..0 bzw 28..0 des Parameters) verknüpft werden.
Beispiele:

```
   can_add_id(  0x123 );                                  // start receiving 11-bit ID 0x123 on CAN1
   can_add_id( cCanIdBit_Bus2 | cCanIdBit_Extd | 0x123 ); // start receiving 29-bit ID 0x123 on CAN2
```

Optional kann beim Registrieren zu empfangender CAN-Message-Identifier auch der Name eines CAN-Empfangs-Handlers übergeben werden. Beispiel:
can_add_id(0x123, addr(CAN_Handler_ID123)); // Call 'CAN_Handler_ID123' on reception of this message ID

9.8 4.10.6.2 CAN.add_filter(<filter>, <mask>, <receive_handler>)

Ähnlich wie can_add_id; diese Funktion registriert allerdings *einen* kompletten **Bereich** (range) von CAN-Message-Identifiern, der vom Script empfangen werden soll.

Funktionsweise des so definierten Filters:

Der empfangene CAN-Message-Identifier wird zunächst mit der Maske (mask) bitweise UND-verknüpft.
Das Ergebnis dieser Verknüpfung wird mit dem 'Filter'-Wert (filter) verglichen.
Wenn alle Bits (*nach* der bitweisen UND-Verknüpfung) übereinstimmen, wird die empfangene CAN-Message an das Script weitergeleitet.

Mit anderen Worten: Alle in 'mask' **gelöschten** Bits bedeuten 'don't care' (d.h. "passen immer");
alle in 'mask' **gesetzten** Bits müssen beim Vergleich von 'filter' mit dem empfangenen CAN-Message-ID übereinstimmen.

Beispiele:

```
CAN.add_filter( 0x2ABCD00, 0x2FFFFF00 ); // receive extended IDs 0x0ABCD00 to 0x0ABCDFF
CAN.add_filter( 0x000, 0x000, addr(MyCanRxHandler) ); // receive standard IDs with a handler
// Note: As in other CAN functions of the script language,
// the 'extended' flag which indicates a 29-bit-ID is encoded in bit 29,
// and the bus number (0..3) is encoded as a two-bit value in bits 31..30 of the ID.
```

Bislang kann pro CAN-Interface nur **ein** derartiges Filter für einen Bereich von CAN-Message-Identifiern definiert werden. Die vom Filter 'durchgelassenen' CAN-Messages werden *zusätzlich* zu den per can_add_id definierten CAN-Telegrammen im Script empfangen.
Diese Funktion wurde im September 2013 implementiert, um *per Script* ein J1939-kompatibles Protokoll zu ermöglichen. J1939 codiert etliche Informationen (z.B. die 'Adresse des Senders', 'SA') im 29-Bit-CAN-Message-Identifier.
Die im CAN-Controller 'hardwaremäßig' implementierte CAN-Filterung wird dadurch fast unbrauchbar, denn ein nichttriviales 'J1939-Diagnosesystem' kennt die zu empfangenen CAN-Message-IDs bei der Initialisierung i.A. noch nicht, und ist daher dazu verdammt(!), 'fast alle' CAN-Bus-Telegramme durch das Akzeptanzfilter passieren zu lassen, und muss die weitere 'Verzweigung nach empfangenem CAN-ID' der Applikation (hier: dem Script) überlassen.
Die Folge ist eine enorme CPU-Last (wenn wirklich 'alle' CAN-Telegramme vom J1939-Diagnose-System empfangen werden); Sie sollten daher auf den Einsatz des Befehls *CAN_add_filter* verzichten, wenn er nicht *unbedingt* zur Implementierung eines exotischen Protokolls (wie z.B. für ein J1939-Diagnosesystem) benötigt wird;
oder zumindest den vom Filter 'durchgelassenen' Bereich von CAN-Identifiern nur so groß machen wie unbedingt nötig.
Ein Beispiel zum Einsatz der Funktion **CAN.add_filter** finden Sie in der Applikation 'J1939-Simulator'.
Per CAN.add_filter() registrierte Empfangs-Handler werden nur aufgerufen, wenn keiner der per

can_add_id() registrierten Handler für das empfangene Telegramm den Wert TRUE lieferte. Details zur Reihenfolge des Aufrufs von CAN-Empfangs-Handlern finden Sie hier.

9.9 4.10.6.3 can_receive (Funktion zum CAN-Empfang per "Polling")

Versucht, das nächste CAN-Telegramm aus dem Script-eigenen CAN-Empfangs-FIFO zu lesen. Bei Erfolg wird das empfangene CAN-Telegramm aus dem FIFO in die Variable can_rx_msg umkopiert, und die Funtion kehrt mit dem Wert 1 (Eins), bzw TRUE zum Aufrufer zurück. Andernfalls (Empfangs-FIFO leer) bleibt der Inhalt von 'can_rx_msg' unverändert, und die Funktion kehrt mit dem Wert 0 (Null), bzw. FALSE, zum Aufrufer zurück.

Hinweis: Werden empfangene CAN-Telegramme in einem selbstdefinierten CAN-Empfangs-Handler verarbeitet, dann ist dort *kein* Aufruf der Funktion **can_receive** nötig ! Das empfangene Telegramm wird dem Handler per Pointer (als Argument) übergeben, und -sofern der Handler mit dem Wert 1 bzw. TRUE zurückkehrt- ***nicht*** in den CAN-Empfangs-FIFO des Scripts kopiert !

9.10 4.10.6.4 can_rx_fifo_usage (Funktion)

Liefert die Anzahl momentan noch im CAN-Empfangspuffer ("RX-FIFO") wartender CAN-Telegramme, ohne den Inhalt des FIFOs zu beeinflussen.
Der Rückgabewert 0 (Null) bedeutet "der Empfangspuffer ist im Moment komplett leer".
Der Rückgabewert 2047 (!) bedeutet "der Empfangspuffer ist komplett voll" (was meistens bedeutet, dass bereits empfangene CAN-Telegramme zumindest für die Script-Anwendung verloren gingen weil die Funktion 'can_receive' nicht oft genug aufgerufen wurde).

4.10.6.5 **can_transmit** (Prozedur)

Variante 1: Aufruf von **can_transmit ohne Parameter**:
Die globale Variable 'can_tx_msg' mit Inhalt füllen (CAN-Sende-Telegramm mit der weiter Unten definierten Struktur), und dann senden.

```
      can_tx_msg.id  := 0x334;  // set CAN message ID (and bus number in the
upper bits)
      can_tx_msg.len := 2;      // set the data length code (number of data
bytes)
      can_tx_msg.b[0] := 0x11;  // set the first data byte
      can_tx_msg.b[1] := 0x22;  // set the second data byte
      can_transmit;  // send the contents of can_tx_msg to the CAN bus
```

Variante 2: Aufruf von **can_transmit mit *einem* Parameter**: Statt der globalen Variablen 'can_tx_msg' eine eigene (hier: lokale) Variable vom Typ 'CAN-Telegramm' mit Inhalt füllen, und deren Adresse als Parameter an die Prozedur 'can_transmit(<Adresse der zu sendenden Message>) übergeben.
Im folgenden Beispiel wird can_transmit mit Parameter aus einem CAN-Empfangs-Handler aufgerufen, wobei die zu sendende Message (als *lokale* Variable) übergeben wird:

```
//-------------------------------------------------------------------
func CAN_Handler_A( tCANmsg ptr pRcvdCANmsg )
  // A CAN-Receive-Handler for a certain CAN message identifier.
```

```
   // Must be registered via 'can_add_id', along with the CAN message ID.
   // Interrupts the normal script processing, and must RETURN to the caller
   //  a.s.a.p. ! Uses a LOCAL variable for transmission (not can_tx_msg).
   // Thus can_tx_msg can safely be used in the script's main loop,
   // even if the main loop may be interrupted at any time by this handler.
   local tCANmsg responseMsg;    // a local variable with type 'CAN message'
   responseMsg := pRcvdCANmsg[0];      // copy the received CAN message into the
response
   responseMsg.id := pRcvdCANmsg.id+1; // response CAN ID := received CAN ID + 1
      // Hinweis: Bits 31 und 30 in tCANmsg.id enthalten die BUS-NUMMER.
      //  Wie üblich beginnt auch hier die Zählung bei NULL (für das erste CAN-
Interface) !
   can_transmit( responseMsg );  // send a response via CAN immediately
   return TRUE;
   // returning TRUE means: "the received message was processed HERE,
   //              do NOT place it in the script's CAN-receive-FIFO".
 endfunc; // end CAN_Handler_A

      ... irgendwo im Initialisierungsteil : ...

   // Register a received CAN ID, and install a CAN-receive-handler for it:
   can_add_id( C_CANID_RX_A, addr(CAN_Handler_A) );
```

Variante 3: Aufruf von **can_transmit mit *zwei* Parametern**: Zusätzlich zur Adresse des zu sendenden Telegramms wird hier noch (als zweiter Parameter) eine 'Sende-Option' angegeben:

```
       can_transmit( responseMsg, cCanTx_Normal ); // Wenn nötig, warten bis TX-
Puffer frei
       can_transmit( responseMsg, cCanTx_NoWait );  // Nicht warten. Bei vollen
Puffer nichts senden.
```

Zum Zeitpunkt der Erstellung dieses Dokuments (2016-06-09) standen die folgenden Optionen zur Verfügung:

- cCanTx_Normal : Wenn nötig, wartet can_transmit für einige Millisekunden bis der CAN-Sendepuffer für die Aufnahme des zu sendenden Telegramms bereit ist.
  Dank des großen Sendepuffers (FIFOs), kann das Script beim Senden annähernd 100 Prozent Buslast erzeugen, ohne das ein einziges zu sendendes Telegramm verloren geht, denn die Hautpschleife des Script wird durch das eventuell 'blockierende' can_transmit-Kommando entsprechend ausgebremst, so dass kein FIFO-Überlauf stattfinden kann.
- cCanTx_NoWait : Nicht warten, selbst denn der Sendepuffer beim Aufruf von can_transmit bereits komplett gefüllt war. Stattdessem wird das zu sendende Telegramm *verworfen*.
  Dies kann passieren, wenn das Script versucht, mehr Telegramme zu senden als der CAN-Bus 'verkraftet' (abhängig von Baudrate und Buslast), oder wenn noch kein zweiter CAN-Knoten aktiv ist, der die gesendeten Telegramme bestätig (d.h. kein "Acknowledge", was dazu führt, das der Sender den Sendeversuch wiederholt, bzw. die zu sendenden Telegramme "nicht loswird").

Die Option cCanTx_NoWait sollte z.B. verwendet werden, wenn can_transmit aus einem Event-Handler aufgerufen wird, z.B. aus einem Timer-Event-Handler.

Ein Beispiel zum *periodischen* Senden von CAN-Telegrammen finden Sie in der Applikation 'TimerEvents.cvt'.

Ein Beispiel zum *ereignisgesteuerten* Senden von CAN-Telegrammen (per Buttons) finden Sie in der Applikation 'ButtonEventDemo.cvt'.

Ein Beispiel zum Eingeben eines Wertes in einem Editierfeld (mit Begrenzung), welcher dann über CAN verschickt wird, finden Sie in 'SendCANFromEditField.cvt'.

Bitte beachten: Normalerweise kehrt can_transmit() 'sofort' zum Aufrufer zurück, d.h. bevor das CAN-Telegramm physikalisch vom CAN-Controller gesendet wurde.
Grund: Alle zu sendenden Telegramme (nicht nur die per Script gesendeten) werden zunächst in einem interruptgetriebenen CAN-Sende-Puffer plaziert. Das eigentliche (physikalische) Senden erfolgt dann i.a. einige Mikrosekunden später, sobald der CAN-Bus frei ist (und alle bereits vorher in den Puffer eingetragenen CAN-Telegramme erfolgreich gesendet wurden).
Aus diesem Grund kann die Prozedur `can_transmit` auch keinen Status-Code an den Aufrufer zurückliefern ("Senden erfolgreich oder nicht").
Mit der Funktion CAN.status, Bitmaske cCANStatusTxError, kann ggf abgefragt werden, ob beim Senden von CAN-Telegrammen Fehler aufgetreten sind.

Um statt eines 'normalen' CAN-Telegramms einen RTR-Frame (Remote Transmit Request) zu senden, verwenden Sie die Konstante cCanRTR in einer bitweisen ODER-Kombination mit dem Längen-Feld in der zu sendenden Message.

Verwenden Sie diese Funktion nur, wenn Sie sich der möglichen Folgen bewusst sind !
Das Senden eines 'falschen' CAN-Telegramms kann unvorhersehbare Folgen haben !

Siehe auch: CAN.EncodeMessage zum Zusammenstellen der CAN-"Nutzdaten" anhand einer vorher importierten Datenbank.

9.11 4.10.6.6 can_rx_msg, can_tx_msg (globale Variablen für CAN)

Auch Sicht des Scripts enthält die Variable 'can_rx_msg' das letzte empfangene CAN-Telegramm. Der Inhalt wurde beim letzten erfolgreichen Aufruf der Funktion can_receive mit neuen Daten gefüllt. Das Script kann damit 'ungestört' auf CAN-ID, Länge des Datenfeldes, und die Nutzdaten zugreifen ('ungestört' da sich der Inhalt von 'can_rx_msg' niemals ohne erneuten Aufruf von 'can_receive' ändern kann).
Die folgenden Komponenten von can_rx_msg (bzw can_rx_msg zum Senden) sind im Script-Programm wie Variablen verwendbar. Nur ein Teil dieser Komponenten
(.id, .len, .tim, .b[], .w[], .dw[]) ist auch in der *Datenstruktur* vom Typ tCANmsg vorhanden !

`.id`

Enthält den CAN-Message-Identifier in den niederwertigen 11 (oder 29) Bits, zusätzlich die 11/29-Bit-Kennung ("extended CAN identifier flag") in bit 29, und die CAN-BUS-NUMBER (!) in den höchstwertigen Bits (Bits 31..30; Zählung beginnt auch hier bei Null) .

Bitte beachten: Bits werden wie allgemein üblich von 0 (niederwertiges Bit, LSB) bis 31 (höchstwertiges Bit, MSB) gezählt. Die Bitwertigkeit ergibt sich dann aus der Zweierpotenz der Bitnumer. In einer 32-Bit-Zahl gibt es daher kein "Bit Nummer 32" !
Um den Script-Quelltext verständlich zu halten, sollten im Identifier statt hexadezimaler Werte vorzugsweise die symbolischen Konstanten cCanIdBit_Bus2 und cCanIdBit_Bus3 verwendet werden.
Bei LIN muss der dort 'frame ID' genannte Identifier zwischen 0 und 63 (0x3F) liegen; zusätzlich muss der Identfier-Wert (0..63) bitweise mit der Maske cCanIdBit_LIN 'verodert' werden, um das Telegramm für den kombinierten CAN-/LIN-Treiber als LIN-'Frame' zu kennzeichnen.

.len

Länge des Datenfeldes der Message in Bytes. Die Länge einer einzelnen CAN-message kann zwischen 0 Bytes (d.h. nur Message-ID, aber keine "Nutzdaten") und 8 Bytes liegen.
Die höherwertigen Bits dieses Feldes *können* spezielle Flags wie cCanRTR enthalten (Remote Transmission Request, siehe Beispiel in ScriptTest3.cvt, SendRTR()).

.tim

Zeitstempel der empfangenen CAN-Message, im hardwarespezifischen Format. Je nach Taktfrequenz des Zeitmarken-Timers (cTimestampFrequency, s.U.) läuft dieser **32-Bit-Integer-Wert** nach ca. 14 bis 30 Stunden ununterbrochenen Betriebs von 0xFFFFFFFF nach 0x00000000 über.
Um die Differenz zwischen zwei Zeitmarken in Sekunden umzurechnen, bilden sie *erst* die Differenz als 32-Bit-Integer-Wert, und rechnen diesen *danach* (***nach*** der Subtraktion zweier Integer-Werte) per Division durch 'cTimestampFrequency' in Sekunden um (lesen Sie bitte unbedingt auch diesen Hinweis).
Auch andere Script-Funktionen verwenden das gleiche Format für Zeitmarken, z.B. system.timestamp (liefert die "aktuelle", systeminterne Zeitmarke) .

.b[N**]**

Ermöglicht den Zugriff auf das N-te Byte (N : 0..7) im CAN-Datenfeld. Jedes Byte kann einen Wert zwischen 0 und 255 enthalten (8 Bit unsigned integer).
Der einfache byte-weise Zugriff auf das CAN-Datenfeld ist auch bei 'normalen' Variablen des Typs tCANmsg möglich. Soll allerdings ein komplettes CAN-Datenfeld kopiert werden, empfiehlt sich der der Zugriff in zwei 32-Bit-Wörtern (.dw):

.dw[N**]**

Greift auf das N-te **Doubleword** (N : 0..1) im CAN-Datenfeld zu. Ein Doubleword ('DWORD') besteht aus 32 Bit, der Wertebereich liegt zwischen 0x00000000 und 0xFFFFFFFF (hex).
Beachten Sie, daß (im Gegensatz zu den weiter Unten aufgeführten Komponenten) der Array-Index in diesem Fall ein 'doubleword'-Index, kein Byte-Index, ist.
Beispiel zum Umkopieren des kompletten 8-Byte-Datenfelds (aus 'ScriptTest3.CVT') :

```
// The 8 databytes are copied as two 32-bit integers,
// because that's faster on an ARM-CPU than a byte-copying-loop:
```

```
response.dw[0] := can_rx_msg.dw[0]; // copy first doubleword (4 bytes)
response.dw[1] := can_rx_msg.dw[1]; // copy second doubleword (4 bytes)
```

Im Gegensatz zu den weiter unten beschriebenen Komponenten ist der oben beschrieben DWORD-weise Zugriff auf das CAN-Datenfeld auch bei 'normalen' Variablen des Typs tCANmsg möglich.

.i16[N**]**
Greift auf das N-te und (N+1)-te **Byte** im CAN-Datenfeld zu, wobei diese beiden Bytes als ein einzelner ***vorzeichenbehafteter*** 16-Bit-Integer-Wert aufgefasst werden, mit 'Intel'-Byte-Order (Byte-Reihenfolge = Little-Endian, d.h. 'niederwertiges Byte zuerst').
Beim Lesezugriff wird das Vorzeichen aus Bit 15 in die höherwertigen Bits der resultierenden Integer-Zahl expandiert; das Ergebnis liegt daher immer zwischen -32768 bis +32767.

.u16[N**]**
Greift auf das N-te und (N+1)-te **Byte** im CAN-Datenfeld zu, wobei diese beiden Bytes als ein einzelner ***vorzeichenloser*** 16-Bit-Integer-Wert aufgefasst werden (unsigned), mit 'Intel'-Byte-Order (Byte-Reihenfolge = Little-Endian, d.h. 'niederwertiges Byte zuerst').
Im Gegensatz zu 'i16' (s.O.) wird hier KEIN Vorzeichen expandiert, der resultierende Wertebereich ist daher 0 bis 65536.

.i32[N**]**
Greift auf das N-te bis (N+3)-te **Byte** im CAN-Datenfeld zu, und behandelt diese wie einen einzelnen ***vorzeichenbehafteten*** 32-Bit Integer-Wert. Dabei wird, wie bei .16, die 'Intel'-Byte Order verwendet (Little-Endian, least significant byte first).
Hinweis: Die obigen Methoden erlauben einen sehr schnellen und effizienten Zugriff auf das CAN-Datenfeld, sind aber nicht in der Lage auf beliebige *Bit*-Positionen im Datenfeld zuzugreifen (der Zugriff erfolgt Byte-, nicht Bit-weise). Die weiter Unten aufgeführte "bitfield"-Methode ist universeller einsetzbar, aber langsamer.

.m16[N**]**
Ähnlich wie '.i16', verwendet das 'Motorola'-Format (Byte-Reihenfolge 'Big Endian', most significant byte first). Dieses Format ist glücklicherweise immer weniger verbreitet...

.m32[N**]**
Ähnlich wie '.i32', allerdings für Integer-Werte im 'Motorola'-Format (s.O.).

.bitfield[<Bit-Index des LSBs> **,** <Anzahl Datenbits>**]**
Zugriff auf einen Teil des CAN-Datenfeldes als 'Bitfeld', welches einen einzelnen *nicht vorzeichenbehafteten* Integer-Wert im 'Intel'-Format enthält (niederwertiges Bit zuerst, höchstwertiges Bit zuletzt).
Unabhängig von Datentypen, Motorola/Intel-Chaos, und der tatsächlichen Länge des CAN-Datenfeldes werden die Bits im CAN-Datenfeld hier *immer* entsprechend der folgenden Tabelle gezählt (im Gegensatz zu manchen "CANdb"-Dateien !). Wie bei Binärzahlen üblich, steht das MSB (höherwertige Bit) eines jeden Bytes *in dieser graphischen Darstellung* auf der linken Seite; und die Numerierung der Bits eines Bytes läuft von 0 (=LSB, rechts) bis 7 (=MSB, links).
Grün markiert sind alle Zellen, die zum folgenden Bitfeld (als Beispiel) gehören:

`can_rx_msg.bitfield[ 18, 15 ]`
("Intel"-Format, LSBit an Bit-Index 18 im CAN-Datenfeld, Länge des Feldes = 15 bit).

Zählweise für die Bits in einem CAN- oder LIN-Datenfeld.
(**grün**: 15-bit signal; .bitfield[18, 15])

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
byte[0]	7	6	5	4	3	2	1	0
byte[1]	15	14	13	12	13	10	9	8
byte[2]	23	22	21	20	19	18	17	16
byte[3]	31	30	29	28	27	26	25	24
byte[4]	39	38	37	36	35	34	33	32
byte[5]	47	46	45	44	43	42	41	40
byte[6]	55	54	53	52	51	50	49	48
byte[7]	63	62	61	60	59	58	57	56

Die Unterstützung für Bitfelder im 'Motorola'-Format (most significant byte first) war zum Zeitpunkt der Erstellung dieser Datei nicht geplant. Mit Hilfe der Bit-Schiebe-Operatoren, und der bitweisen UND- / ODER-Verknüpfung, können Sie notfalls selbst (im Script) eine Integer-Zahl im Motorola-Format bitweise zerlegen oder zusammenfügen.

Die globale Variable 'can_tx_msg' hat die gleiche Struktur wie oben für 'can_rx_msg' beschrieben. Der einzige Unterschied ist der Einsatzzweck: can_rx_msg dient zum Empfang, und can_tx_msg zum Senden von CAN-Telegrammen (aus sicht des Scripts). Beide werden typischerweise zur Implementierung eigener 'CAN-Protokolle' in der Script-Sprache verwendet; auch für maschinell erzeugte Scripte mit 'CCP'/'XPC'.

Die Applikation 'ScriptTest3.cvt' enthält einige Beispiele zum Senden und Empfangen von CAN-Telegrammen per Script.

Hinweise und Tipps:

Zum Testen der CAN-Funktionen in Ihrer Script-Applikation empfehlen wir den Anschluss eines geeigneten CAN-Interfaces am PC (z.B. von Kvaser). Dann kann das Programmiertool als 'echte' Simulation (inklusive CAN-Funktionalität) verwendet werden. Der spezielle CAN-Empfangspuffer funktioniert mit der im Programmiertool integrierten Script-Runtime-Library, d.h. CAN-Telegramme werden entsprechend bis zum im Debugger laufenden Script durchgeschleust.
Sollte das nicht möglich sein (z.B. weil kein geeignetes CAN-Interface am PC angeschlossen werden kann), kann alternativ die CAN-Playback-Funktion im Programmiertool verwendet werden. Damit können vorher als 'Konserve' aufgezeichnete CAN-Telegramme abgespielt werden, als ob sie von einem 'echten' CAN-Netzwerk empfangen wurden. Das Testen selbstentwickelter CAN-Protokolle (mit Handshake) funktioniert damit allerdings nicht.

Hilfreich bei der Implementierung eigener CAN-Protokolle in der Script-Sprache kann auch die von fast allen Geräten unterstützte Trace-Historie sein. Diese kann bei Remote-Betrieb auch per Web-Browser ausgelesen werden, z.B. unter der URL http://upt/trace.htm . Im Gegensatz zu externen CAN-Diagose- Tools wird bei der Anzeige in der Trace-Historie

zwischen (aus Sicht des Terminals) gesendeten und empfangenen CAN-Messages unterschieden.

9.12 4.10.6.7 CAN.DecodeMessage(tCANmsg ptr msg)

Dieses Kommando kann z.B. aus einem CAN-Empfangs-Handler aufgerufen werden, wenn die im empfangenen CAN-Telegramm enthaltenen Signale *sofort* (d.h. noch innerhalb des Handlers) in die entsprechenden *Display-Variablen* umgerechnet werden sollen. Dazu wird die per Programmiertool importierte CAN-Datenbank (*.DBC, aka 'CANdb'-Datei) nach einem passenden CAN-Message-Identifier durchsucht, und alle mit der Message verknüpften Display-Variablen aktualisiert.
Dies kann in besonders anspruchsvollen Anwendungen nötig sein, denn der CAN-Empfangs-Handler wird vom Laufzeitsystem aufgerufen *bevor* die Message (per FIFO) zum Decodieren an das Anzeigemodul übergeben wird.
Wird kein passender CAN-Message-Identifier in der Datenbank gefunden, dann ändert CAN.DecodeMessage 'nichts' (in den Display-Variablen), verbraucht aber je nach Größe der importierten Datenbank eine nicht unwesentliche Zeit.
Aus dem Grund sollte CAN.DecodeMessage() nur aufgerufen werden, wenn dies für die weitere Signalverarbeitung im CAN-Empfangs-Handler wirklich nötig ist. Für die normale Anzeigefunktionalität ist der Aufruf von CAN.DecodeMessage nicht nötig.
Wurde eine passende Message-Definition in der Datenbank gefunden (mit passendem CAN-ID und ggf passendem Multiplexer), dann liefert CAN.DecodeMessage() den Rückgabewert TRUE.
Wurde keine passende Message-Definition gefunden (kein passender CAN-ID oder, bei gemultiplexten Telegrammen, kein passender Multiplexer), dann liefert CAN.DecodeMessage() den Rückgabewert FALSE.

Eingangsparameter:
msg = Addresse der zu decodierenden CAN-Message (aka 'CAN frame', mit bis zu 8 Datenbytes).
Ausgabe:
Alle *Display-Variable* (display.XYZ) die mit der zu decodierenden Message verbunden sind, abhängig von der importierten CAN-Datenbank (.dbc).
Verwendung:
Üblicherweise in einem CAN-Receive-Handler .
Test / Beispiel:
Siehe CAN.EncodeMessage() .

9.13 4.10.6.8 CAN.EncodeMessage(int msgID, tCANmsg ptr msg)

Dieses Kommando ist -mehr oder weniger- die 'Umkehrfunktion' zu CAN.DecodeMessage.
Für den im ersten Argument (msgID) übergebenen CAN-Message-Identifier stellt sie anhand der geladenen CAN-Datenbank ein entsprechendes CAN-Telegramm *im Speicher* zusammen (zweites Argument: Adresse des CAN-Telegramms).
Die im CAN-Telegramm enthaltenen Signale werden aus den entsprechenden Display-Variablen übernommen, wobei sie ggf. noch (abhängig vom Inhalt der geladenen CAN-Datenbank) entsprechend skaliert werden.
Wurde eine passende Message-Definition in der Datenbank gefunden, dann liefert

CAN.EncodeMessage() den Rückgabewert TRUE.
Wenn keine passende Message-Definition gefunden wurde (kein passender Message-ID), liefert CAN.EncodeMessage() den Wert FALSE.

Hinweis: CAN.EncodeMessage() **sendet** keine CAN-Telegramme. Die Funktion stellt lediglich eine 'sendefähige' Botschaft *im Speicher* zusammen !
Um die Botschaft zu senden, kann deren Adresse an die Funktion can_transmit(msg) übergeben werden.
Bits im CAN-Datenfeld, die *nicht* von CAN-Signalen aus der Datenbank belegt sind, bleiben beim Aufruf der Funktion CAN.EncodeMessage() *unverändert*. Der Data Length Code (DLC im CAN-Frame) wird automatisch angepasst.

Funktionsargumente:

msgID : CAN-Message-Identifier der zu erstellenden CAN-Message.
In den höherwertigen Bits sind die Bus-Nummer und das Standard/Extended-Flag enthalten.
msg : Addresse der zu codierenden CAN-Message (aka 'CAN frame' vom Typ tCANmsg).

Weitere Eingangsparameter:

Alle *Display*-Variablen, die in der zu erstellenden Message enthalten sind, abhängig von der importierten CAN-Datenbank (.dbc).

Ein Beispiel für die Verwendung von CAN.EncodeMessage (und CAN.DecodeMessage) ist in der Applikation programs/script_demos/CANstress.cvt enthalten.
Hier ein vereinfachter Ausschnitt aus dem entsprechenden Script, um das Prinzip zu verdeutlichen. Die Signale (bzw. Display-Variablen) 'ThreeSines1', 'ThreeSines2', 'ThreeSines3' wurden aus der Datenbank 'MktStandardSignals1.dbc' importiert. Sie werden im folgenden Script-Fragment zunächst gesetzt (1,2,3), danach in eine CAN-Message übernommen ("gemappt"), und das Ergebnis in Form von drei 16-Bit-Integer-Werten im CAN-Datenfeld überprüft:

```
  // Test CAN.EncodeMessage( int msgID, tCANmsg ptr msg ) :
display.ThreeSines1 := 1; // set display variable (input for CAN.EncodeMessage)
display.ThreeSines2 := 2;
display.ThreeSines3 := 3;
pVarDef := display.GetVarDefinition( "ThreeSines1" ); // get database entry for
this display variable
CAN.EncodeMessage( pVarDef.CAN_Msg_ID, MyTxMessage ); // encode CAN message (but
don't transmit it yet)
  // Check the encoded CAN message (which contains 3 signals here):
if( (MyTxMessage.w[0] != 1) or (MyTxMessage.w[1] != 2) or (MyTxMessage.w[2] !=
3) ) then
  print("\nSomething wrong in the database, or CAN.EncodeMessage() ?");
endif;
  // At this point, the values from 'ThreeSines1' to 'ThreeSines3'
  // have been mapped into the CAN message (MyTxMessage) by CAN.EncodeMessage().
  // On this occasion, also check the inverse function, CAN.DecodeMessage():
display.ThreeSines1 := 0; // clear to see if CAN.DecodeMessage() really sets
these...
display.ThreeSines2 := 0;
display.ThreeSines3 := 0;
CAN.DecodeMessage( MyTxMessage ); // TEST, should overwrite
display.ThreeSines1...3
```

```
if( (display.ThreeSines1 != 1) or (display.ThreeSines2 != 2) or
(display.ThreeSines3 != 3) ) then
  print("\nBug in CAN.DecodeMessage() ?");
endif;
```

Nach dem Aufruf von CAN.EncodeMessage() kann das so zusammengestellte Telegramm dann z.B. per can_transmit gesendet werden:

```
can_transmit( MyTxMessage );
```

9.14 4.10.6.9 CAN.VarNameToMessageID(string sVarName)

Sucht in der geladenen CAN-Datenbank nach dem CAN-Message-Identifier, mit dem die *Display*-Variable mit dem angegebenen Namen auf dem CAN- oder LIN-Bus 'transportiert' wird.
Im Erfolgsfall liefert CAN.VarNameToMessageID den *numerischen CAN-Message-Identifier*, andernfalls cCanInvalidID (= 0xFFFFFFFF, was im Gegensatz zu 'Null' niemals kein gültiger CAN-Message-ID sein kann).
Hinweise:

- In vielen Fällen werden mit *einer* CAN-Message *mehrere* Variablen übertragen.
- Um eine Liste von ALLEN Variablen zu erhalten, die mit einem bestimmten CAN-Message-Identifier übertragen werden, rufen Sie wiederholt die Funktion CAN.MessageIDToVarName() auf, beginnend mit dem 'laufenden Index' n=0 (um den Namen der *ersten* in der Message übertragenen Variablen zu ermitteln).
- In Bits 31 und 30 des Return-Werts (Message-ID) ist die CAN-Bus-Nummer codiert !
- Um weitere Parameter zur Übertragung der Display-Variablen als "CAN-Signal" zu ermitteln (z.B. Message-Layout und Skalierung), kann die Funktion display.GetVarDefinition() verwendet werden.

4.10.6.10 CAN.MessageIDToVarName(int iMessageID, int n)

Sucht in der CAN-Datenbank nach der n-ten Variablen, die mit dem angegebenen CAN-Message-Identifier (iMessageID) auf dem CAN-Bus 'transportiert' wird.
Bei Erfolg gibt CAN.MessageIDToVarName *den Namen* der n-ten Anzeige-Variablen zurück, andernfalls einen leeren String.
Beispiel (aus Script-Test #3):

```
iMsgID := CAN.VarNameToMessageID( "FourSines1" );
// Which OTHER variables are contained in the same CAN message ?
print("\nVariables in Msg ID 0x",hex(iMsgID,8)," :" );
i := 0;
repeat
  sVarName := CAN.MessageIDToVarName( iMsgID, i++ );
  print("\n  ", sVarName );
until sVarName=="";
```

Hinweise:

- Da mit *einer* CAN-Message i.A. *mehrere* Signale übertragen werden:
  Der Aufruf mit dem zweiten Argument n=0 liefert den ERSTEN(!) Namen, n=1 den ZWEITEN, usw.
- Um den CAN-Message-Identifier für einen bestimmten Variablennamen zu ermitteln (z.B. "EngineRPM"), verwenden Sie die Funktion CAN.VarNameToMessageID().
- Auch hier ist in Bits 31 und 30 im Message-ID (Parameter iMessageID) die CAN-Bus-Nummer codiert !

4.10.6.11 Spezialfunktionen zur CAN-Bus-Diagnose

Die meisten der folgenden 'sehr speziellen' CAN-Funktionen sind nur in bestimmten Geräten verfügbar, aber nicht im Programmiertool / Simulator auf dem PC. Das erste Funktionsargument ist üblicherweise die Nummer des zu verwendeten CAN-Ports. Gültige Werte für diesen Parameter sind:

`cPortCAN1` (1. CAN bus),
`cPortCAN2` (2. CAN bus) .

Die Applikation script_demos/ErrFrame.CVT verwendet einige dieser Funktionen, um einen CAN-Bus "absichtlich" mit Error-Frames zu 'traktieren'.

`CAN.status( <port> )`

Liefert den aktuellen Status das CAN-Controllers für den spezifizierten Port (cPortCAN1 oder cPortCAN2).
Der Status ist eine Bitkombination aus den folgenden Konstanten:

OK (0)	'kein Problem' (zumindest nicht mit *diesem* CAN-Port)
HWFault (1)	nicht näher definierbares Problem mit der CAN-Hardware bzw. diesem Port
RxOverflow (2)	Überlauf des CAN-Empfangspuffers
TxOverflow (4)	Überlauf eines CAN-Sende-Puffers
IRQFailed (8)	Problem mit dem CAN-Interrupt-System
TxError (16)	Fehler beim Versuch, ein CAN-Telegramm zu senden (kein Acknowledge, nichts angeschlossen, Abschlu
BusError (32)	Schwerwiegender CAN-Bus-Fehler (möglicherweise 'error passive', d.h. kein aktives Senden mehr)
Warning (64)	Warnung (genaue Bedeutung hängt vom verwendeten CAN-Controller ab)
BusOff (128)	'BUS-Off'. Dies ist der schwerwiegendste Fehler. Der Controller nimmt weder aktiv (d.h. Senden) noch passiv (Empfang) am CAN-Bus-Verkehr teil. Die Holzhammer-Methode um alle CAN-Controller zurückzusetzen ist das Kommando system.reboot .

Im Gegensatz zu den weiter Unten folgenden CAN-Funktionen steht CAN.status auch in Geräten mit CANopen-Protokoll zur Verfügung.

`CAN.rx_counter( <port> )`

Liefert die Anzahl empfangener CAN-Telegramme, die seit dem Einschalten des Gerätes von diesem CAN-Port **empfangen** wurden.
Im Programmiertool sind darin auch die per CAN-Logfile-Player 'simulierten' CAN-Telegramme enthalten.

`CAN.tx_enable`

Aktiviert oder deaktiviert das periodische *Senden* von CAN-Signalen (durch das MKT-View), die aus einer CAN-Datenbank (*.dbc) importiert wurden, und als 'zu sendende Signale' mit entsprechenden Display-Variablen verbunden wurden. Das gleiche Flag ist als 'signals.tx_enable' auch im *Display-Interpreter* verfügbar. Details in der Dokumentation zum 'Senden von per Datenbank definierten Signalen'.
Hinweis: CAN.tx_enable hat *keinen* Einfluss auf das Script-Kommando can_transmit ! CAN.tx_enable steuert nur den 'Scheduler' für das periodische oder durch die Firmware gesteuerte Senden 'im Hintergrund'.

CAN.tx_counter(<port>)

Liefert die Anzahl der über diesen Port *seit dem Einschalten* erfolgreich **gesendeter** CAN-Telegramme.

CAN.err_counter(<port>)

Liefert die Anzahl von CAN-Fehlern und -Warnungen, die seit der Initialisierung im Interrupt-Handler des CAN-Treibers registriert (gezählt) wurden.
Darin sind auch 'vergleichsweise harmlose' Fehler enthalten, wie z.B. Bit-Stuffing-Fehler enthalten, die im Normalbetrieb eines CAN-Netzwerkes gelegentlich auftreten. Diese Funktion dient daher nur für Diagnosezwecke; die zu erwartende Fehlerrate hängt von der Buslast und vom Umfeld (EMV) ab !

CAN.err_register(<port>)

Liefert den Inhalt eines 'Fehler-Registers' im CAN-Controller, der beim letzten Auftreten eines CAN-Error-Interrupts aus dem entsprechenden Hardware-Register gelesen wurde.
Da das Format des 'CAN-Error-Registers' extrem hardwareabhängig ist, und die Spezifikation des Inhalts den Rahmen dieses Dokumentes sprengen würde, entnehmen Sie die Bedeutung der einzelnen Bits in diesem Register bitte dem Datenblatt des Herstellers (des Controllers):

- Für MKT-View II (mit CPU = LPC2468) : Details in NXP's "UM10237" (LPC24XX User Manual), Chapter 18.8.4, "Interrupt and Capture Register";
- Für MKT-View III (mit CPU = LPC1788) : Details in NXP's "UM10470" (LPC178x/7x User Manual), Chapter 20.7.4, "Interrupt and Capture Register"; in UM10470 Rev. 1.5 auf Seite 514 bis 517 des 1035-seitigen Datenblatts;
- In Geräten mit anderen Controllern und für das Programmiertool (Simulator) hat die Funktion CAN.err_register() keine Funktion, und liefert i.A. den Wert Null !

CAN.err_frame_counter(<port>)

Liefert die Anzahl von "error frames", die seit dem Einschalten auf dem angegebenen CAN-Port registriert wurden.
Strenggenommen ist dies die Anzahl von "bit stuffing - Fehlern", die der CAN-Controller durch Aufruf einer entsprechenden Fehler-Routine (CAN-Interrupt) seit dem Einschalten an die Firmware gemeldet hat. Bei abnorm hoher Fehlerrate könnten auf dem Bus noch mehr Fehler aufgetreten sein, als der Controller (per Interrupt) zählen konnte. Treten weniger als 1000 Error-Frames pro Sekunde auf, ist die Zählung aber hinreichend genau.
Bei CAN besteht ein 'bit-stuffing error' aus mindestens 6 aufeinander folgenden dominanten Bits auf dem Bus (physical layer).
Idealerweise sollten in einem CAN-Netzwerk überhaupt keine Error-Frames auftreten. Mit

diesem Error-Frame-Zähler kann dies überprüft werden. Sporadische Fehler werden oft durch 'Knack-Störungen' hervorgerufen - am Arbeitsplatz des Autors z.B. durch Ein/Auschalten des Lötkolbens(!) oder der Deckenbeleuchtung.

CAN.PulseOut(<port> , <Pulsdauer in Mikrosekunden>)

Erzeugt einen einzelnen Impuls (dominanten Zustand) auf dem spezifizierten CAN-Port, mit der angegebenen Länge (Dauer) in Mikrosekunden.
Diese relativ selten benutzte Funktion eignet sich zum absichtlichen Senden(!) von CAN-Error-Frames (= sechs dominante Bits), was mit einem 'normalen' CAN-Controller nicht möglich ist.
Dies funktioniert nur mit einer geeigneten Hardware (z.B. Geräte mit LPC24xx / ARM-7 oder Cortex-M), aber nicht auf dem PC, und nicht auf Linux-basierten Systemen. Ein Beispiel für die Verwendung dieser Funktion befindet sich in Programm ErrFrame.CVT im Unterverzeichnis 'script_demos'.

Verwenden Sie diese Funktion nicht in einem kritischen Umfeld (fahrendes Fahrzeug) ! Das "absichtliche Senden" von CAN-Error-Frames kann unvorhersehbare Folgen haben !

CAN.timestamp_offset

Diese Variable kann zum 'Verschieben' der Zeitmarke beim Konvertieren von CAN-Messages (Datentyp tCANmsg) verwendet werden. Der *interne Zeitmarken-Generator* startet Einschalten der Spannungsversorgung bei Null.
Beim Aufzeichnen empfangener CAN-Telegramme in einer Datei (per Script) soll die Aufzeichnung i.A. aber beim Zeitpunkt "Null" beginnen. Dies kann z.B. durch folgende Anweisung beim Start der Aufzeichnung beginnen:

```
  CAN.timestamp_offset := system.timestamp; // offset for converting to
Vector ASC format
```

Hinweis: Die vom CAN-Treiber gelieferten Zeitstempel werden von CAN.timestamp_offset *nicht* beeinflusst.
Beeinflusst wird lediglich *die Konvertierung* von tCANmsg in eine Zeichenkette ("Vector ASC") mit der Funktion string().
Die Einheit von CAN.timestamp_offset entspricht system.timestamp (Einheit: Timer-Takte, Frequenz=cTimestampFrequency).

CAN.string_format

Definiert das Format für die Umwandlung einer CAN-Message (Script-Datentyp tCANmsg) in eine Zeichenkette mit der string()-Funktion (oder einem entsprechenden Typecast).
Beispiel:

```
  CAN.string_format := sfVectorASC; // when converting tCANmsg to string,
use "Vector ASC" format
```

CAN.test_id := <CAN-Message-ID>

Sonderbefehl zum Definieren eines speziellen CAN-Message-Identifiers für Hard- und Softwaretests. Bei Geräten mit einer speziell compilierten Firmware (auf Wunsch für das MKT-View IV verfügbar) wird beim Empfang einer CAN-Message mit diesem Identifier bereits im CAN-Interrupt (d.h. mit einer Latenz von wenigen Mikrosekunden) die per CAN.test_action definierbare Aktion durchgeführt (z.B. "Klicken" des Piepers).

CAN.test_action := <N>

Sonderbefehl zum Definieren der 'Aktion', die bereits im CAN-Interrupt beim Empfang einer Message mit dem per CAN.test_id eingestellten Identifier. Momentan (2018-12-18) implementierte Aktionen:

0 (Default) : keine Aktion

Keine 'Sonderbehandlung' eines CAN-Telegramms mit Message-ID = CAN.test_id.

1 : Ansteuern des "Pieper"-Ausgangs per Bit 0 im Datenfeld des Telegramms.

Diese Funktion diente zum Testen des 'Gleichlaufs' von Zeitmarken in Audio- und CAN-Bus-Signalen, und zum Messen der Latenz im Signalpfad des Mikrofons (Audio-Codec mit I2S-Schnittstelle und Hardware-FIFO). Dazu wurde das Mikrofon in der Nähe des Piepers platziert, und die Zeitdifferenz per Y(t)-Diagramm gemessen. Details und Testresultate sind im Script (Quelltext) der Applikation script_demos/diagrams.cvt enthalten.

9.15 4.10.6.12 Spezialfunktionen für LIN

Einige LIN-spezifische Funktionen mussten in der Script-Sprache implementiert werden, weil sich die entsprechende Funktionalität nicht mit der im ersten Teil von Kapitel 4 beschriebenen CAN-API (Application Interface) realisieren liess.
Details zu den folgenden LIN-Bus-spezifischen Funktionen finden sie im verlinkten Dokument.

LIN.SendResponse(tCANmsg ptr pData)

Nur für per Script implementierte LIN-Slaves. Aufruf *nur* aus einem Empfangs-Handler, nachdem der LIN-Frame-Identifier empfangen wurde, auf den der Slave mit dem Senden eines Datenfelds (mit Prüfsumme) reagieren soll. In der als Pointer übergebenen Struktur (vom Typ tCANmsg) ist die Anzahl zu sendender Datenbytes, und das aus 1 bis 8 Bytes bestehende Datenfeld enthalten. Der Frame-Identifier wird ignoriert, denn ein LIN-Slave *kann (darf) keinen Identifier senden* !
Details zur maximal zulässigen Verzögerung zwischen dem Empfang des LIN-Frame-Headers (Master -> Slave) und dem Senden der Response (Slave -> Master) ist der aktuellen LIN-Spezifikation zu entnehmen ("in-frame response space", deren Maximum aber nicht explizit spezifiziert ist, sondern umständlich aus der "maximalen Gesamtdauer des Frames" berechnet werden muss).

10 4.10.7 Steuern der programmierbaren Anzeigeseiten per Script

Prozeduren und Funktionen, die mit dem Schlüsselwort "display" beginnen, dienen zur Steuerung der *programmierbaren Anzeigeseiten* ("UPT-Seiten").

`display.``<var-name>`

Zugriff auf eine Anzeige-Variable ("Display-Variable") per Script. Darin ist <var-name> der Name einer Anzeige-Variablen, die im Programmiertool auf der Registerkarte 'Variablen' definiert wurde, oft mit Informationen zum Transport der Variablen per Netzwerk.

`display.goto_page(` <Nummer oder Name einer UPT-Anzeige-Seite> **`)`**

Schaltet zur angegebenen Display-Seite um.
Die neue Anzeigeseite kann (als Integer-Zahl) über deren Nummer, oder (als String) über deren Namen adressiert werden.

`display.page_name`

Liefert den Namen der *aktuellen* Anzeigeseite (Read-Only).

`display.page_index`

Liefert den *Index* der aktuellen Anzeigeseite (Read-Only).
Zur Beachtung: Wie fast alle 'Indizes' in den meisten Programmiersprachen, beginnt auch hier die Zählung bei *Index Null* (nicht "Eins") !
Zum Umschalten der aktuellen Seite (per Script) dient das Kommando display.goto_page .
Beispiel: Mit der Anweisung `display.goto_page( display.page_index + 1 )` schalten Sie zur *nächsten* Seite um.

`display.num_pages`

Liefert die Anzahl der in der Display-Applikation vorhandenen Anzeigeseiten (read-only).
Im Beispielprogramm 'page menu' wird diese Funktion verwendet, um während der Laufzeit ein Auswahlmenü aufzubauen, in dem die Namen aller in der Applikation existierenden aufgelistet sind.

`display.num_lines`

Liefert die Anzahl der Definitionszeilen auf der aktuellen Anzeigeseite (read-only).
Mit dieser Information kann das Script z.B. eine Schleife aufbauen, in der 'alle Elemente auf der aktuellen Seite' modifiziert (z.B. übersetzt) werden.

`display.page[N].name`

Liefert den Namen der N-ten Anzeigeseite in der Display-Applikation (read-only).
Hinweis: Der gülige Seiten-Index, 'N', läuft von **Null** bis **display.num_pages minus Eins** !

`display.exec( < command string > )`

Mit Hilfe dieses Kommandos kann das Script (fast) jedes beliebige Display-Interpreter-Kommando aufrufen (Link funktioniert nur in HTML, nicht im PDF.). Dieses Kommando sollte *nur mit Vorsicht, und wenn unbedingt nötig* eingesetzt werden ! Es könnte die Ausführung des Scriptes stark 'ausbremsen', weil *Display-Kommandos im Interpreter* interpretiert (!), nicht compiliert werden, und der asynchrone Aufruf mancher Display-Interpreter-Befehle zu

Nebeneffekten führen kann.
Beispiel:

```
display.exec( "bl(0)" ); // turn the display's backlight off via display interpreter
```

Um den direkten Aufruf von Interpreter-Kommandos aus dem Script zu vermeiden, empfiehlt sich der Einsatz von 'Flag-Variablen', die z.B. im Script gesetzt, und in einem globalen oder lokalen Event zum nächstmöglichen Zeitpunkt durch den Display-Interpreter abgefragt wird. Auf diese Weise wird das Script nicht durch den Display-Interpreter 'ausgebremst' (oder für einige Dutzend Millisekunden blockiert). Da viele ehemals nur im Display-Interpreter implementierte Befehle mittlerweile auch (zusätzlich) direkt in der Script-Sprache nutzbar sind, erübrigt sich der Einsatz von display.exec fast immer.

display.pixels_x

Liefert die Breite des LC-Displays, gemessen in *Pixeln* (read-only). Verwendung beispielsweise im QuadBlocks-Demo, um abhängig von der Auflösung des Bildschirms (auch abhängig von "Portrait" / "Landscape") zu einer bestimmten Anzeigeseite zu springen.

display.pixels_y

Liefert die Höhe des LC-Displays, gemessen in *Pixeln* (read-only).

display.rotation

Liefert den Drehwinkel (in Grad) gegenüber der "normalen" Montage des Gerätes.
Mögliche Werte (bei Geräten die sowohl Quer- als auch Hochformat unterstützen):

0 (Grad)
: keine Drehung; das Gerät wird momentan in seiner 'natürlichen' Lage betrieben; z.B. MKT-View mit Kabel am rechten Rand des Gerätes.

90 (Grad)
: Drehung um 90° im Uhrzeigersinn, z.B. MKT-View mit "Kabel nach unten"

-90 (Grad)
: Drehung um 90° gegen den Uhrzeigersinn, z.B. MKT-View mit "Kabel nach oben"

Wird z.B. in das MKT-View III (mit 480 * 272 Pixeln) eine für das Hochformat ('portrait mode') entworfene Applikation geladen, dann wird abhängig vom Geräte-Setup (Display Setup .. Portrait Rotate: CLOCKWISE / COUNTER-CLOCKWISE) der Inhalt des Bildschirms im Uhrzeigersinn (90°) oder gegen den Uhrzeigersinn (-90°) gedreht. Durch Abfragen von 'display.rotation' kann das Script gegebenenfalls auch die Funktion der Cursortasten entsprechend anpassen (die beim Drücken einer Taste gelieferten Tastencodes hängen *nicht* von der oben beschriebenen Drehung ab). Ein Beispiel für die Auswertung von 'display.rotation' für die Auswertung von Cursor-Tasten (oder entsprechenden Schaltflächen mit Pfeil-Symbolen) finden Sie in der Applikation MacPan.

display.fg_color

Liefert die Vordergrundgrundfarbe bzw 'Textfarbe', die für die aktuelle Anzeigeseite als Defaultwert definiert ist.

Im Programmiertool finden Sie die entsprechende Farbdefinition im 'Kopf' einer Seitendefinition (Textfarbe, Vorgabe).

display.bg_color

Liefert die Hintergrundfarbe, die für die aktuelle Anzeigeseite als Defaultwert definiert ist. Im Programmiertool finden Sie die entsprechende Farbdefinition im 'Kopf' einer Seitendefinition (Hintergrundfarbe, Vorgabe).

display.night

Ermöglicht das Lesen und Setzen des 'Tag/Nacht' - Flags (Farbschema). Beispiel:

```
display.night := TRUE;  // Farbschema 'Nacht' verwenden
display.night := FALSE; // Farbschema 'Tag' verwenden
```

display.menu_mode , display.menu_index

Dies sind die Äquivalente der Script-Sprache für die (alten) Display-Interpreter-Funktionen "mm" und "mi" .
Ein Beispiel finden Sie in der Applikation 'DisplayTest.cvt' .
In der Script-Sprache sind die möglichen Zustände von display.menu_mode (mm = **menu modes**) in Form der folgenden Konstanten definiert:

`mmOff` : Weder 'Navigation' noch 'Editieren'

`mmNavigate` : Navigation, d.h. per Cursor wird zwischen verschiedenen Editierfeldern und Menüzeilen auf einer Seite umgeschaltet.

`mmEdit` : Editieren, d.h. der Inhalt eines (i.A. numerischen) Editierfeldes wird mit Hilfe der Cursor-Tasten modifiziert.

display.EditValueMin, display.EditValueMax

Diese Variablen definieren den zulässigen Wertebereich beim Editieren eines numerischen Wertes per 'Up/Down' (Inkrement/Dekrement per Cursortasten oder Drehknopf) in einem Eingabefeld auf der aktuellen Anzeigeseite. Die Funktion wurde im Juni 2016 implementiert, da die Anzeige beim Editieren von *Script-Variablen* (im Gegensatz zu *Display-Variablen*) keine Information über den beim Editieren zulässigen Bereich enthält. Verwenden Sie in Ihrem Script das 'Begin Edit'-Event im OnControlEvent-Handler, um durch Setzen von display.EditValueMin/Max den zulässigen Wertebereich zu definieren.
Sowohl display.EditValueMin als auch display.EditValueMax können per Script jederzeit gelesen oder modifiziert werden. Die Min/Max-Werte werden aber vom System mit den im Programmiertool definierten Parametern (min,max aus der Tabelle mit den Variablendefinitionen) überschrieben, wenn eine normale Display-Variable editiert werden soll. Dies erfolgt *kurz vor* dem Aufruf des OnControlEvent - Handlers mit event=evBeginEdit. Daher kann per Script auch beim Editieren von *Display-Variablen* der zulässige Wertebereich dynamisch begrenzt werden.

display.elem[<Element-Name>].visible

Mit diesem Flag kann das Script ein bestimmtes Anzeigeelement verbergen und wieder sichtbar machen. Beispiele:

```
display.elem["Arrow"].visible := TRUE;  // show the element named "Arrow"
```

```
   display.elem["Popup"].visible := FALSE; // hide the element
named "Popup"
```

Hinweis: Nach dem Laden einer Seite (bei der Seitenumschaltung) sind alle Elemente per Default *sichtbar*.
Wird ein vorher *sichtbares* Element per Script *unsichtbar* gemacht, dann wird beim nächsten Hauptschleifendurchlauf die komplette Anzeigeseite automatisch *komplett* neu gezeichnet, um das Element verschwinden zu lassen.

display.elem[<Element-Name>].xyz oder
display.elem[<Element-Index>].xyz

Entspricht in der *Script-Sprache* der älteren *Interpreter-Funktion* "disp.<Element>.xyz" (mit den gleichnamigen addressierbaren Komponenten '.xyz').
Beispiele (siehe auch 'Display-Test'):

```
// Den NAMEN des aktuell selektierten Elements anzeigen :
print("\r\n Name:", display.elem[ display.menu_index ].na );

display.elem[i].bc := rgb(255,127,127); // Hintergrundfarbe
dieses Elements auf 'Hellrot' setzen
```

Das Anzeige-Element kann (wie oben) durch seinen *Index* adressiert werden, oder (wie unten) über seinen *Namen* :

```
display.elem["BtnNext"].bc := rgb(0,i,255-i); // erste
Hintergrundfarbe ändern
display.elem["BtnNext"].b2 := rgb(0,255-i,i); // zweite
Hintergrundfarbe ändern
```

Im unteren Beispiel ist "BtnNext" der *Name* eines UPT-Anzeige-Elements. Dieser wurde bei der Definition der Anzeigeseite im UPT-Programmiertool eingestellt.
Der Script-Compiler erkennt anhand des Datentyps zwischen den eckigen Klammern ("Index-Klammern"), welche der beiden Adressierungsarten verwendet werden soll: [Integer] = Index, [String] = Name. Im Zusammenhang mit Ereignissen wie evClick, evBeginEdit, evEndEdit können Anzeige-Elemente auch per Index adressiert werden. Der Element-Index wird dazu als Funktionsargument 'param1' an den Event-Handler (OnControlEvent) übergeben.

Hinweis (gilt auch für display.elem_by_id[ ]):

> Wenn auf der aktuellen Anzeigeseite **kein** Element mit dem angegebenem *Namen*, *Index*, oder *Identifier* existiert,
> dann wird der hier beschriebene Lese- oder Schreibzugriff **keinen** Laufzeitfehler verursachen, und das Script wird nicht gestoppt.
> Ein Schreibzugriff auf ein nicht existierendes Element geht 'ins Leere'.
> Ein Lesezugriff auf ein nicht existierendes Element liefert (je nach Datentyp der Komponente) den Wert Null oder eine leere Zeichenkette.

display.elem_by_id[<Control-ID>].xyz

Ähnlich wie display.elem, hier wird das Anzeige-Element allerdings nicht durch seinen Namen, sondern durch seinen (numerischen) Control-ID adressiert. Der Control-ID wird

normalerweise im Zusammenhang mit der Event-Verarbeitung im Script verwendet. Sein Wert wird ebenfalls in der Seiten-Definitions-Tabelle des UPT-Programmiertools eingestellt. Der Zugriff auf ein Anzeige-Element per Control-ID vereinfacht den Einsatz in Event-Handlern, da beim Aufruf des entsprechenden Handlers der (numerische) Wert des Control-IDs direkt als Funktionsargument übergeben wird. Details zu diesem 'Thema für Fortgeschrittene' finden Sie hier .

display.dia.XYZ

Befehle zum Steuern eines Y(t)- oder X/Y-Diagramms auf der aktuellen Anzeigeseite. Details finden Sie in der gesonderten Beschreibung für das Display-Element 'Diagramm'. 'XYZ' steht hier als Platzhalter für das Kommando aus der obigen Beschreibung, z.B. clear, run, ready, ch[N].min, ch[N].max, scale[N].min, scale[N].max, unix_time, horz_zoom_factor, horz_zoom_offset, usw.

display.arr`[row][column][component],`
display.arr.dim`(n_rows, n_columns, n_components),`
display.arr.n_rows, .n_columns, .n_components

Ermöglicht auch aus Scripten den direkten Zugriff auf das alte 'globale' Array des Display-Interpreters, welches weiterhin (aus Gründen der Abwärtskompatibilität) zum Zeichnen von *Polygonen* im oben erwähnten Diagramm (display.dia.XYZ) verwendet wird. Details zum 'globalen' Array des Display-Interpreters (arr[][][]) finden Sie hier.

display.pause := TRUE;

Unterbindet die Aktualisierung der aktuellen "programmierten UPT-Anzeigeseite". Diese Funktion kann als einfache Methode zur Synchronisation zwischen Script und Anzeige verwendet werden: Vor dem Berechnen eines neuen Satzes von Werten für die Anzeige wird die Anzeige pausiert, und (nachdem *alle* Werte fertig im Script berechnet wurden) wird die Anzeige mit dem folgenden Befehl wieder aktiv geschaltet.
Durch das kurzfristige 'Pausieren' der Anzeige wird die Konsistenz (Widerspruchsfreiheit) gewährleist, während das Script (im Hintergrund) z.B. neue Werte für die Anzeige erzeugt. Ohne solche Vorkehrungen könnte, wegen des Pseudo-Multitaskings zwischen Display und Script, ein Teil des Displays noch die 'alten', und ein anderer Teil des Displays bereits die 'neuen' Werte anzeigen.

display.pause := FALSE;

Hebt den Befehl 'display.pause := TRUE' wieder auf, d.h. ermöglicht wieder die 'normale', zyklische Aktualisierung der Anzeige (d.h. der UPT-Bildschirm wird aktualisiert, während das Script im Hintergrund weiterläuft).
Das Testprogramm "LoopTest.cvt" verwendet das Flag 'display.pause' um Flackern des Displays zu vermeiden, während der Puffer für (auf einem Text-Panel sichtbaren) "Textbildschirm" mit neuen Daten gefüllt wird.

display.redraw := TRUE;

Setzt einen Merker (flag) für den *Display-Interpreter*, damit dieser den Inhalt der aktuellen UPT-Anzeigeseite aktualisiert, selbst wenn sich (aus Sicht des Display-Interpreters) "scheinbar" nichts auf der aktuellen UPT-Anzeigeseite geändert hat. Nach erfolgtem Neu-Zeichnen wird dieses Flag automatisch gelöscht.

Im Normalfall ist es *nicht* nötig, das Display durch Setzen des Redraw-Flags per Script zum Neu-Zeichnen zu zwingen, da der *Display-Interpreter* von sich aus die Werte aller mit den Anzeige-Elementen verknüpften (numerischen) Variablen überwacht und -bei Änderung der Variablen- die entsprechenden Elemente automatisch neu zeichnet.

`display.UpdateDiagram`

Kann *nach* dem Berechnen neuer Werte (im Script) aufgerufen werden, um ein eventuell vorhandenes Diagramm zum Neuzeichnen zu veranlassen. Ist ein Diagramm-Kanal wie hier beschrieben mit einen (Script-)Array als "Datenquelle" verbunden, dann werden per 'UpdateDiagram' auch die Daten aus dem (Script-)Array in den Diagramm-eigenen Speicher übernommen ("doppelte Pufferung"). Details und Beispiele zu diesem Befehl finden Sie im Kapitel Kommandos zum Steuern der Diagramm-Anzeige (per Script).

`display.GetVarDefinition(<variable>)`

Spezialfunktion zum Abfragen der *Definition* einer Display-Variablen.
Damit sind *fast* alle Parameter aus der Registerkarte "Variablen" auch zur Laufzeit im Script nutzbar. Für einfache *Display*-Anwendungen wird diese Funktion nicht benötigt.
Eingangsparameter : <variable> = Name der Variablen (als String) oder Definitionsindex (als Integer-Wert).
Der Rückgabewert ist im Erfolgsfall ein Pointer auf eine Struktur vom Typ tDisplayVarDef . Andernfalls (wenn keine Display-Variable mit dem angegebenen Namen oder Definitionsindex existiert) liefert die Funktion einen NULL-Pointer zurück.
Einige Komponenten der Struktur tDisplayVarDef entsprechen den Spalten auf der Registerkarte 'Variablen' im Programmiertool:

.Name

Name der Display-Variablen

.AccessRights

Zugriffsrechte; in bestimmten Fällen wird damit auch die "Übertragungsrichtung" (Senden / Empfangen) definiert.

.BusNr

Bus-Nummer. Bei CAN-Signalen wird dieser Parameter beim Import der DBC-Datei festgelegt.

.CAN_Msg_ID

CAN-Message-Identifier (falls diese Variable mit einem "CAN-Signal" verknüpft ist, und dadurch zu einer 'Netzwerk-Variablen' wird).
Ähnlich wie bei tCANmsg.id ist in Bits 31 und 30 die Bus-Nummer codiert.

.CommChannel

Kommunikationskanal. Entspricht der Spalte "Kanal" in der Variablen-Definitions-Tabelle im Programmiertool.

.CopIndex

CANopen Object Index (1..65535). Entspricht dem ersten Teil der Spalte 'OD-Index' in der Tabelle 'Variablen' im Programmiertool.
Bei einem von Null verschiedenen Eintrag ist die Display-Variable Bestandteil des CANopen-Objektverzeichnisses, und kann z.B. per SDO-Zugriff "von außen" modifiziert werden.

.CopSubindex
CANopen Sub-Index (0..255). Verwendung im Zusammenhang mit dem OD-Index. Details in der Beschreibung des CANopen-Objektverzeichnisses.

.CycleTime_ms
Kann bei CAN-Signalen ("CANdb") je nach Übertragungsrichtung als periodischer Sendezyklus oder Empfangs-Timeout-Intervall dienen.

.MinValue
Minimal zulässiger Wert. Wird gelegentlich als 'Begrenzung' für die Anzeige oder für Editierfelder verwendet.

.MaxValue
Maximal zulässiger Wert. Wird gelegentlich als 'Begrenzung' für die Anzeige oder für Editierfelder verwendet.
In den meisten Fällen sind sowohl 'MinValue' als auch 'MaxValue' Null. Dann findet *keine* Begrenzung statt.

.MuxValue
Wert eines optionalen MULTIPLEXERS aus der CAN-Datenbank ("CANdb", importiert aus *.DBC).

.MuxLSBit
Index des niederwertigen Bits des *Multiplexers* im CAN-Datenfeld (0..63). Details im Dokument über 'CANdb'.

.MuxNumBits
Anzahl Datenbits des *Multiplexers* im CAN-Datenfeld (1..16).

.MuxByteOrder
Byte-Reihenfolge des *Multiplexers* im CAN-Datenfeld, falls dieser mehr als 8 Bit enthält (extrem selten!) .

.RawSigType
Typ bei per CAN ("CANdb") übertragenen Signalen: (U)nsigned, (S)igned, (F)loat, oder (I)nvalid.

.RawSigByteOrder
Byte-Reihenfolge bei per CAN ("CANdb") übertragenen Signalen:(M)otorola, (I)ntel.

.RawSigLSBit
Index des niederwertigen Bits des CAN-Signals im CAN-Datenfeld (0..63). Details im Dokument über 'CANdb'.

.RawSigNumBits
Anzahl Datenbits des per CAN(?) übertragenen Signals .

.ScaleFactor
Skalierungsfaktor. Das 'rohe' CAN-Signal wird für die Umrechnung in die physikalische Einheit mit diesem Fliesskomma-Parameter multipliziert.

.ScaleOffset
Skalierungsoffset. Nach der oben beschriebenen Multiplikation wird noch dieser Offset addiert.

.SDO_Data_Type
Datentyp 'a la CANopen'. Wird z.B. bei der Übertragung per SDO (Service Data Object) benötigt.

.Unit
Zeichenkette mit der physikalischen *Einheit*. Stammt i.A. aus einer importierten DBC-Datei. Maximal 8 Zeichen.

.UpdateTime_ms
Aktualisierungs-Intervall in Millisekunden

.ValueTable
Zeichenkette mit Wertepaaren für die Anzeige "als String". Zur Zeit (2018) noch ohne Funktion.

Bitte beachten: Fast alle der oben aufgeführten Komponenten von **tDisplayVarDef** sind (aus Sicht des Scripts) nur lesbar !
Sollten Sie (für 'sehr spezielle Anwendungen') den *Aufbau der Display-Variablen* während der Laufzeit per Script ändern müssen, setzen Sie sich bitte mit dem Entwickler bei MKT Systemtechnik in Verbindung.
Ein Beispiel für die Verwendung der Funktion display.GetVarDefinition() finden Sie in der Applikation ScriptTest3.cvt:

```
var
   string             sVarName; // name of a display-variable
   tDisplayVarDef ptr pVarDef;  // definition of a display-variable
endvar;

...

sVarName := "FourSines1";        // name of a display-variable

// Show CAN message layout of this *DISPLAY*-variable:
```

```
pVarDef := display.GetVarDefinition( sVarName );
print("\n  ", sVarName, " : bits ", pVarDef.RawSigLSBit, "..",
              pVarDef.RawSigLSBit + pVarDef.RawSigNumBits - 1 );
```

Weitere Details zur Interaktion zwischen *Script* und *Display-Applikation* finden Sie hier:

- Zugriff auf ***Display-Variablen*** per Script
- Zugriff auf ***Script-Variablen*** per Display-Interpreter (in "Display-Events", etc)
- Aufruf von ***Script-Prozeduren*** aus dem Display-Interpreter
- Aufruf von ***Script-Funktionen*** aus Anzeigeseiten (per Backslash-Sequenz im Format-String, z.B. für Internationalisierung)
- Asynchrone Verarbeitung von Ereignissen (Event Handling) per Script (und wie bestimmte Evens im Script *abgefangen* werden können)

4.10.8 'System'-Funktionen

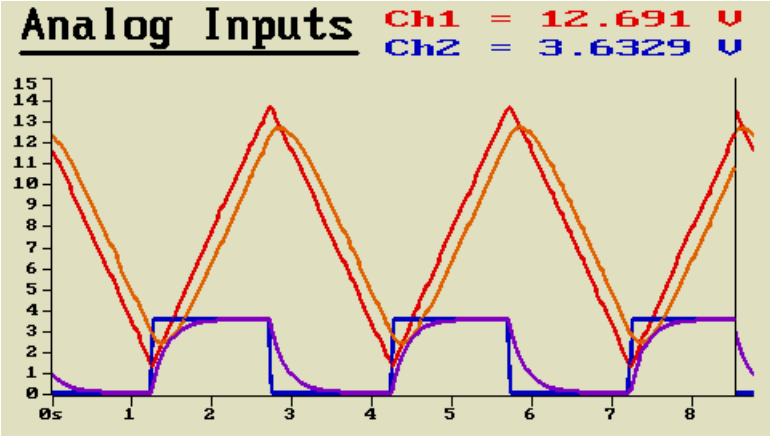
Funktionen und Prozeduren, die mit dem Schlüsselwort "system" beginnen, dienen in der Script-Sprache zum Zugriff auf diverse Systemparameter und 'low-level' Ein-/Ausgabe-Einheiten. Zum Zeitpunkt der Erstellung dieses Kapitels (2017-10-04) waren die folgenden 'System'-Funktionen in der Script-Sprache implementiert :

system.analog_in .audio_ptt .audio_vol .beep .click_vol .dwInputs .dwOutputs .dwFirmware .dwVersion
system.exec .feed_watchdog .led .nv[0..31] .reboot .resources .serial_nr .shutdown .temp .timestamp .ti_ms
system.unix_time .unix_time_boot .vsup .vcap
system.counter_mode .counter_gate_time .counter_frequency .counter_value

`system.analog_in[0] .. system.analog_in[3]`

Mit dieser Funktion kann der aktuelle Wert eines analogen Eingangs abgefragt werden. Im MKT-View III stehen z.B. zwei, im MKT-View IV vier analoge Eingänge 'on-board' zur Verfügung. Details zum Anschluss der analogen Eingange und zur Auflösung des A/D-Wandlers entnehmen Sie bitte dem Datenblatt des jeweiligen Gerätes. Beim MKT-View III sind die beiden Onboard-Analog-Eingänge z.B. an Pins 12 und 13 der 14-poligen Lemo-Buchse verfügbar; als "Analog-Ground" dient dort Pin 14. Beim MKT-View IV stehen zwei weitere analoge Eingänge (system.analog_in[2] und system.analog_in[3]) an der 3-poligen Klemme 'X3' zur Verfügung.
Im Gegensatz zu den alten Display-Interpreter-Funktionen "ain1" / "ain2" liefert die Script-Funktion system.analog_in[] immer einen normalisierten Fliesskomma-Wert zwischen 0.0 und 1.0 (unabhängig vom im Gerät bestückten Spannungsteiler, und unabhängig von der Auflösung des verwendeten A/D-Wandlers). Für *positive und negative* Spannungen geeignete Analogeingänge würden einen Wertebereich von -1.0 bis +1.0 liefern - bei der Erstellung dieser Beschreibung existierten allerdings im MKT-View noch keine derartigen Eingänge. Die gleiche Skalierung (Wertebereich 0 bis 1.0 bzw. -1.0 bis +1.0 bei bipolaren Analogeingängen) wird auch beim Abtasten der Eingänge per DAQ-Einheit verwendet. Auch bei zukünftigen Geräten mit 'besserem' A/D-Wandler, oder bei per digitaler Signalverarbeitung verbesserter Auflösung, wird dieser Wertebereich beibehalten.
Beim MKT-View III / IV mit Originalbestückung entspricht der Wert 1.0 einer Spannung von

15 Volt (=Maximum).
Im Programmiertool / Simulator können die analogen Eingänge wie hier beschrieben simuliert werden.
Ein Beispiel zur Nutzung der analogen Eingänge finden Sie in der Applikation programs/script_demos/AnalogIn.cvt. Darin werden zwei analoge Eingänge mit digitalen Tiefpassfiltern geglättet, und als Y(t)-Diagramm auf dem Display angezeigt:



Screenshot aus Beispielprogramm 'AnalogIn.cvt' auf einem MKT-View III.
Rot: Kanal 1,direkt; orange: Kanal 1,gefiltert; blau: Kanal 2,direkt; violett: Kanal 2,gefiltert

Hinweis: Mit Hilfe der DAQ (Data Acquisition Unit) können die Analogeingänge mit Abtastraten von bis zu 20 kHz abgefragt werden. Durch einfaches Pollen (von system.analog_in[]) sind solche Abtastraten nicht erreichbar.

Ein weiteres Beispiel für die Nutzung der analogen Eingänge (zum Messen von Temperaturen in °C mit einfachen 22-kOhm-NTCs) finden Sie in der Applikation programs/script_demos/Thermo.cvt .

`system.audio_vol`

Ermöglicht das Auslesen oder Setzen der Laustärke des Audio-Ausgangs ("Speaker Volume"). Der gleiche Parameter kann auch im System-Menü des Terminals eingestellt werden. Die Funktion existiert nur bei Geräten mit analogem Audio-Ausgang - siehe feature matrix . Leider sind Dynamikbereich und Skalierung des elektronischen Potentiometers für den Audio-Ausgang hardwareabhängig. Im 'CVT-MIL 320' wurde z.B. ein digitales Potentiometer mit 128 linear (! - nicht logarithmisch - !) geteilten Abgriffen verwendet; in dem Fall ist der einstellbare Wertebereich 0 bis 127. Siehe auch: Audio-Einstellungen und Signalpfade im MKT-View III / IV.

`system.beep(` frequency [,time [,volume [,freq_mod [,ampl_mod]]]])

Erzeugt einen Ton mit dem in manchen Geräten vorhandenen 'Pieper' (Summmer, Piezo, o.Ä.). Der Befehl ersetzt das Kommando 'beep' aus dem alten Display-Interpreter.
Parameter:

frequency : Tonfrequenz in Hertz. Der 0 (Null) schaltet den Ton aus.
time: Dauer des Tons in 100-Millisekunden-Schritten. Fehlt dieser Parameter (und alle

weiteren), oder ist der Wert Null, dann dauert der Ton solange, bis er per Kommando system.beep(0) wieder abgeschaltet wird.
volume: Pseudo-Lautstärke in Prozent (0...100). Da der Pieper i.A. nicht analog, sondern digital (mit einem PWM-Signal) angesteuert wird, werden unterschiedliche Lautstärken durch unterschiedliche Tastverhältnisse realisiert. Dadurch ändert sich mit der Lautstärke leider auch das Oberwellen- Spektrum des so erzeugten Tons. Die maximale Lautstärke (100 Prozent) wird bei einem Tastverhältnis von 1:1 erreicht.
freq_mod: Frequenzmodulation. Damit können z.B. sirenenähnliche Klänge oder "Chirps" (Pfiffe) erzeugt werden werden. Die Einheit ist 'Hertz pro 100 Millisekunden'. Bei positivem Wert steigt die Frequenz an (solange der Ton hörbar ist), bei negativem Wert fällt die Frequenz ab. Die Startfrequenz wird durch den ersten Parameter (frequency) definiert.
ampl_mod: Amplitudenmodulation. Kann verwendet werden, um Tone zu erzeugen, die z.B. mit leise beginnen und dann lauter werden (oder umgekehrt). Nicht besonders effizient (da keine "echte" Amplitudenmodulation möglich ist).
Beispiel: system.beep(150,20,50,100)
Erzeugt einen 2 Sekunden andauernden Ton ("Chirp"), dessen Frequenz von anfangs 150 Hz bis auf 2150 Hz ansteigt (=150 Hz + 2 Sekunden * 100 Hz/0.1sec).
Siehe auch: Audio-Einstellungen und Signalpfade im MKT-View III / IV.

system.play_notes(string)

Ähnelt system.beep, ist aber vielseitiger. Der Befehl verwendet den gleichen 'Pieper' (Summmer, Piezo, o.Ä.) zum Abspielen einer Sequenz von Noten mit unterschiedlicher Dauer und Tonhöhe. Das Format (Syntax des Strings mit 'Noten') entspricht dem alten Interpreter-Kommando "play" (unter diesem Link finden Sie die Spezifikation und Beispiele mit einigen kurzen 'Melodien').
Das Programm arbeitet bereits weiter, während die Melodie noch (im Hintergrund) wiedergegeben wird. Der Befehl verbraucht also nur eine unwesentliche Zeit, und darf deswegen auch in Event-Handlern verwendet werden.

system.click_vol

Nur für Geräte mit Touchscreen. Dient zum Einstellen der "Klick-Lautstärke". Der gleiche Parameter kann auch im System-Menü des Terminals eingestellt werden. Im Gegensatz dazu wird der per system.click_vol eingestellte Wert aber nicht dauerhaft im Konfigurations-EEPROM abgespeichert.
Einheit: Prozent (0..100).
Siehe auch: Audio-Einstellungen und Signalpfade im MKT-View III / IV.

system.backlight

Nur für Geräte mit beleuchtetem Display. Dient zum Einstellen der *normalen Intensität* der Hintergrundbeleuchtung / Tag-Modus, wie im System-Setup unter "Display Setup" / "Brightness".
Wertebereich (wie für die LED-Pulsweitenmodulation): 8 Bit, 0 (aus) ... 255 (max).

system.backlight_low

Nur für Geräte mit beleuchtetem Display. Dient zum Einstellen der *gedimmten Intensität* der Hintergrundbeleuchtung / Nacht-Modus, wie im System-Setup unter "Display Setup" / "Low Brightness".

system.bl_timeout

Nur für Geräte mit beleuchtetem Display. Dient zum Einstellen der *Timeout-Zeit in Sekunden* für die Hintergrundbeleuchtung, wie im System-Setup unter "Display Setup" / "LCD-Off-Time".

system.led(index, pattern, red, green, blue)

Nur für Geräte mit Mehrfarb-LEDs (z.B. MKT-View 2,3). Mit diesem Befehl kann per Script das Blinkmuster, und die RGB-Farbmischung der angegebenen LED gesetzt werden. Wie üblich beginnt die Zählung bei Index Null.
Parameter:

```
  index :  0=erste LED (beim MKT-View 2 unter Funktionstaste F1) ...
2=dritte LED.
  pattern: 8-Bit Blinkmuster. Jedes Bit steuert ein 100-Millisekunden-
Interval.
           Nach einem Zyklus von 8 * 100 ms beginnt dieser von vorne.
           Die Blinkmuster aller derart steuerbaren LEDs sind miteinander
synchronisiert.
  red, green, blue: Definiert die RGB-Farbmischung mit 3 * 8-bit (siehe
Beispiele).
```

Soll ein LED-Parameter (pattern, red, green, blue) *nicht* modifiziert werden, dann kann stattdessem der Wert -1 (bzw. ein negativer Wert) übergeben werden.
Beispiele:

```
  system.led( 0, 0xFF, 0xFF, 0x00, 0x00 ); // 1st LED permanently on, RED
  system.led( 1, 0x0F, 0x00, 0xFF, 0x00 ); // 2nd LED slowly blinking GREEN
  system.led( 2, 0x55, 0x3F, 0x3F, 0xFF ); // 3rd LED rapidly flashing BLUE
  system.led( 2, 0x00, -1,   -1,   -1   ); // 3rd LED off without changing
the colour
  system.led( 2, 0xFF, -1,   -1,   -1   ); // 3rd LED on without changing
the colour
```

Bei Geräten, die statt RGB-LEDs nur eine ROTE und GRÜNE LED enthalten (eventuell kombiniert in einem Gehäuse, z.B. als CANopen-Status-Indikator wie beim HBG-18 und HBG-22), gilt folgende Zuordnung:

- Die 'erste LED' (Index 0) leuchtet GRÜN.
- Die 'zweite LED' (Index 1) leuchtet ROT.
- Beide LEDs (Farbkomponenten) zusammen ergeben ORANGE.
- Da in diesem speziellen Fall beide Farbkomponenten unabhängig angesteuert werden können, kann mit einem komplementären Blinkmuster (0x55 und 0xAA) auch 'zweifarbiges Wechselblinken' realisiert werden.

system.dwInputs

Ermöglicht den Zugriff auf die digitalen Eingänge des Gerätes als 32-bit 'doubleword'. Je nach vorhandener Hardware können bis zu(!) 32 digitale Eingänge ("onboard") mit einem einzigen Zugriff ausgelesen werden (z.B. "UPT 320 I/O", andere Geräte haben keine oder nur sehr wenige digitale I/O-Leitungen "an Bord").
Bit Null (das LSB) enthält den Zustand des ersten digitalen Eingangs, usw.
Beispiel:

```
iDigitalInputs := system.dwInputs; // poll all onboard digital
inputs
if( iDigitalInputs & 0x00000001 ) then // check bit zero =
first input
  print("DigIn1 = high");
else
  print("DigIn1 = low");
endif;
```

Siehe auch: Frequenz- und Ereignis-Zähler für die digitalen Eingänge .

system.dwOutputs

Ermöglicht das Setzen der eventuell vorhandenen digitalen Ausgänge des Gerätes, die hier als ein einzelnes 32-bit 'doubleword' zusammengefasst sind. Je nach vorhandener Hardware können bis zu(!) 32 digitale Ausgänge ("onboard") mit einem einzigen Zugriff gesetzt werden (24 beim "UPT 320 I/O", andere Geräte haben keine oder nur sehr wenige digitale I/O-Leitungen "an Bord").
Bit Null (das LSB) steuert ersten digitalen Ausgang, usw.
Beispiel:

```
system.dwOutputs := system.dwOutputs | 0x0001;    // set the
first onboard-output
system.dwOutputs := system.dwOutputs & (~0x0001); // clear the
first onboard-output
system.dwOutputs := system.dwOutputs EXOR 0x0001; // toggle
the first onboard-output
```

Ein einfaches Beispiel für die Nutzung der digitalen Ausgänge (beim "UPT 320 I/O") finden Sie im 'TrafficLight'-Demo (Ampel).

system.dwFirmware

Liefert die *hardware-spezifische* Firmware-**Artikel-Nummer** als Integer-Wert mit 32 Bit.
Beispiel:

```
print("FW-Art-Nr.=",system.dwFirmware);
```

Ausgabe (wenn das obige Beispiel auf verschiedenen Zielsystemen ausgeführt wird):

```
FW-Art-Nr.=11314       (auf einem MKT-View II mit 'CANdb'-Firmware)
FW-Art-Nr.=11315       (auf einem MKT-View II mit 'CANopen'-Firmware)
FW-Art-Nr.=11392       (auf einem MKT-View III mit 'CANdb'-Firmware)
FW-Art-Nr.=11393       (auf einem MKT-View III mit 'CANopen'-Firmware)
FW-Art-Nr.=11222       (im 'Simulatorbetrieb', auf einem PC laufend)
```

system.dwVersion

Liefert die Firmware-**Versionsnummer** (im Target) als 32-Bit Integer mit folgendem Format:

```
bits 31 .. 24 = major version number (Hauptversionsnummer)
bits 23 .. 16 = minor version number (Nebenversionsnummer)
bits 15 ..  8 = revision number
bits  7 ..  0 = build nummer
```

Beispiel:

```
print("FW-Version=0x"+hex(system.dwVersion,8));
```

-> output: `FW-Version=0x01020304` (if the firmware version was "V1.2.3 - build 4")

system.serial_nr

Liefert die Serialnummer (Seriennummer) des Gerätes als Integer-Wert.
Bei Geräten ohne eindeutige Serialnummer (im EEPROM) liefert diese Funktion den Wert 0 (Null).
Beispiel:

```
print("Serial Number="+itoa(system.serial_nr,5));
```

system.shutdown

Befehl zum 'Selbst-Abschalten' (MKT-View II, III, IV). Entspricht dem *Display-Interpreter-Befehl* sys.poff ("power off").
Wie das Gerät danach wieder eingeschaltet werden kann, hängt von der verwendeten Hardware, und z.T. von den Einstellungen im System-Menü ab. Details dazu finden Sie im Dokument Nr. 85115 (PDF), Kapitel "Power on/off" .
Im Beispiel zum Erkennen von 'Bus-Ruhe' wird dieser Befehl verwendet, um ein MKT-View (II/III/IV) nach 60 Sekunden ohne CAN-Bus-Aktivität automatisch abzuschalten.
Das Script kann per OnSystemShutdown-Handler von einem kurz bevorstehenden 'Shutdown' (Abschalten) informiert werden.
Der 'Grund' für das Abschalten wird dabei als Integer-Wert (iReason) übergeben:

```
      1 = "manuelles" Abschalten (durch den Bediener, per Taste oder
Shutdown-Screen)
      2 = automatsche Abschaltung wegen unzureichender Versorgungsspannung
      3 = automatsche Abschaltung wegen Über- oder Untertemperatur
      0 = Abschaltung aus irgendeinem anderen Grund
```

Ein einfaches Beispiel für einen OnSystemShutdown-Handler finden Sie in programs/script_demos/SystemTest.cvt . Darin wird kurz vor dem Abschalten das Datum, die Uhrzeit, der Grund für das Abschalten, die aktuelle Spannung an den Kondensatoren der USV, und einige weitere Meßwerte in einer Textdatei ("SYSLOG.TXT") abgelegt.

system.nv[0..31]

Zugriff auf einen der 32 'nichtflüchtigen' Integer-Werte, wie mit der Funktion 'nv' im Display-Interpreter. Es bestehen die gleichen Einschränkungen bzgl. der EEPROM- Lebensdauer, wie hier (zum Display-Interpreter) beschrieben. Die Zuweisung eines neuen Wertes hat nicht das

sofortige Speichern im EEPROM zur Folge. Stattdessem wird der Wert zunächst in einem Zwischenspeicher (Latch) abgelegt:

```
system.nv[0] := 12345; // write 32-bit integer into the non-volatile memory latch
```

Um den Inhalt der 32 Zwischenspeicher (latches) dauerhaft im EEPROM zu speichern (nachdem *alle* zu speichernden Werte geändert wurden), rufen Sie das folgende Kommando auf:

```
system.nv_save; // write all modified nv locations into the EEPROM
```

Um *Fliesskommawerte* im EEPROM zu speichern, konvertieren Sie diese vor der Zuweisung an system.nv[] per FloatToBinary() formal in einen Integer-Wert. Im Gegensatz zur automatischen Typumwandlung (von 'float' nach 'int') bleiben dabei alle Bits (Vorzeichen, Mantisse, Exponent, Vorzeichen des Exponenten) erhalten. Beim Auslesen wandeln Sie den Wert dann per BinaryToFloat() wieder zurück in das Script-kompatible Fliesskommaformat.

Siehe auch: Default-Einstellungen für die Werte im nv[]-Array,
Vermeiden des Überschreibens von nv[]-Werten beim Laden einer neuen Applikation.

system.resources

Diese Funktion dient nur zur Fehlersuche während der Entwicklung. Der Rückgabewert ist ein kombinierter Indikator für die 'noch verbleibenden System-Resourcen', gemessen in **Prozent**.
Der Wert ist das *Minimum* aus den folgenden, für die korrekte Abarbeitung des Scripts wichtigen System-Parameter:

- Verbleibender freier Platz auf dem Stack (für lokale Variablen und Berechnungen)
- Verbleibender freier Platz im dynamisch allozierten Speicher ("heap")

Fallen die verbleibenden System-Resourcen während des Betriebs unter 10 Prozent, kann davon ausgegangen werden daß im Script noch ein Fehler steckt; z.B. illegale Rekursion, Allozierung von zu vielen Strings oder Arrays, usw.
Die Ursache für einen Mangel an Resourcen kann z.Z. nur im Debugger/Simulator, d.h. im Programmiertool (unter memory usage display) untersucht werden. Die Funktion 'system.resources' funktioniert allerdings auch im Zielsystem (d.h. in der Mikrocontroller-Firmware), und kann ggf. zum Anzeigen einer Warnmeldung, oder zum Absetzen einer Alarm-Meldung per CAN verwendet werden.

system.temp

Liefert die aktuelle Temperatur ***im Inneren des Gerätes***, gemessen in der Nähe des TFT-Displays. Sie können diesen Wert zu Testzwecken in Ihrer Anwendung auf dem Display anzeigen, zum Loggen in eine entsprechend deklarierte Variable umkopieren, etc.
Im Gegensatz zur älteren Display-Interpreter-Funktion 'sys.temp' liefert die Script-Funktion 'system.temp' einen Fliesskomma-Wert, gemessen in **°C (Grad Celsius)**.
Hinweis: Die gleiche Temperatur wird auch für das 'automatische Abschalten bei zu hoher

Temperatur' verwendet, wie im gleichnamigen Kapitel im Dokument Nr. 85115 beschrieben.

system.timestamp

Liefert den aktuellen Zählerstand des Zeitmarken-Generators (Zählregister eines Hardware-Timers) als Integer-Wert, gemessen in *Timer-Ticks* (vgl. system.timestamp_sec) . Der gleiche Generator liefert u.A. auch die Zeitmarken für den CAN-Treiber. Daher kann durch Vergleich mit dem Zeitstempel in einem empfangenen CAN-Telegramm ermittelt werden, wie viele "Timer-Takte" seit dem Empfang des Telegramms vergangen sind. Um die Differenz in eine Angabe von Sekunden (oder ggf. Millisekunden) umzurechnen, kann die Differenz durch die Konstante cTimestampFrequency dividiert werden (Details dazu später).
In Kombination mit dem Kommando 'wait_ms()' ("Warte für eine bestimmte Anzahl von Millisekunden") kann das Ergebnis auch zur Synchronisation des Scripts mit einem externen Prozess verwendet werden.
Beispiel: Senden einer zeitlich präzise abgepassten 'Antwort' für ein CAN-Telegramm:

```
display.pause := TRUE; // don't let the display-interpreter
interfere now
Tdiff := can_rx_msg.tim - system.timestamp; // timestamp
difference between 'now'
                                            // and a certain
CAN message reception
Tdiff := (1000*Tdiff)/cTimestampFrequency; // convert to
milliseconds
wait_ms(100-Tdiff);    // wait until 100 ms have passed since
CAN msg reception
can_transmit;
display.pause := FALSE; // resume normal display operation
```

Hinweis: Dieses Beispiel ist nicht zu 100 Prozent 'bullet-proof'. Es funktioniert nur, wenn der oben gezeigte Code spätestens 100 Millisekunden nach dem Empfang der zu beantwortenden CAN-Message aufgerufen wird. In seltenen Fällen (gebremster Task-Wechsel durch Display-Update bei sehr komplexer Grafik; oder Script wartet bereits an anderer Stelle) könnte dies fehlschlagen. Um das Risiko einer 'verspäteten' Antwort zu minimieren, kann das Aktualisieren der Anzeige vorübergehend unterbunden werden. Dann (und nur dann) steht die gesamte CPU-Zeit für das Abarbeiten des Scripts zur Verfügung, mit Reaktionszeiten im Bereich weniger Millisekunden.
Trotzdem ist die im Anzeigeterminal integrierte Script-Sprache kein Ersatz für einen 'hart echtzeitfähige' PLC (SPS), obwohl die meisten Terminals über einen schnellen, präemptiven Multitasking-Kernel verfügen.

Die 32-Bit-Zeitstempel werden von einem Hardware-Timer/Counter erzeugt, der beim Einschalten mit dem Zählerstand Null startet. Danach zählt er mit einer Taktfrequenz von cTimestampFrequency aufwärts. Diese Taktfrequenz ist hardwareabhängig; sie liegt i.A. zwischen 30 und 60 kHz. Bei 40 kHz springt läuft der 32-Bit-Wert nach 2^31 / 40000 Hz = 53687 Sekunden, d.h. circa 14 Stunden, von 0x7FFFFFFF (=größte darstellbare POSITIVE Zahl) auf 0x80000000 (=negativste darstellbare Zahl) über.
Trotz Überlauf können (bei der Verwendung von *vorzeichenbehafteter 32-Bit-Integer-Arithmetik*) aber problemlos Differenzen berechnet werden - selbst wenn der Timer 'zwischendurch' von 0x7FFFFFFF auf 0x80000000 oder von 0xFFFFFFFF auf 0x00000000 übergelaufen ist. Aus dem Grund dürfen Zeitmarken (system.timestamp, can_rx_msg.tim, etc) ***nicht*** erst in Sekunden (oder irgendeine andere Einheit) umgerechnet werden, bevor die

Differenz gebildet wird ! Berechnen Sie ***erst*** die Differenz zwischen zwei Zeitmarken (wie im obigen Beispiel):

```
Tdiff := can_rx_msg.tim - system.timestamp;
```

und rechnen ***danach*** die Differenz in die gewünschte Einheit um - hier Millisekunden:

```
Tdiff := (1000*Tdiff) / cTimestampFrequency; .
```

Siehe auch: Programmierbare Timer, Timer-Events, CAN.timestamp_offset, StartStopwatch(), ReadStopwatch_ms(), Konvertieren von Datum und Uhrzeit .

system.timestamp_sec

Liefert den aktuellen Stand des Zeitmarken-Generators als Fliesskomma-Wert (float oder double), skaliert in *Sekunden* (dieselbe 'Quelle' wie system.timestamp, Details dort).
Im Gegensatz zu system.unix_time startet der Zeitmarken-Generator beim *Einschalten des Gerätes* bei Null, und bietet daher (bei der Speicherung als 32-Bit-Fliesskomma-Wert mit 24-Bit-Mantisse) eine bessere Auflösung als die Unix-Zeit:
Nach 24 Stunden Laufzeit muss die 24-Bit-Mantisse den Wert 24*60*60 Sekunden speichern, ein 2^24-stel davon entspricht ungefähr 5 Millisekunden. Ist eine bessere Auflösung erforderlich, dann muss das Ergebnis als double (mit 64 Bit) gespeichert werden.
Die von system.timestamp_sec gelieferte Wert ("Anzahl Sekunden seit Power-On") wird u.A. auch beim Zugriff auf die DAQ-Einheit, und zur Synchronisierung der Anzeige von Y(t)-Diagrammen verwendet.

system.ti_ms

Liefert den aktuellen Stand des Zeitmarken-Generators (Hardware-Timer), skaliert in Millisekunden.
Intern wird der Wert aus der gleichen Quelle wie system.timestamp abgeleitet. Bei der im MKT-View (II,III,IV) verwendeten Timer-Taktfrequenz von 40 kHz läuft der 32-Bit-Timer nach ungefähr 2^32/(40kHz * 60 * 60) = 29.8 Stunden über, was einen "Sprung" im Wert von 'ti_ms' verursacht. Nach einem Überlauf oder nach Power-On startet dieser Timer wieder bei Null (er wird nicht von der batteriegepufferten Uhr beeinflusst).
Im *Display-Interpreter* steht mit ti_ms eine äquivalente Funktion zur Verfügung.

system.unix_time

Falls das Gerät mit einer batteriegepufferten Echtzeituhr (RTC) ausgestattet ist, kann mit dieser Funktion das aktuelle Datum und Uhrzeit als 'Unix-Zeit' ausgelesen werden.
Die 'Unix-Zeit' (auch 'Unix-Sekunde' genannt) ist wie folgt definiert:

Die Anzahl von Sekunden, die seit dem 1. Januar 1970 um Mitternacht vergangen sind

(1970-01-01 00:00:00 UTC) .

In diesem Zusammenhang: Vorsicht vor dem "Unix Millenium Bug" (Year 2038 Bug), von dem alle Systeme betroffen sein werden, die (wie viele heutige Geräte) eine vorzeichenbehaftete 32-Bit-Integer-Zahl zur Speicherung der 'Unix-Zeit' verwenden ! Seit November 2018 liefert die Script-Funktion system.unix_time das Ergebnis nicht mehr als int, sondern als double mit Nachkomma-Anteil.
Details finden Sie im Beispiel (bzw. Testprogramm) TimeTest.cvt .

Das 'System' (programmierbares Terminal) kümmert sich bezüglich der Uhrzeit nicht um Zeitzonen. Wir empfehlen, die eingebaute Uhr immer in UTC (ehemals "Greenwich Mean Time") laufen zu lassen, und ggf für die *ANZEIGE* eine zur Zeitzone passenden Offset zu addieren. Nur dann kann die Funktion 'system.unix_time' (wie auch system.unix_time_boot, s.U.) wie beabsichtigt eine Zeit in UTC zurückliefern, wie es der Definition der "Unix-Zeit" entspricht.
Siehe auch: time.unix_to_str, Funktionen zum Konvertieren von Datum und Uhrzeit, Modifiziertes Julianisches Datum (MJD),
Zeitstempel im Array-Header, Zeitstempel für Y(t)-Diagramme.

system.unix_time_boot

Liefert Datum und Uhrzeit (kombiniert als 'Unix-Zeit'), an dem das System eingeschaltet wurde, und an dem der interne hochauflösende Zeitmarken-Generator (system.timestamp) mit dem Zählerstand "Null" startete.

system.vsup

Liefert die aktuelle Versorgungsspannung ***des Terminals***, gemessen "kurz hinter der Verpolschutzdiode". Der Wert ist daher nicht sehr präzise. Sie können diese Funktion zu Diagnosezwecken einsetzen, wenn das Terminal z.B. aus dem Bordnetz eines KFZ versorgt wird.
Im Gegensatz zur älteren Display-Interpreter-Funktion 'sys.vsup' liefert die Script-Funktion 'system.vsup' einen Fliesskomma-Wert, gemessen in **Volt**.

system.vcap

Ähnlich wie system.vsup, hier wird allerdings die Spannung an den Pufferkondensatoren in der internen USV (Unterbrechungsfreie Stromversorgung) gemessen. Einheit: Volt. Das Maximum liegt (beim MKT-View II,III,IV) bei knapp unter 5 Volt.
Bei Geräten ohne interne USV liefert diese Funktion den Wert Null.

system.audio_ptt

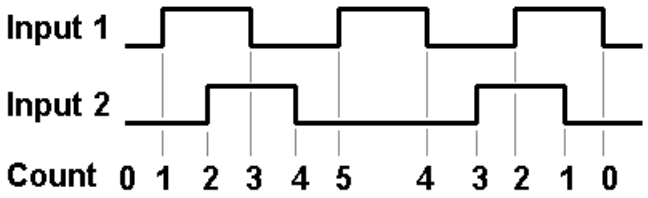
Aktiviert ein Schaltrelais für die 'Push-To-Talk'-Funktion des Audio-Ausgangs. Nur bei wenigen Geräten vorhanden.

system.counter_mode

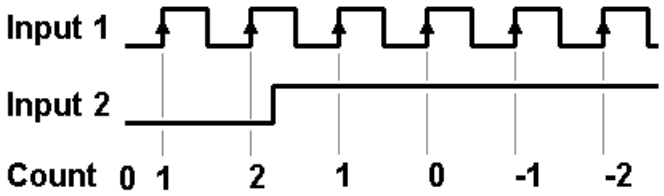
Betriebsart des bei *manchen* (*) Geräten implementierten Frequenz- oder Ereigniszählers. Die folgenden Betriebsarten waren zum Zeitpunkt der Erstellung dieser Beschreibung (2017-10-11) implementiert:

- 0 : Zähler aus (passiv). Dies ist der Default-Zustand nach dem Einschalten.
- 1 : Einfacher Zähler für steigende Flanken an Digitaleingang 1
- 2 : Einfacher Zähler für steigende Flanken an Digitaleingang 2
- 3 : Zwei unabhängige Frequenz- oder Ereigniszähler an Digitaleingängen 1 und 2
- 4 : Komplexer Zähler mit Quadratur-Eingang, auch für Inkrementalgeber oder 'Encoder' geeignet.
  Anhand der Phasenlage zwischen Eingang 1 und 2 wird die Drehrichtung erkannt.
  Der Zählerstand wird bei *jeder* Flanke (steigend oder fallend) an *jedem* Eingang

inkrementiert oder dekrementiert.
Dies gilt auch für die gemessene Frequenz, die dadurch (z.B. bei "Linksdrehung") *negativ* werden kann.



- 5 : Vorwärts/Rückwärts-Zähler mit Pulseingang (steigende Flanke) an Digitaleingang 1, Richtungssteuerung an Digitaleingang 2 (LOW=Inkrementieren, HIGH=Dekrementieren).



`system.counter_gate_time`
Torzeit für *Frequenzmessungen* mit dem oben beschriebenen 'digitalen Zähler'.
Einheit: ***Millisekunden***. Default: 1000 Millisekunden.

`system.counter_frequency[0]`
Mit dem oben beschriebenen Zähler gemessene *Frequenz in Hertz* am ersten Eingang (Array-Index Null). Bedingt durch das Messprinzip kann das Ergebnis auch bei Torzeiten unter 1 Sekunde eine Auflösung von weniger als einem Hertz haben, daher Ergebnis als Fliesskommazahl (float) !

`system.counter_frequency[1]`
Aktuelle Frequenz am **zweiten** Eingang. Nur Verfügbar in Betriebsart 3 ('Zwei unabhängige Frequenz- oder Ereigniszähler').

`system.counter_value[0]`
Aktueller Wert des Impuls- oder Ereigniszählers für den ersten Eingang (Array-Index Null). Im Gegensatz zur Frequenz immer unabhängig von der Torzeit.
Schreib- und lesbar. In einigen Betriebsarten (z.B. Vorwärts/Rückwärtszähler) kann das Ergebnis negativ sein.

`system.counter_value[1]`
Aktueller Wert des Impuls- oder Ereigniszählers für den **zweiten** Eingang. Nur Verfügbar in Betriebsart 3 ('Zwei unabhängige Frequenz- oder Ereigniszähler').

(*) Der Frequenz / Ereigniszähler wurde im Oktober 2017 zunächst im MKT-View III / IV implementiert.
In älteren Geräten (z.B. MKT-View II) ist er nicht verfügbar.
Angaben zur Eingangsspannung und zur *maximal* messbaren Frequenz entnehmen Sie bitte dem Datenblatt.
Typische Werte sind: Vin_low <= 3.0 V, Vin_high >= 6.5 V, f_max = 20 kHz (mit modifiziertem Eingangstiefpaß).
Auf Wunsch können die Digitaleingänge so modifiziert werden, daß für TTL-Pegel(Schwellwert ca. 3 V)
und für höhere Frequenzen geeignet sind (MKT-View IV ohne Tiefpass: bis zu 200 kHz).
Bitte beachten: Der Frequenzzähler ist eine der 'erweiterten', freizuschaltenden Script-Funktionen.
Ein Beispiel zum Messen von Frequenzen an den digitalen Eingängen finden Sie in der Applikation programs/script_demos/DigitalInputFrequency.cvt .

`system.feed_watchdog( < number of milliseconds > )`

Dieses Kommando sollte nur im *äußersten Notfall* verwendet werden, wenn z.B. ein Event-Handler, oder der Aufruf einer Script-Funktion per Backslash-Sequenz im Format-String einer Anzeigezeile, ***länger als 400 Millisekunden für die Ausführung benötigen könnte***.
Dies ist glücklicherweise nur sehr selten nötig; eine einfache Zuweisung an ein 'Signal' (welches dann in der Script-Hauptschleife abgefragt werden kann, wo dann die eigentliche Verarbeitung erfolgt) benötigt z.B. deutlich unter einer Millisekunde.
Im weiteren Verlauf sind mit 'Event-Handler' auch die z.B. per Backslash-Sequenz aus dem Display-Interpreter aufgerufenen Funktionen gemeint (siehe z.B. GetText-Beispiel).
Es folgen technische Details, die Sie (als Entwickler) hoffentlich nicht benötigen, da sich in den meisten Fällen langwierige Berechnungen und ähnliches aus den Event-Handlern in die Hauptschleife des Scripts verlagern lassen (in der keine Zeitbeschränkung für Unterprogramme besteht).
Endlosschleifen sind in Event-Handlern (u.Ä., s.O.) tabu, weil das Gerät dann nicht mehr bedienbar, bzw. aus Sicht des Bedieners 'abgestürzt' wäre. Das Laufzeitsystem terminiert die aufgerufene Funktion daher (per 'watchdog'), wenn diese nicht schnell genug 'freiwillig' wieder zum Aufrufer zurückkehrt ! Im normalen Script-Kontext besteht diese Einschränkung nicht, denn dort sorgt das Laufzeitsystem dafür, dass selbst bei einer Endlosschleife in Script (ohne "wait") das Gerät bedienbar bleibt.
Sollte die im Kapitel 'Event-Handling' spezifizierte Zeit für den Event-Handler nicht ausreichen, kann der Watchdog *im äußersten Notfall* z.B. mit dem Befehl

```
system.feed_watchdog(500); // Script-Watchdog für 500 Millisekunden "füttern"
```

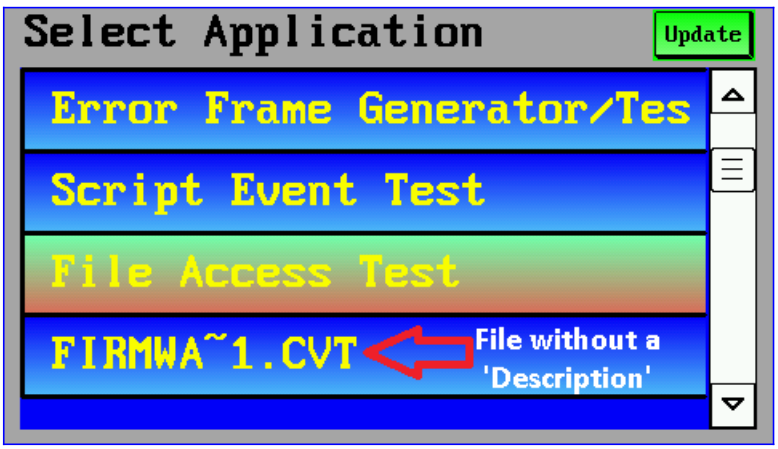
auch im Script nachgeladen werden, - was aber die bereits erwähnten Konsequenzen auf die Bedienbarkeit des Gerätes haben kann, wenn diese Verweildauer im Event-Handler wirklich ausgenutzt wird.

`system.reboot`

Booted das komplette System (Anzeige-Terminal) neu, inkl. Hardware-Reset. Damit können auch hartnäckige Fehler 'behoben' werden, z.B. ein in den Bus-Off-Zustand geschalteter CAN-Controller.
Dieser Befehl eignet sich auch, um aus der per 'App-Selektor' gestarteten Anwendung wieder in den 'App-Selektor' zurückgeschaltet werden.
Vor dem Aufruf dieses Befehlt sollte Ihr Script alle offenen Dateien schließen.
Andernfalls könnte das System beim Neustart die folgende Warnung anzeigen:
"System has not been shut down properly - this may damage the memory card".

system.exec(filename)

Lädt die Datei mit dem angegebenen Dateinamen (*.cvt oder *.upt) von der Speicherkarte in den Hauptspeicher, compiliert das darin enthaltene Script neu, und startet die neue Applikation. Die im geräteinternen FLASH-Speicher abgelegte Applikation (z.B. der 'App-Selektor') bleibt dabei erhalten, so dass mit dem Kommando
system.reboot
nach getaner Arbeit wieder in das im Flash gespeicherte Programm (z.B. den "App-Selektor", s.U.) umgeschaltet werden kann.
Im Simulatorbetrieb (im Programmiertool) ist der system.exec() nur dann aktiv, wenn das laufende Programm seit der letzten Änderung *abgespeichert* wurde.
Das Nachladen, Kompileren und Starten eines Programms dauert beim MKT-View III (Cortex-M3 mit 96 MHz) circa 2 bis 4 Sekunden, beim MKT-View IV (Cortex-M4F mit 200 MHz) i.a. unter 2 Sekunden.
Ein Beispiel für den Einsatz des Kommandos system.exec finden Sie im 'App-Selektor'.



Screenshot aus dem 'Applikations-Selektor'

Hinweis: Die Funktion system.exec existiert nur in Geräten mit "großem" RAM, z.B. MKT-View III / IV, aber *nicht* in älteren Geräten wie z.B. MKT-View I / II !

Siehe auch (hilfreich für 'nachgeladene' Scripte): Mit dem Attribut 'noinit' deklarierte Variablen.

system.dwKeyMatrix

Liefert den aktuellen Zustand der Tastaturmatrix als 32-bit 'doubleword'. Jeder Taste wird durch ein Bit repräsentiert; dadurch sind auch *Kombinationen* von Tasten möglich, für die

kein entsprechender Tastencode definiert ist (vgl. getkey).
Die von system.dwKeyMatrix gelieferte Bitkombination entspricht weitgehend der Display-Interpreter-Funktion km und (bei Geräten mit CANopen V4) Objekt 0x5001:

- Bit 0..7 : Funktionstasten F1 (Bit 0) bis F8 (Bit 7) .
- Bit 8..15 : Cursortasten Links (Bit 8), Rechts(9), Up(10), Down(11);
  ENTER (Bit 12), ESCAPE (Bit 13), erste Shift-Taste (Bit 14), zweite Shift-Taste (Bit 15) .
- Bit 16..23: Numerische Tasten '0' (Bit 16) .. '7' (Bit 23) .
- Bit 24..31: Numerische Tasten '8' (Bit 24) .. '9' (Bit 25),
  "Punkt" (Bit 26), "Plus" bzw. "Plus/Minus" (Bit 27),
  Tabulator-Taste oder 'zwei horizontale Pfeile' (Bit 28),
  Fragezeichen, Help, o.Ä. (Bit 29),
  "A bis Z", "Alpha", o.Ä. Umschaltung numerische / alphanumerische Eingabe : Bit 30,
  "Backspace" oder "spezieller Pfeil nach Links" : Bit 31 .

Zur besseren Lesbarkeit sollten bei der Abfrage der Tastaturmatrix (auch im Event-Handler, s.U.) ausschliesslich die folgenden symbolischen Konstanten verwendet werden:

```
  kmF1 .. kmF8 : Funktionstasten F1 bis F8
  kmLeft    :  Cursortasten..
  kmRight
  kmUp
  kmDown
  kmEnter   : ENTER-Taste
  kmEscape  : ESCAPE-Taste
  kmShift1  : erste Shift-Taste ("Umschalttaste"?)
  kmShift2  : zweite Shift-Taste ("Umschalttaste"?)
  kmDigit0 .. kmDigit9 : Dezimale Zifferntasten (nur bei manchen Geräten
vorhanden)
  kmDot     : decimal separator (dot, sometimes comma)
  kmPlus    : "PLUS" (sometimes labelled "+/-" or "* +")
  kmTab     : TAB key, or "two horizontal arrows"
  kmMode    : Questionmark, HELP, HELP/MODE, or similar labelled key
  kmAtoZ    : key labelled "ABC"/"A"/"Alph" (for TEXT INPUT)
  kmBackspace: Backspace,"special arrow pointing left", or similar labelled
key
```

Um schnell auf Änderungen der Tastaturmatrix zu reagieren (ohne Polling), implementieren Sie einen OnKeyMatrixChange-Handler im Script.
Ein Beispiel finden Sie in der Applikation script_demos/KeyMatrix.cvt .

getkey

Liest den nächsten Tastencode aus dem Tastaturpuffer des Gerätes. Der gleiche Puffer wird auch von der (alten) Interpreterfunktion 'kc' verwendet, was zur Folge hat, daß per 'getkey' aus dem Puffer entfernte Tastencodes auch von der (eventuell kurze Zeit später aufgerufenen) Funktion 'kc' nicht mehr registriert werden können, und umgekehrt !
War der Tastaturpuffer beim Aufruf der Funktion nicht leer, dann liefert getkey einen von Null verschiedenen Wert. Dieser Wert wird vorzugsweise in einer select-case-Liste mit den *key*-Konstanten verglichen, um z.B. beim Betätigen einer Taste (unabhängig von der aktuellen Display-Seite) eine bestimmte Aktion auszulösen.

Bedenken Sie in diesem Zusammenhang, daß einige Geräte nur "wenige", bzw im Extremfall *keine* "echten" Tasten enthalten.
Einige MKT-Geräte enthalten nur Funktionstasten (Symbole keyF1 bis keyF3),
andere enthalten nur vier Cursortasten (keyUp, keyDown, keyLeft, keyRight),
oder nur 'Enter' (keyEnter) und 'Escape' (keyEscape).
Beispiele zur Tastaturabfrage per 'getkey' finden Sie in den Beispielprogrammen TScreenTest.cvt und 'QuadBlocks' .
Erweiterte Abfragemöglichkeiten, z.B. um zu erkennen ob und wann eine bestimmte Taste *gedrückt* und wieder *losgelassen* wurde, verwenden Sie statt getkey die 'Low-Level'-Event-Handler OnKeyDown und OnKeyUp.

Siehe auch : display (Funktionen), Schlüsselwörter, Übersicht .

11 4.10.9 Funktionen zum Konvertieren von Datum und Uhrzeit

Die folgenden Prozeduren / Funktionen dienen in der Script-Sprache zum Konvertieren verschiedener Datums- und Uhrzeit-Formate. Dazu gehören auch wichtige, nichttriviale "Kalender-Funktionen" .

`time.unix_to_str(string format, int unix_date_and_time)`

Diese *Function* wandelt ein Datum (genauer: ein Datum mit Uhrzeit im Unix-Format) in eine Zeichenkette um. Das Format wird durch den Format-String (erstes Argument) definiert, z.B.:
"YYYY-MM-DD hh:mm:ss" : erzeugt ein ISO 8601-konformes Format mit Datum *und* Uhrzeit.
(ohne das häßliche 'T', welches in ISO 8601 als Trenner zwischen Datum und Uhrzeit empfohlen wird)
"YYYY-MM-DDThh:mm:ss" : dies wäre das von ISO 8601 favorisierte Format, für Bildschirmanzeigen aber zu häßlich.
"DD.MM.YYYY" : erzeugt ein 'unlogisches' Datumsformat, welches leider in Deutschland weit verbreitet ist.
"MM-DD-YYYY" : ebenfalls ein 'unlogisches' aber weit verbreitetes Datumsformat.
"MMM-DD-YYYY" : ähnlich aber eindeutiger: Drei BUCHSTABEN (keine Ziffern) für den Monat.
(MMM, drei Großbuchstaben, stehen für JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV oder DEC)
"Mmm DD, YYYY" : ähnelt dem traditionellen amerikanischem Format, aber nur 3 Buchstaben für den Monat.
(Mmm stehen als Platzhalter für Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)

Der Autor empfiehlt die Verwendung der ISO-8601-ähnlichen Formate; speziell für deutsche Firmen mit Partnern in Übersee.
Wie würden Sie, und wie würde Ihr Kunde das Datum 3/12/2001 interpretieren ? Zwölfter März oder dritter Dezember ?

Beispiel:

```
    print("Es ist jetzt ", time.unix_to_str("YYYY-MM-DD
hh:mm:ss",system.unix_time) );
```

time.date_to_mjd(in int year, in int month, in int day)

Diese *Funktion* kombiniert ein aus Jahr (1858..2113), Monat (1..12), und Tag-des-Monats (1..31) bestehendes Kalenderdatum in das modifizierte Julianische Datum (MJD, Definition s.U.). Das Ergebnis ("Return-Wert") ist das MJD.

time.mjd_to_date(in int mjd, out int year, out int month, out int day)

Diese *Prozedur* konvertiert das modifizierte Julianische Datum (MJD) in das übliche Gregorianische Datum, d.h. "zerlegt" die MJD-Tageszahl (s.U.) in Jahr (1858..2113), Monat (1..12), und Tag des Monats (1..31).

Das Modifizierte Julianische Datum (MJD) wird folgendermaßen definiert:

***die Anzahl von Tagen seit Mitternacht am 17. November 1858* (im ISO8601-Format: 1858-11-17 00:00:00) .**

Dieses Format wird oft verwendet, wenn die Differenz von zwei Kalenderdaten berechnet werden soll.
Das MJD-Format kann ohne großen Aufwand auch in die entsprechende 'Unix'-Zeit umgewandelt werden, denn das 'UNIX-Geburtsdatum' (1970-01-01 00:00:00) ist als MJD Tag-Nummer 40587.0 .
Im Gegensatz zum MJD zählt die 'Unix-Zeit' die Anzahl von Sekunden (beginnend bei Sekunde Null am Unix-Geburtsdatum), daher ist die Konvertierung trivial.
Ein Beispiel zur Verwendung der genannten Funktionen finden Sie in der UPT-Applikation programs/script_demos/TimeTest.cvt .

Siehe auch (im Zusammenhang mit Zeitmessungen per Script):
Programmierbare Timer, Timer-Events, CAN.timestamp_offset, StartStopwatch(), ReadStopwatch_ms(), GPS .

12 4.10.10 Funktionen zum Zugriff auf den GPS-Empfänger

Im *Script* stehen ähnliche Funktionen wie im *Display-Interpreter* zur Verfügung, um den GPS-Empfänger zu steuern und auf aktuelle Positionsdaten zuzugreifen. Der Zugriff auf den GPS-Empfänger wird im Dokument 'gpsdec_49.htm' genauer erläutert. In beiden Fällen wird der Modul-Präfix 'gps.' verwendet, und die Syntax ist weitgehend identisch.

Hinweis

In der Script-Sprache steht der Datentyp 'double' für Fliesskommawerte mit 'doppelter Präzision' (64 Bit) zur Verfügung.
Der 32-Bit-Standard-Datentyp 'float' mit 23-Bit-Mantisse reicht zum Speichern von Breiten- und Längengrad *in Grad* mit einer Auflösung von wenigen Metern nicht aus. Ähnliches gilt für das kombinierte GPS-Datum im Unix-Format (Anzahl Sekunden seit 1970-01-01 00:00:00.0 UTC). Verwenden Sie zum Speichern oder für Berechnungen im Script im Zusammenhang mit den Funktionen gps.lat_d, gps.lon_d, und gps.unix_time daher besser den Datentyp *double* statt *float*.

Wenn der OnLocationChanged-Handler im Script definiert ist, wird dieser bei jeder neuen Positionsmeldung aufgerufen, d.h. je nach Empfänger 1 oder 4 mal pro Sekunde.

13 4.10.11 Funktionen zum Steuern der Trace-Historie

Die folgenden Prozeduren / Funktionen in der Script-Sprache dienen zum Steuern der Trace-Historie ("Ablaufverfolgung").

trace.print(<Parameter>)

Druckt die angegebenen Parameter (Zeichenketten und numerische Werte) als einzeiligen Eintrag an das Ende der Trace-Historie.
Dies funktioniert im Simulator (Programmiertool) aber auch den meisten Geräten mit 32-Bit-CPU (ARM).
Beispiel (druckt das aktuelle Datum und Uhrzeit in die Trace-Historie):

```
  trace.print( "Date,Time: ", time.unix_to_str( "YYYY-MM-DD hh:mm:ss", system.unix_time ) );
```

trace.enable

Mit dieser formalen Variablen kann das Script die Trace-Historie stoppen, z.B. um keine weiteren Ereignisse vom CAN-Bus dort eintragen weil im Script bereits ein CAN-Bus-Problem festgestellt wurde, und der später folgende CAN-Bus-Verkehr für die Fehlersuche irrelevant ist.
Die globale Varialble 'trace.enable' enthält eine Bit-Kombination aus den folgenden Konstanten:

- traceCAN1 : alle CAN-Telegramme von/für CAN-Bus 1
- traceCAN2 : alle CAN-Telegramme von/für CAN-Bus 2
- traceCAN_UDP: alle UDP-Telegramme, die zum CAN-via-UDP-Protokoll gehören
- ~~traceFlexRay: alle UDP-Telegramme, die zum Flexray-via-UDP-Protokoll zählen~~
- traceUDP : alle UDP/IP-Datagramme (die vom Terminal empfangen oder gesendet werden)
- traceTCP : alle TCP/IP-Fragmente (die vom Terminal empfangen oder gesendet werden)

Mit trace.enable := 0 (Null) werden keine der oben aufgeführten Ereignisse in der Trace-Historie gespeichert (d.h. abgesehen von 'trace.print' ist die Trace-Historie gestoppt).

Beispiel:

```
  trace.enable := traceCAN1 + traceCAN2; // trace messages from CAN1 and CAN2
  if( can_rx_fifo_usage > 500 ) then
    trace.print( "CAN FIFO usage: ", can_rx_fifo_usage );
    trace.enable := 0; // don't add more CAN messages to the trace history
  endif;
```

Per Default (d.h. nach dem Einschalten des Gerätes) ist die Trace-Historie für CAN1 und CAN2 aktiv, d.h. alle vom 'eigenen' CAN-Controller empfangenen und gesendeten Telegramme werden dort eingetragen.

Auf das Kommando trace.print hat 'trace.enable' keinen Einfluss: Das Script kann *immer* per Kommando in die Trace-Historie 'drucken'.

trace.stop_when_full

Dient zum automatischen Stoppen der Trace-Historie, wenn der Trace-Historienspeicher voll ist.
Per Default (d.h. mit trace.stop_when_full=FALSE) werden die ältesten Einträge überschrieben.
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

trace.num_entries

Liefert die momentane Anzahl von Einträgen in der Trace-Historie.
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

trace.max_entries

Liefert die Größe des Trace-Speichers. Dieser Wert ist zwar konstant, aber firmware-abhängig.
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

trace.oldest_index

Liefert den Puffer-Index, unter dem momentan(!) der älteste Eintrag in der Trace-Historie abgelegt wurde.
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

trace.head_index

Liefert den zirkularen Puffer-Index, unter dem der nächste Eintrag in der Trace-Historie abgelegt werden wird (Futur!).
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

trace.entry[n]

Liefert den n-ten Eintrag in der Trace-Historie als String (Zeichenkette).
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

trace.can_blacklist[i]

Liefert den i-ten Eintrag in der CAN-Identifier-Sperr-Liste, mit der bis zu 10 individuelle CAN-Message-Identifier für die Anzeige in der Trace-Historie **gesperrt** werden können.
Der Index (i) darf z.Z. (2013-11-27) zwischen 0 und 9 liegen, da die Sperr-Liste maximal **zehn** individuelle CAN-Identifier von der Anzeige in der Trace-Historie ausschliessen kann.
Ein Beispiel zum Sperren bestimmter CAN-Message-Identifier finden Sie in der Applikation TraceTest.CVT.

trace.file_index

Liefert die aktuelle Datei-Sequenz-Nummer zum Speichern die Trace-Historie als Datei.
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

trace.save_as_file

Speichert die Trace-Historie als Textdatei.
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

`trace.clear`
Löscht den Inhalt der Trace-Historie.
Z.Z. nur im englischsprachigen Originaldokument spezifiziert.

14 4.10.12 Funktionen zum Steuern der virtuellen Tastatur

In der Script-Sprache stehen seit 06/2020 (*) die folgenden Funktionen zum Steuern der virtuellen Tastatur zur Verfügung:

`vkey.show(N)` : öffnet die virtuelle Tastatur.
Parameter 'N' legt das Aussehen fest:
0 = virtuelle Tastatur nicht *zeigen(show)* sondern *verbergen (hide)*.
1 = kleine Tastatur mit Cursortasten für die Navigation, 'ESCAPE', und 'ENTER'-Taste.
2 = mittelgroße Tastatur für *numerische* Eingaben.
3 = große alphanumerische Tastatur für *alphanumerische* Eingaben.
4 = große alphanumerische Tastatur mit Sonderzeichen / Umlauten / Funktionstasten.
Das folgende Bit-Flag kann mit den o.g. Werten für 'N' verknüpft werden (bitweise Oder-Verknüpfung):
8 = Fenster maximieren (bildschirmfüllend, sinnvoll nur für die 'großen' Tastaturen mit eigenem Editierfeld).

`vkey.move(X,Y)` : Verschiebt die virtuelle Tastatur.
Der Parameter X,Y definiert die obere linke Ecke in Screen-Koordinaten.

`vkey.enable` : Erlaubt das Öffnen der Tastatur per Doppelklick.
Dies ist der voreingestellte Zustand.

`vkey.disable` : Verbietet das Öffnen der Tastatur per Doppelklick.
Wird z.B. in Applikationen verwendet, die den Doppelklick *selbst* verarbeiten, oder die das Öffnen der virtuellen Tastatur per Doppelklick aus anderen Gründen verhindern müssen.

`vkey.connect(var,caption,options[,row,col])` : Verbindet die virtuelle Tastatur mit einer Script-Variablen.
Parameter:
`var` : die mit dem Editierfeld zu 'verbindende' Variable, Typ string oder tTable
`caption` : in der Kopfzeile der virtuellen Tastatur anzuzeigender Text
`options` : reserviert für zukünftige Anwendungen, 0="keine besonderen Optionen"
`row,col` : Zeile und Spalte, falls **`var`** ein tTable-Objekt ist.
Diese 'Zell-Koordinate' wird auch an
OnVirtualKeyboardEvent(event=evEndEdit, param1=row, param2=col) übergeben.

Hinweis: vkey.connect() macht die virtuelle Tastatur wieder sichtbar. Der Aufruf von

vkey.show(N) erübrigt sich dadurch in vielen Fällen (außer zum Script-gesteuerten Umschalten des Tastaturlayouts).

vkey.text : Momentaner Text im Editierfeld der virtuellen Tastatur.
Wird z.B. in Applikationen verwendet, die bereits während der Eingabe (noch vor ENTER) den vom Benutzer eingegebenen Text überprüfen wollen. Diese Pseudo-Variable hat den Datentyp string. Sie wird üblicherweise nur im Event-Handler OnVirtualKeyboardEvent verwendet.

vkey.editing : Flag 'Virtuelle Tastatur im Editier-Modus unter Script-Kontrolle'
Dieses Flag ist nur nach dem Aufruf von vkey.connect (und ggf. vkey.show) gesetzt, bis zum Abschluss der Eingabe durch den Benutzer (i.A. per ENTER). Diese Pseudo-Variable hat den Datentyp bool, d.h. kann nur TRUE (1) oder FALSE (0) sein.
Sie wird üblicherweise nur im Event-Handler OnVirtualKeyboardEvent verwendet.

OnVirtualKeyboardEvent(event, param1, param2) : Optionaler Event-Handler für die virtuelle Tastatur.
Bitte beachten Sie auch die allgemeinen Hinweise zu Event-Handlern ('unverzügliche Rückkehr').

(*) Diese Funktionen erfordern eine Firmware und Programmiertool ab Compilationsdatum 2020-06-05 .
Bei Geräten mit älterer Firmware stehen ähnliche Funktionen nur im Display-Interpreter zur Verfügung.

Beispiele zur Nutzung der virtuellen Tastatur finden Sie in script_demos/VKeyTest.cvt und script_demos/TableTest.cvt .

15 4.10.13 Interaktion zwischen Script und dem Internet-Protokoll-Stack

Bei Geräten mit Ethernet-Schnittstelle ist meistens auch ein TCP/IP-Stack implementiert.
Die in den folgenden Unterkapiteln vorgestellten Funktionen und Event-Handler können für die 'direkte' Interaktion zwischen Script und dem 'Internet Protokoll' (IP) verwendet werden. Für den normalen Einsatz des TCP/IP-Protokolls als Web-Server ist dies allerdings nicht nötig. Zum Beispiel können per Web-Server (HTTP-POST) in die RAMDISK hochgeladene Dateien mit den standardmäßigen Datei-Funktionen bearbeitet werden; und per Script-Sprache in der RAMDISK erzeugte Dateien können per Web-Server (HTTP-GET) aus dem Gerät ausgelesen werden.

Die Entwicklung eigener Internet/Ethernet-Funktionalität ist alles Andere als trivial. Vieles kann unerwartet 'schief gehen', z.B. Probleme mit der Netzwerk-Konfiguration, Probleme mit Firewalls, Portfreigaben, Routern, Verkabelung, Protokollfehler, usw. Im Programmiertool (und, in einigen Fällen, in der Gerätefirmware) stehen deswegen einige Methoden zur Verfügung, die das Entwickeln eigener 'netzwerk-fähiger' Anwendungen erleichtern sollen.
Diese Methoden werden im später folgenden Kapitel 'Testen von Internet/Ethernet-Funktionen' vorgestellt.

< ToDo : Callbacks und Funktionen für die Interaktion zwischen Script und IP-Protokollstack

beschreiben >

Die Beschreibung einer Berkeley-Socket-ähnlichen API finden Sie z.Z. nur in der englischsprachigen Dokumentation .
Ähnliches gilt für die geplanten Erweiterungen (z.B. JSON) für eine openABK-kompatible Schnittstelle in der Script-Sprache.

15.1 4.10.13.1 Übersicht der Internet-'Socket'-API (Script-Funktionen)

Neben dem in der *Firmware* enthaltenen Web-Server kann auch per Script eine eigene 'Internet-Funktionaliät' implementiert werden. Die Script-Sprache enthält zu dem Zweck eine kleine Untermenge der Berkeley Socket API. Details zu den 'Berkeley Sockets' finden Sie in entsprechender Fachliteratur, und in den im englischen Originaltext enthaltenen Beispielen.

Der Begriff 'Socket' hat in diesem Zusammenhang einige Ähnlichkeiten mit den 'Stöpseln' bzw. Verbindungskabeln in historischen Telefonvermittlungen, oder mit der Telefonsteckdose: Vor dem Telefonieren muss das Telefon mit der Gegenstelle verbunden werden. Anno 1890 brauchten die 'Stöpsel' vor deren Verwendung allerdings nicht erst umständlich 'erzeugt' werden.

iSock := **inet.socket**(int address_family,int type,int protocol);

Erzeugt einen neuen Socket ('Verbindungskabel') für einen bestimmten Verwendungszweck. Der zurückgegebene Wert (socket) ist ein Integer, mit dem diese Verbindung für alle weiteren 'Socket'-Funktionen identifiziert wird. Dazu muss der Socket in einer Integer-Variablen gespeichert werden, bis der Socket wieder geschlossen wird (Telefon-Analogie: bis der "Stöpsel" wieder gezogen wird).
Die folgenden Funktionen verwenden den Socket als erstes Argument.

inet.close(int socket)

Gibt alle mit dem Socket zusammenhängenden Resourcen wieder frei. Bei TCP-Sockets wird die Verbindung aktiv getrennt.
Im Gegensatz zu anderen Socket-API-Funktionen gibt inet.close keinen Wert zurück.
Wenn beim Schließen des Sockets noch Netzwerk-Operationen ausstehen (z.B. Zustand CLOSING vor CLOSED), dann steht der Socket *nicht sofort* wieder für andere Zwecke zur Verfügung.

result := **inet.bind**(int socket, int port_number)

Dieser Befehl wird *Server-seitig* verwendet. Auf der *Client-Seite* wird stattdessem der Befehl inet.connect verwendet.
"bind" verbindet den Socket mit einer bestimmten Port-Nummer. Im Gegensatz zur Berkeley-Socket-API interessiert sich inet.bind() um IP-Adressen, denn als server-seitige IP-Addresse wird *immer* die im Netzwerk-Setup des Gerätes (MKT-View) eingestellte Adresse verwendet. Nur die Port-Nummer ist frei wählbar.

result := **inet.listen**(int socket, int nConnections)

Der *Server-seitig* verwendete Befehl 'listen' 'lauscht' in der Telefon-Analogie auf eingehende Anrufe. Im Fachjargon befindet sich der Sockt danach im Zustand 'listening'.

iAcceptedSocket := **inet.accept**(int iListeningSocket)

Wird im Zusammenhang mit 'listen' ebenfalls auf der *Server-Seite* verwendet.
Mit diesem Befehl kann ein eingehender Anruf ('Verbindungswunsch') akzeptiert werden. Dabei wird für den neuen 'Anrufer' ein *neuer* ("akzeptierter") Socket erzeugt, der später (nach dem Datenaustausch) wieder geschlossen und freigegeben werden muss um die dafür allozierten Resourcen wieder freizugeben.
Wie auch bei inet.socket() muss der von inet.accept() gelieferte Socket für die weitere Verwendung in einer Integer-Variablen gespeichert werden.

his_name := **inet.getpeername**(int iAcceptedSocket)

Diese Funktion wird i.A. auf der *Server-Seite* verwendet, um nach erfolgreich 'akzeptierter Verbinding' den Namen des Anrufers zu ermitteln ("IP-Adresse als String).
Hinweis: Die Berkeley-Socket-API nutzt für 'getpeername' eine umständliche Struktur. In der Script-Sprache wird stattdessem eine einfache Zeichenkette (String) verwendet.

iSockState := **inet.getsockstate**(int socket)

Liefert den aktuellen Zustand des angegebenen Sockets.
Das Ergebnis ist eine der als 'SCKS_...' definierten Konstanten - siehe socket states.

result := **inet.connect**(int socket, int timeout_ms, string destination)

Wird *Client-seitig* verwendet, um dem Socket eine freie 'lokale' Port-Nummer zuzuordnen.
Bei TCP-Sockets versucht diese Funktion, eine Verbindung aufzubauen (daher der Name). Abhängig vom Verbindungstyp kann diese Funktion einige hundert Millisekunden benötigen, bis die Verbindung *wirklich* aufgebaut ist. Wird z.B. die Zieladresse nicht 'numerisch' (IP-Adresse) sondern als Name angegeben, dann muss der TCP/IP-Protokoll-Stack zunächst per ARP und/oder DNS alle benötigten Informationen ermitteln.
Der zweite Parameer (timeout_ms) gibt die dafür maximal zu 'investierende' Zeit (in Millisekunden) an. Nach dieser Zeit kehrt inet.connect wieder zum Aufrufer zurück, wobei der Erfolg (oder Misserfolg) per Return-Wert signalisiert wird:
0 = SOCK_SUCCESS,
negative Werte sind die hier definierten Fehlercodes.

result := **inet.send**(int socket, int timeout_ms, input_arguments)

Sendet Daten zum 'anderen Ende' der Verbindung. Ist der Sendepuffer bereits voll, wird das Kommado blockiert, bis genug Daten gesendet wurden um die neuen Daten an den Puffer anzuängen. Die maximale Wartezeit kann begrenzt werden (timeout_ms in Millisekunden). Ist noch ausreichender Platz im Sendepuffer vorhanden, wartet send() *nicht*, denn das eigentliche Senden findet im Hintergrund (in einer anderen Task) statt.

result := **inet.recv**(int socket, int timeout_ms, output_arguments)
Empfängt Daten vom 'anderen Ende' der Verbindung. Liefert bei Erfolg die Anzahl empfangener Bytes (auch Null ist dabei kein Fehler, dies bedeutet lediglich 'keine neuen Daten empfangen'). Im Fehlerfall liefert inet.recv einen negativen Fehlercode (SOCK_ERROR_ ...).
Optional kann die Funktion auch auf den Empfang neuer Daten *warten*. Verwenden Sie dazu einen positiven Timeout-Wert (Anzahl von Millisekunden).
Die Daten werden *per Referenz* übergeben, d.h. mit dem 'Adress-Operator' vor dem Namen der Zielvariablen.

Beispiele zur Verwendung der Internet-'Socket'-API finden Sie z.Z. nur in der englischsprachigen Dokumentation !

15.2 4.10.13.2 Zustandsdiagramm eines Internet-Sockets

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.3 4.10.13.3 Fehlercodes für die Internet Socket Services

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.4 4.10.13.4 inet.socket(int address_family, int socket_type, int protocol)

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.5 4.10.13.5 inet.bind(int socket, int port_number)

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.6 4.10.13.6 inet.listen(int socket, int nConnections)

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.7 4.10.13.7 inet.accept(int iListeningSocket)

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.8 4.10.13.8 inet.connect(int socket, int timeout_ms, string destination)

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.9 4.10.13.9 inet.send(int socket, int timeout_ms, input_arguments)

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.10 4.10.13.10 inet.recv(int socket, int timeout_ms, output_arguments)

Details sind z.Z. nur in der englischsprachigen Dokumentation verfügbar !

15.11 4.10.13.11 JSON (Javascript Object Notation)

Auch wenn die Script-Sprache in keinster Weise kompatibel mit Javascript ist, wird sie auch das im Internet oft verwendete JSON-Format unterstützen (geplant).
Damit wird z.B. auch eine einfache Schnittstelle zu von BMW initiierten openABK ('offenes Anzeige- und Bedien-Konzept'), realisierbar sein.
Details zum JSON-Format finden Sie z.Z. leider nur in der englischsprachigen Dokumentation !

15.12 4.10.13.12 Testen der Internet / Ethernet - Kommunikation

Trotz (oder grade wegen) der weiten Verbreitung von Internet-basierenden Übertragungsprotokollen ist die Implementierung eigener TCP/IP- oder UDP-basierender Applikationen alles Andere als trivial.
Vieles kann 'schiefgehen' - z.B. falsche oder inkompatible Netzwerk-Einstellungen, Ärger mit Firewalls, Routern, Kabeln, Protokollen, blockierte oder bereits belegte Ports, usw.
Um die Fehlersuche in Netzwerk-basierenden Applikationen zu erleichtern, bieten das Programmiertool und die Geräte-Firmware die folgenden Hilfsmittel:

- Der Status und die Aufruf-Reihenfolge der in diesem Kapitel beschriebenen Internet-Funktionen kann in der Trace-Historie verfolgt werden
  (klicken Sie dazu auf den 'Schraubenschlüssel' auf der Registerkarte 'Script', wählen den Menüeintrag 'Optionen für die Trace-Historie', und setzen den Haken bei ***trace 'inet' function calls*** / ***'inet'-Funktionsaufrufe protokollieren***)
- Aus Sicht des Terminals *empfangene* und *gesendete* Ethernet-Pakete können mit der Wireshark-kompatiblen Packet-Capture-Option aufgezeichnet werden, und -danach- per Wireshark analysiert werden.
  Bei Geräten wie z.B. MKT-View IV kann der Inhalt des Capture-Puffers auch *ohne Wireshark* auf dem Display 'aufgelistet' werden. Wählen Sie dazu im 'Sytem-Menü / Setup' zunächst 'Diagnostics', dann 'Ethernet Capture'.
- Tritt beim Aufruf einer 'inet'-Funktion ein Fehler auf, z.B. weil der Netzwerk-Protokoll-Stack die gewünschte Operation nicht durchführen kann, wird eine entsprechende Fehlermeldung (oder Warnung) am Ende der Liste auf der Registerkarte Fehler und Meldungen im Programmiertool (Simulator) angezeigt. Beispiel (mit der Beschreibung des Fehlers, vom Betriebssystem auf einem Windows-PC erzeugt):
  `inet.bind(port=49152) failed : "Only one usage of each socket address (..) is normally permitted"`
  Die Ursache für den oben gezeigten Fehler (im Simulator) wird im weiteren Verlauf dieses Kapitels (*) noch genauer erläutert.

- Programmieren Sie ihre Scripte 'defensiv'. Rechnen Sie damit, dass der Aufruf jeder der in diesem Kapitel beschriebenen 'Internet'-Funktionen fehlschlagen könnte.
  Die meisten dieser Funktionen liefern im Erfolgsfall einen nicht-negativen Wert zurück, oder -im Fehlerfall- einen negativen Fehlercode, dessen Wert auch die Fehlerursache eingrenzen kann. Die in der Script-Sprache verwendeten Fehlercodes finden Sie in einem anderen Kapitel.
  Sie könnten z.B., wenn Sie die Fehlerursache nicht innerhalb des Scripts selbst abfangen können, den Bediener mit einer entsprechenden Meldung (als Klartext) auf das Netzwerk-Problem hinweisen.
  Im 'Internet Demo' wird eine selbstdefinierte Funktion (InetErrorCodeToString) verwendet, um alle möglichen 'Socket'-Fehlercodes in einen entsprechenden String umzuwandeln.

Wird ein Script (mit Internet-Funktionalität) *auf dem PC* (im Simulator) abgearbeitet, treten oft Fehler auf, die im 'echten' Gerät nicht zu beobachten sind. So wird z.B. der Versuch, per 'bind' einen Internet-Socket ("Anschluss") mit einem bestimmten Port zu verbinden, oft vom Betriebsystem verweigert. Die Ursache war meistens ein anderes Programm (welches auf dem gleichen PC läuft), welches den Port für 'eigene Zwecke' geöffnet hatte.
Der Simulator versucht durch den Aufruf einer Betriebssystem-Funktion einen Klartext mit einer Fehlerbeschreibung zu erzeugen. Dieser Klartext wird an die eigentliche Fehlermeldung angehängt, und im Programmiertool auf der Registerkarte 'Fehler und Meldungen' angezeigt. Beispiel:

```
  Quelltext (script) :
     // Bind the 'listening' socket to the server's port number:
     iResult := inet.bind( iListeningSocket, display.RemotePort );

  Fehlermeldung (beim Versuch, das Script im Simulator laufen zu lassen):
     inet.bind(port=49152) failed : "Only one usage of each socket address (..)
is normally permitted"
                      |___|            |
_____________________________________________________________|
                        |                                          |
                        |__ Ist dieser Port eventuell schon "belegt" ?  Von
Windows erzeugter Fehlertext
```

Um herauszufinden, ***wer*** (auf dem PC) welchen TCP-Port belegt, verwenden Sie das Kommando 'netstat' in einem Konsolenfenster ("cmd.exe").
Beispiel (auf einem PC mit Windows 7; unter Windows 10 funktioniert "netstat -abnop TCP" leider nur mit Administratorrechten):

```
C:\>netstat -abnop TCP

Aktive Verbindungen

  Proto  Lokale Adresse          Remoteadresse          Status           PID
  TCP    0.0.0.0:80             0.0.0.0:0              ABHÖREN         1296
(HTTP server)
 [ctptwin1.exe]
  TCP    0.0.0.0:135            0.0.0.0:0              ABHÖREN         400
```

```
  .....
[svchost.exe]
 TCP    0.0.0.0:5357           0.0.0.0:0              ABHÖREN         4
Es konnten keine Besitzerinformationen abgerufen werden.
 TCP    0.0.0.0:49152          0.0.0.0:0              ABHÖREN         772
[wininit.exe]
  ......
 TCP    127.0.0.1:49797        127.0.0.1:49796        HERGESTELLT     620
[firefox.exe]
  ......
```

In Kurzform bedeutet die obige netstat-Ausgabe: 'Pech gehabt, denn TCP-Port 49152 ist bereits von "wininit.exe" belegt', und -höchstwahrscheinlich- wird der Socket-Funktion 'bind' die Nutzung dieses Ports aus dem Grund verwehrt.
Wird das Script dagegen im 'echten' Gerät (z.B. MKT-View III / IV) ausgeführt, tritt dieses Problem nicht auf, denn (im Gegensatz zu einem Windows-PC) wird dort *garantiert* keine Software namens 'wininit.exe' laufen, die Port 49152 blockiert !

Siehe auch (externer Link): Wireshark-kompatible 'Packet Capture'-Option zum Aufspüren von Netzwerk-Problemen.

16 4.10.14 Interaktion zwischen Script und dem CANopen-Protokoll-Stack

Hinweis: Die nachfolgend beschriebenen Funktionen stehen nur bei Geräten mit CANopen-Protokoll-Stack und im 'UPT-Programmiertool II' (als Simulator) zur Verfügung !
Da *die meisten* Geräte (z.B. "MKT-View" II / III / IV) eher mit 'CANdb-Unterstützung' (automotive) statt mit der Firmware-Variante für CANopen (automation) ausgeliefert werden, empfiehlt sich ggf. die Abfrage der Verfügbarkeit von CANopen per cop.supported, bevor andere CANopen-Funktionen aus dem Script aufgerufen werden. So können portable Scripte programmiert werden, die die CANopen-Funktionen nur dann verwenden, wenn sie zur Verfügung stehen. Ob für Ihr Gerät eine Firmware-Variante mit CANopen zur Verfügung steht, können Sie in der Feature-Matrix kontrollieren. Dort finden Sie ggf. auch die Artikelnummer für die Firmware mit 'CANopen' oder 'CANdb'.

Kommandos und Funktionen mit dem Prefix 'cop.' (Kürzel für 'CANopen'):

cop.supported
Liefert den Wert 1 (TRUE), wenn die aktuell geladene Geräte-Firmware die nachfolgend beschriebenen CANopen-Funktionen unterstützt.
Andernfalls liefert cop.supported den Wert 0 (FALSE). In dem Fall stehen statt CANopen nur die Kapitel 4.10.6 beschriebenen Funktionen zum Senden und Empfangen von CAN-Telegrammen zur Verfügung.

cop.obd(Index,Subindex[,Data_type])
Ermöglicht den Zugriff auf ein Objekt im *eigenen* CANopen-Objektverzeichnis (Object Dictionary, OD).
Da bei CANopen-Geräten *fast alles* über das Objektverzeichnis gesteuert wird, kann das Script mit dieser Funktion (als Schreibzugriff, d.h. formale Zuweisung) das Verhalten des *eigenen* Gerätes bezüglich der CANopen-Kommunkation beeinflussen, z.B.:

- Umkonfigurieren der Prozessdaten-Kommunikation mit Hilfe der PDO-**Kommunikations**-Parameter
  (Objekt-Indizes 0x1400=RPDO1 CommPar, 0x1401=RPDO2, .. , 0x1800=TPDO1, 0x1801=TPDO2, ..)
- 'Umprogrammieren' des *Inhalts* der Prozessdaten-Objekte mit Hilfe der PDO-**Mapping**-Parameter
  (Objekt-Indizes 0x1600=RPDO1 Mapping, 0x1601=RPDO2, .. , 0x1A00=TPDO1, 0x1A01=TPDO2, ..)
- Umkonfigurieren der SDO-Clients (die hier u.A. zur Kommunikation per cop.sdo dienen)
  (Objekt-Indizes z.B. 0x1280=erster SDO-Client, 0x1281=zweiter SDO-Client, etc..)
- Umkonfigurieren der SDO-Server (die anderen CANopen-Geräten den Zugriff auf das Terminal ermöglichen)
  (Objekt-Indizes z.B. 0x1200=erster SDO-Server, 0x1201=zweiter SDO-Server, etc..)

Ein Beispiel zum Umprogrammieren eines PDO-Mapping-Parameters per 'cop.obd' finden Sie unter CANopen1.upt .

cop.sdo(Index,Subindex[,Data_type][,Timeout_in_milliseconds]) ,
cop.sdo2(SDO-Channel,Index,Subindex[,Data_type][,Timeout_in_milliseconds])

Diese Funktionen dienen zum Zugriff auf ein Objekt in einem *fremden* (externen) CANopen-OD per SDO (Service Data Object).
Beide unterscheiden sich nur geringfügig: cop.sdo() verwendet den ersten SDO-Client-Kanal, bei cop.sdo2() wird die Kanalnummer als erstes Funktionsargument übergeben (0..5, siehe SDO-Kanäle).
Die Funktion benötigt einige Zeit für die Abwicklung des CANopen-SDO-Protokolls. Während dieser Zeit wird der Aufrufer (im Script) blockiert, und das Script in den Zustand 'Warten' umgeschaltet. Die Wartezeit kann allerdings durch Event-Handler (z.B. Timer-Events) unterbrochen werden. Aus dem Grund **darf *cop.sdo* nicht in Event-Handlern verwendet werden**.
Rufen Sie *cop.sdo* daher nur aus der Hauptschleife des Scripts auf !

Da cop.sdo() verschiedene Datentypen (int,float,string,...) zurückliefern kann, empfiehlt es sich zur Speicherung des Ergebnisses eine Variable vom Typ anytype zu verwenden (siehe Beispiel unter 'anytype'). Tritt beim Zugriff per CANopen (SDO) ein Fehler auf, dann erhält das Ergebnis den Typ 'Fehler' (d.h. typeof(result) == dtError); der Wert des Ergebnisses ist in diesem Fall ein im CANopen-Standard (CiA 301) als 'Abort Code' bezeichneter Fehlercode. Einen Auszug aus der Liste von CANopen-Abort-Codes finden Sie in der Beschreibung der Funktion cop.error_code.

Beispiele für Zugriffe auf das Objekt-Dictionary (OD), "lokal" und per SDO:

```
   // Objekt 0x1018, Subindex 0x01 = "Vendor ID", Teil vom "Identity
Object", siehe CiA 301 :
   my_vendor_id  := cop.obd( 0x1018, 0x01, dtDWord );  // read vendor ID
from this device's own OD
   his_vendor_id := cop.sdo( 0x1018, 0x01, dtDWord );  // read vendor ID
from a REMOTE device's OD
   cop.error_code := 0;  // Clear old 'first' error code ('abort code')
before the next SDO access
   cop.sdo(od_index, subindex ) := iWriteValue;        // try to write into
remote object via SDO
   if( cop.error_code <> 0 ) then  // if the previous SDO accesses were ok,
cop.error_code is zero
      print("\r\nSDO access error, abort code =
0x",hex(cop.error_code,8) ); // show abort code
   endif;
   iReadBackValue := cop.sdo(od_index, subindex, dtInteger ); // try to
read remote object via SDO
   if ( iWriteValue <> iReadBackValue ) then
     print("\r\nFehler: Gelesener Wert(",iReadBackValue,") <> geschriebener
Wert(",iWriteValue,")");
   endif;
   print("\r\n Mein Name ist ",cop.obd( 0x1008,0, dtString ) );  //
"Hase" ? wohl kaum, sondern..
   print("\r\n Sein Name ist ",cop.sdo( 0x1008,0, dtString ) );  // ..
'Manufacturer Device Name'
```

Weitere Beispiele zur Verwendung der CANopen-SDOs finden Sie im Demoprogramm CANopen1.upt .

SDO-Clients werden üblicherweise wie hier beschrieben im Programmiertool konfiguriert.
Die Spezifikation der CANopen-SDO-Protokolle ist in CANopen CiA 301 enthalten.

cop.error_code

Diese Variable wird beim Auftreten eines SDO-Protokollfehlers auf den vom SDO-Server per CAN-Bus gesendeten 'Abort Code' gesetzt. Der Wert wird i.A. als achtziffriger Hex-Code angezeigt, oder kann (wie im Beispiel CANopen1.upt, "ErrorCodeToString") mit einer select-case-Liste in eine lesbare Fehlermeldung umgesetzt werden.
Tritt *kein* Fehler beim SDO-Transfer auf, ändert sich der Wert von cop.error_code *nicht*. Ist cop.error_code bereits ungleich Null (von einem noch nicht 'quittierten' Fehler), ändert sich der Wert von cop.error_code (und cop.error_index und cop.error_subindex) ebenfalls *nicht*.
Zusammen mit **cop.error_code** werden auch die Variablen **cop.error_index** (auf den CANopen-OD-Index des Objektes, bei dessem Zugriff der erste Fehler auftrat) und **cop.error_subindex** (auf den entsprechenden Objekt-Subindex) gesetzt.
Zum Quittieren des Fehlers (d.h. wenn das Script nach einem oder mehreren SDO-Zugriffen den Abort-Code 'zur Kenntnis genommen hat') setzen sie cop.error_code wie im folgenden Beispiel wieder auf Null:

```
cop.error_code := 0;  // Clear old CANopen error code (0="no error")
```

Die CANopen-SDO-Abort-Codes sind in CANopen CiA 301 spezifiziert. Hier nur ein *kleiner Auszug*:

CANopen Abort Code	Beschreibung
0x05030000	Toggle bit not alternated
0x05040000	SDO protocol timed out
0x06010000	Unsupported access to an object
0x06010001	Attempted to read a write-only object
0x06010002	Attempted to write a read-only object
0x06020000	Object does not exist in the object dictionary
0x08000000	General error

Zusätzlich zu dem oben gezeigten Abort-Codes kann die Funktion cop.sdo *im Fehlerfall* einen der folgenden 'Pseudo'-Abort-Codes zurückliefern, die *nichts mit dem CANopen-Standard* zu tun haben.
In diesen 'Pseudo'-Abort-Codes ist Bit 28 gesetzt (hex. Maske 0x10000000), was bei den CANopen-Abort-Codes nie vorkommt.
Der Typ des von cop.sdo zurückgelieferten Wertes ("Return-Wert") ist dann dtError.

Pseudo Abort Code	Beschreibung
0x10000001	Cannot call cop.sdo() now due to CAN bus trouble (check CAN status !)
0x10000002	Cannot call cop.sdo() now due to NMT state ("bootup" or "stopped")
0x10000003	Cannot call cop.sdo() now because CANopen is not initialized yet
0x10000004	Cannot call cop.sdo() now because the SDO client is currently busy from another
0x10000005	Cannot call cop.sdo() because the SDO client is not available (wrong channel, no

0x10000006	Cannot call cop.sdo() due to data type incompatibility ("can't convert")
0x10000007	Cannot call cop.sdo() because the simulator is not running
0x10000008	Cannot call cop.sdo() because the SDO client is occupied by the 'node scanner'
0x10000009	Cannot call cop.sdo() because the CAN port is in 'Gateway' mode
0x10000000	Other 'internal' reason why the script must not call cop.sdo() at the moment

Siehe auch: Tabelle mit weiteren CANopen-SDO-Abort-Codes in der Beschreibung des alten *Display-Interpreters*.

cop.nmt_state

Liefert den aktuellen NMT-Zustand des CANopen-Gerätes (im Terminal, auf dem das Script ausgeführt wird).
Die folgenden Werte sind möglich:

- **cNmtStateBootup** (0) :
  Das CANopen-Gerät befindet sich in der Initialisierungsphase; es kommuniziert nicht per CANopen, und das Objektverzeichnis (OD) ist noch nicht verfügbar (weder per SDO noch per lokalem Zugriff).
- **cNmtStatePreOperational** (127) :
  Im NMT-Zustand 'Pre-Operational' ist zwar Kommunikation per SDO (Service-Daten-Objekt) möglich, das OD existiert, aber PDO-Kommunikation (Prozessdatenobjekte) ist nicht erlaubt (...)
  Das CANopen-Gerät kann in den NMT-Zustand 'Operational' geschaltet werden, indem ein entsprechendes NMT-Kommando ('Start Remote Node') gesendet wird, oder durch die 'lokale Steuerung' (bei den Geräten von MKT z.B. durch den CANopen-'Bootup-Timer'.
- **cNmtStateOperational** (5) :
  Alle "Kommunikationsobjekte" (CANopen-Slang) sind aktiv. Während der Umschaltung nach 'Operational' wurden auch alle PDOs erzeugt, wobei die entsprechenden PDO-Kommunikations- und -Mapping-Parameter im OD ausgewertet wurden.
- **cNmtStateStopped** (4) :
  Abgesehen von Node-Guarding oder Heartbeat nimmt das CANopen-Gerät nicht (mehr) an der CAN-Kommunikation teil.

Details zur 'NMT state machine' eines jeden CANopen-Gerätes finden Sie in CiA 301 (früher als 'DS 301' bekannt..), Version 4.2.0 (Februrar 2011), Kapitel 7.3.2, Seite 83 bis 85. Die restriktiven Nutzungsbedingungen (in CiA 301) erlauben es leider nicht, die Information aus dem CANopen-Standard hier zu übernehmen. Besorgen Sie sich daher bitte selbst eine Kopie der CANopen-Spezifikation CiA 301.

cop.node_id

Liefert die 'lokale Knotennummer', d.h. den CANopen-Node-ID (1..127) des Gerätes, auf dem das Script ausgeführt wird.

Der Wert ist nur lesbar. Der Node-ID des Gerätes kann (bei MKT-Geräten) nur über das System-Setup geändert werden.

cop.SendNMTCommand(int node_id, int wanted_nmt_state)

Sendet ein NMT-Kommando an den eigenen CANopen-Slave oder *an das CANopen-Netzwerk*, um einen (oder alle) CANopen-Slaves in den gewünschten NMT-Zustand (wanted_nmt_state) umzuschalten.
Gültige CANopen-Node-IDs sind 1 bis 127. Zusätzlich sieht das NMT (Network Management) - Protokoll die Node-ID 0 (Null) zum Adressieren *aller* Slaves im Netzwerk vor.
Stimmt der als erstes Funktionsargument angegebene Node-Id mit dem lokalen Node-ID überein, dann schaltet auch der lokale Knoten (d.h. das Gerät auf dem das Script läuft) in den neuen NMT-Zustand um.
Wie auch bei cop.nmt_state werden folgende symbolischen Konstanten für den NMT-Zustand verwendet (hier: 'wanted_nmt_state', d.h 'gewünschter' NMT-Zustand) :

- **cNmtStateBootup** : Erzwungener Neustart (Bootup), sendet das NMT-Kommando 'Reset Node' an einen oder alle Slaves.
- **cNmtStatePreOperational** : Umschalten in den Zustand 'Pre-Operational'.
- **cNmtStateOperational** : Umschalten in den Zustand 'Operational'.
- **cNmtStateStopped** : Umschalten in den NMT-Zustand 'Gestoppt'.

Rückgabewert (wenn cop.SendNMTCommand als *Funktion* aufgerufen wird) :
TRUE = ok
FALSE= Fehler (Funktion nicht verfügbar, ungültiger Node-ID, ungültiges CANopen-NMT-Kommando, etc...)

Siehe auch: cop.nmt_state : Liefert den *aktuellen* NMT-Zustand

cop.SetPDOEvent(int pdo_comm_par_index)

Setzt das 'Event'-Flag für ein Prozessdatenobjekt (PDO), was u.U. das Senden des Prozessdaten-Telegramms bewirkt.
Zur Identifikation des PDOs dient der CANopen-OD-Index des entsprechenden 'Kommunikationsparameters', z.B.:
0x1800 = TPDO1 (erster Transmit-PDO), **0x1801** = TPDO2, usw.
Ob das Setzen des 'Event'-Flags wirklich zur sofortigen Sendung führt, hängt u.A. auch vom 'PDO Transmission Type' ab, der üblicherweise im Programmiertool per Dialog für PDO-Kommunikations-Parameter eingestellt wird.
Darüberhinaus könnte das 'sofortige' Senden eines PDOs auch durch dessen *inhibit time* verhindert werden (die 'inhibit time', ebenfalls Teil des PDO-Kommunikationsparameters, begrenzt die maximale Frequenz mit der ein PDO gesendet werden kann).
Rückgabewert (wenn cop.SetPDOEvent als *Funktion* aufgerufen wird) :
TRUE = ok (Setzen des PDO-Event-Flags "hat funktioniert")
FALSE= Fehler (Funktion nicht verfügbar, ungültiger CANopen-OD-Index, ...)

Leider sind sowohl CANopen als auch die Objekte im CANopen-OD sehr komplex.
Eine komplette Beschreibung von CANopen würde den Rahmen dieser Beschreibung sprengen.
Wir empfehlen eine gut verständliche Fachliteratur. Die offiziellen CANopen-Standards (allen

voran CiA 301) sind leider eher 'für Experten' !

Einige einfache Beispiele für die Verwendung der CANopen-Funktionen, incl. 'Umprogrammieren' eines Prozessdaten-Objektes per Script, finden Sie in der Applikation CANopen1.upt .

Siehe auch:

- Das (eigene) CANopen-Objektverzeichnis (OD) des Terminals
- Objekt 0x5001 im OD : PDO-map-bare 'Tastatur-Matrix'
- Besonderheiten der programmierbaren Terminals mit "CANopen V4""
- PDO-Mapping (Definition des Aufbaus der 'Prozessdaten-Telegramme')
- SDO-Abort-Codes (Fehlercodes bei Kommunikation per CANopen-SDO)
- CANopen-Spezifikationen von CiA (CAN in Automation), besonders wichtig: CiA 301

17 4.10.15 Erweiterungen für die Kommunikation per J1939

Hinweis: Die nachfolgend beschriebenen Funktionen stehen nur bei Geräten zur Verfügung, in denen auch die elementaren CAN-Sende- und Emfpangs-Funktionen *in der Script-Sprache* nutzbar sind ! Bei manchen Geräten (z.B. MKT-View II, III) ist dazu eine Freischaltung nötig.

Zur Erleichterung der Kommunikation per J1939 enthält die Script-Sprache seit September 2013 die folgenden Ergänzungen im Datentyp tCANmsg:

`.PRIO` : J1939 'Message Priority'
Alias für CAN-ID Bits 26 bis 28.

`.EDP` : J1939 'Extended Data Page' bzw. 'Reserved' (z.Z. immer Null)
Alias für CAN-ID Bit 25.

`.DP` : J1939 'Data Page' (z.Z. wohl auch immer Null)
Alias für CAN-ID Bit 24.

`.PF` : J1939 'PDU Format'
Alias für CAN-ID Bits 16 bis 23.

`.PS` : J1939 'PDU Specific'
Alias für CAN-ID Bits 8 bis 15.

`.SA` : J1939 'Source Address' (gehört *nicht* zur PGN)
Alias für CAN-ID Bits 0 bis 7.

`.PGN` : 'Parameter Group Number' mit *meistens* 16 Bit.
Besteht i.A. aus den Feldern 'DP' und 'PF', aber ohne 'PS' ! Gilt nur wenn PF<240 !
Wenn PF>=240, dann besteht die PGN strenggenommen nicht nur aus 'DP' und 'PF', sondern enthält (in den niederwertigsten Bits) auch das Bitfeld 'PS'. In manchen Quellen (z.B. 'Bussysteme in der Fahrzeugtechnik') werden Bits 8 bis 15 im CAN-ID ("PS" alias "PDU Specific" oder "Destination Address" bzw. "Group Extension") als Bestandteil der 'PGN' betrachtet; *frei nach* der Präsentation 'Understanding SAE J1939' von Simma Software:

```
if PF < 240
  then PGN := (DP << 15) + (PF << 8);
  else PGN := (DP << 15) + (PF << 8) + PS;
```

Die Fallunterscheidung, ob in den niederwertigen 8 Bits *innerhalb der PGN* die 8 bit breite 'PS' ('PDU-Specific', d.h. je nach PDU-Format 'Destination Address' oder 'Group Extension') eingeblendet werden muss, muss ggf. *im Script* implementiert werden !
Beispiele zum Senden und Empfangen verschiedener PGN (Parameter-Gruppen-Nummern) finden Sie im Demo 'J1939-Simulator'.

In der Praxis sind bei J1939 sowohl das 'reservierte' EDP-Bit ('Extended data page') als auch das 'DP'-Bit ('Data Page') Null. Daher sind *fast alle* in der Praxis verwendeten PGN kleiner als 65536. Sie werden meistens als Dezimalzahl angegeben, und sind in SAE J1939/71 (Datenübertragung im Fahrbetrieb) und J1939/73 (Diagnose bzw. 'Off-Board-Kommunikation') spezifiziert.

Da für eine nicht-triviale J1939-Implementierung eine große Anzahl verschiedener CAN-Message-Identifier *empfangen* werden muss (weit mehr als die per can_add_id registrierbaren), wurde das Kommando CAN.add_filter implementiert. Damit ist es möglich, den für J1939 verwendeten 'riesigen', und i.A. nicht von vornherein bekannten CAN-Message-Identifier-Bereich durch einen einzigen Prozeduraufruf für den Empfang im Script zu registrieren (im folgenden Beispiel

zusammen mit einem CAN-Empfangs-Handler, der kurz nach dem Empfang eines CAN-Telegramms im entsprechenden Identifier-Bereich aufgerufen wird):

```
// Register a large range of 29-bit-CAN-message-identifiers for reception:
CAN.add_filter( 0x20000000, 0x20000000, addr(OnCANReceive) );
  // .. acually receives EVERY possible message with 29-bit-ID,
  //    which may be too much for old microcontrollers (MKT-View II)..
```

Ein Script-Beispiel zur 'Simulation' eines Steuergerätes (ECU) mit J1939-Protokoll-Handler finden Sie in der Applikation 'J1939-Simulator' (script_demos/J1939sim.cvt) .

17.1 4.10.16 Erweiterungen für die Kommunikation per ISO 15765-2 ("ISO-TP")

Die Unterstützung für ISO 15765-2 ("ISO-TP") *in der Script-Sprache* befand sich im Mai 2015 noch in der Entwicklungsphase.
Dazu zählen u.A. die Aliase für ISO-TP im 29-Bit CAN-ID (z.B. tCANmsg.ISO_TA, tCANmsg.ISO_TA).
Ein einfaches Script zur Kommunikation per ISO-TP finden Sie in den mitgelieferten Beispielen.
Mangels einer geeigneten Testumgebung (Steuergerät mit brauchbarer Dokumentation) konnten die ISO-TP-Funktionen bislang nicht getestet werden.

4.11 Die Behandlung spezieller Ereignisse ('Event-Handling' in der Script-Sprache)

Als Ersatz für die alte (und nicht sehr komfortable) Bearbeitung von 'Events' im Display-Interpreter kann auch die Script-Sprache verwendet werden, um auf bestimmte Aktionen des Benutzers zu reagieren. Dazu gehören z.B. das Drücken einer Taste, das Betätigen des Touchscreens, oder das Drehen oder Drücken eines eventuell vorhandenen Drehknopfes.
Das Script ist auch in der Lage, bestimmte Ereignisse abzufangen(!), so daß deren normale Bearbeitung durch das System ausbleibt (im Fachchinesisch: Es kann verhindert werden, daß für bestimmte Ereignisse deren 'Default-Handler' aufgerufen wird).

In der Script-Sprache können eigene Message- (bzw. Event-) Handler in der Form von anwenderdefinierten Funktionen programmiert werden. Die Funktionsnamen sind dabei allerdings (im Gegensatz zu "normalen" Funktionen) nicht frei wählbar. Der Rückgabewert der Funktion teilt dem System mit, ob das Ereignis verworfen werden soll (z.B. weil die betätigte Taste in der Script-Funktion verarbeitet wurde, und nirgendwo anders einen Effekt haben soll) oder nicht. Mehr dazu später. Im folgenden Kapitel werden zunächst die Message-Handler für die 'unterste Ebene' (low-level message handler) vorgestellt: Tastatur, Encoder, und Touchscreen.

Hinweis:

In der Script-Sprache programmierte Event-Handler **unterbrechen** den normalen Programmablauf in dem Moment, in dem das Ereignis vom System erkannt wird. Dies ist u.A. nötig, damit das System anhand des Return-Wertes entscheiden kann, ob nach dem Aufruf des Handlers im Script auch der systemeigene "Default-Handler" aufgerufen werden soll. Um zu vermeiden, dass das System dadurch 'ausgebremst' oder blockiert wird, muss der Event-Handler so schnell wie möglich wieder zum Aufrufer zurückkehren - im Idealfall nach weniger als 50 Millisekunden. Bleibt das Script wegen eines Programmierfehlers im Event-Handler für länger als ca. 500 Millisekunden "stecken", dann wird die Bearbeitung (im Script) abgebrochen, und das Ereignis (unabhängig vom bis dahin noch unbekannten 'Rückgabewert') an den Default-Handler weitergegeben.
Darum: Halten Sie Ihren Event-Handler möglichst kurz, und kehren 'so schnell wie möglich' aus dem Interrupt-ähnlichen Zustand wieder zurück !
Um 'langsame' Verarbeitung im Event-Handler zu vermeiden, setzen Sie im Handler ggf. nur einen Merker ("Flag"), den Sie dann (später) in der Haupschleife des Scriptes abfragen können, und die eigentliche Verarbeitung so aus dem "Interrupt" in die Hauptschleife verlagern.
Verwenden Sie in Event-Handlern keine potenziell 'blockierenden' Kommandos wie wait_ms, inet.connect, inet.recv, inet.send ! Der Aufruf solcher Kommandos könnte dazu führen, dass der Aufruf des Event-Handlers unplanmässig terminiert wird um das System nicht (aus Sicht des Bedieners) 'abstürzen' zu lassen.

Ähnliche Einschränkungen gelten auch für Aufrufe des Scripts aus dem Anzeigeprogramm (Display-Interpreter).

Sollte die oben erwähnte Zeit zum Abarbeiten des Event-Handlers nicht ausreichen, kann *im äußersten Notfall* die Bearbeitungsdauer per 'system.feed_watchdog' verlängert werden. Dies kann aber die bereits erwähnten Konsequenzen bezüglich der 'flüssigen' Bedienbarkeit haben.

Alle in den folgenden Kapiteln vorgestellten Event-Handler werden erst aktiv, wenn das Script mit seiner eigenen Initialisierung fertig ist (z.B. der Initialisierung von globalen Variablen, Laden von Übersetzungstexten aus Dateien, etc). Am Ende der Initialisierungssequenz sollte im Script dazu *genau ein* Aufruf von **init_done** stehen. Erst durch den Aufruf von "init_done" werden u.A. die Event-Handler aktiviert.
Beispiel (aus script_demos/ButtonEventDemo.cvt):

```
var
  int i,imax;
  int CanSignals[10]; // declare an array with 10 integers
endvar;

// Initialize the script's own variables:
imax := CanSignals.size(0)-1;
for i:=0 to imax
   CanSignals[i] := 11*i; // fill array with defaults
next i;

init_done; // let the system know "we're open for business" (enable event
handlers)

  ...

func OnControlEvent(
         int event,     // [in] type of the event, like evClick, etc
         int controlID, // [in] control identifier (from page-def-table)
         int param1,    // [in] 1st message parameter, depends on event
         int param2 )   // [in] 2nd message parameter, depends on event
  // Called when 'something happens' with a certain control element
  // (button, menu item, edit field, etc) on the current display page .
 ...
```

Fehlt der explizite Aufruf von **init_done** im Script, wird im Interesse der Abwärtkompatibilität das entsprechende interne Flag *sofort* (nach dem Compilieren) gesetzt; Event-Handler und der Aufruf des Scripts aus den Anzeigeseiten sind dann *direkt nach dem Compilieren* aktiv.

1 4.11.1 Low-Level Event Handler

Um bestimmte Benutzeraktionen bereits auf niedrigstem Level zu erkennen, oder diese sogar *abzufangen* (d.h. Ereignisse nicht zur normalen Bearbeitung kommen zu lassen), definieren Sie eine entsprechende Handler-Funktion in Ihrem Script :

```
func OnKeyDown(int keyCode )      // eine 'normale' Taste wurde soeben gedrückt
func OnKeyUp( int keyCode )       // eine 'normale' Taste wurde soeben
losgelassen
func OnKeyMatrixChange( int oldMatrix, int newMatrix ) // Zustand der
Tastaturmatrix geändert
```

```
func OnEncoderButton(int button) // Drehknopf (encoder) wurde gedrückt
(button>0) oder losgelassen (0)
func OnEncoderDelta( int delta ) // Drehknopf wurde links oder rechts gedreht,
um 'delta' Schritte. Beispiel.
func OnPenDown( int x, int y)    // Stift (oder Finger) wurde auf Touchscreen
gedrückt
func OnPenUp( int x, int y)      // Stift (oder Finger) wurde vom Touchscreen
abgehoben
func OnPenMove( int x, int y)    // Stift (oder Finger) wurde auf Touchscreen
BEWEGT
func OnGesture( int gestureCode, int gestureSize) // Touchscreen-Geste beendet,
Stift abgehoben
proc OnPageLoaded( int iNewPage, int iOldPage ) // eine neue Anzeigeseite wurde
aus dem FLASH geladen
proc OnPageEnter( int page_nr, string page_name) // die angegebene Anzeigeseite
wird grade "neu betreten"
proc OnPageQuit( int page_nr, string page_name) // die angegebene Anzeigeseite
wird grade verlassen
proc OnPageUpdate( int page_nr, string page_name, int element) // Aktualisierung
im Framebuffer
proc OnVirtualKeyboardEvent(int event, int param1, int param2) // Ereignis von
der virtuellen Tastatur
proc OnLocationChanged()      // GPS-Empfänger hat neue Daten geliefert (siehe
gps.xyz)
proc OnSystemShutdown(int iReason) // System wird 'heruntergefahren'
(abgeschaltet)
```

Die oben aufgeführten Handler müssen nicht 'registriert' werden. Sind sie im Script-Quelltext vorhanden, werden sie beim Eintreffen des entsprechenden Ereignisses automatisch aufgerufen. Beim Aufruf kann das Script anhand der übergebenen Funktionsparameter erkennen, 'was genau' passiert ist (d.h. WELCHE Taste gedrückt oder losgelassen wurde, WO auf dem Touchscreen etwas passierte, usw.).

Für andere Arten von Ereignissen können beliebige Handler-Namen verwendet werden, z.B. für CAN-Empfangs-Handler und periodische Timer-Events. Es wird allerdings empfohlen, auch für solche Handler einheitliche Namen zu verwenden (beginnend mit 'On', um Event-Handler von normalen Funktionen zu unterscheiden), z.B.:

```
func OnCAN_ID123( tCANmsg ptr pRcvdCANmsg) // CAN-Empfangs-Handler (aktiviert
per can_add_id)
func OnTimer1( tTimer ptr pTimer)          // Timer-Event-Handler (gestartet per
setTimer)
```

Wie bei vielen anderen Event-Handlern definiert der Return-Wert, ob das Ereignis "wie üblich" auch vom Default-Handler des Systems verarbeitet werden soll:

Ein Handler liefert normalerweise den Wert 0 (Null, bzw. boolsch 'FALSE') zurück. Damit teilt er dem System mit, daß dieses Ereignis 'ganz normal' weiterverarbeitet werden darf. Der Return-Wert 1 (Eins, bzw. boolsch 'TRUE') bedeutet "Ich habe dieses Ereignis komplett in meinen Script

verarbeitet, und will nicht, daß das System dafür den Default-Handler aufruft um das Ereignis "normal" weiterzuverarbeiten). Auf diese Weise kann der Handler z.B. die 'normale' Verarbeitung bestimmter Tasten verhindern.

Hinweis:

> Wird eine benutzerdefinierte Funktion *nicht* mit der Anweisung 'return <value>' verlassen, dann kehrt die Funktion (beim Erreichen von 'endfunc') mit dem Wert Null (integer) zurück. Dies bedeutet im Falle eines Event-Handlers, dass das Ereignis 'ganz normal' vom System verarbeitet wird (d.h. das Ereignis wird dann *nicht* unterdrückt).

Einige einfache Beispiele finden Sie im Script der Applikation 'EventTest' .

1.1 4.11.1.1 OnPageLoaded(int iNewPage, int iOldPage)

Wenn dieser Handler im Script existiert, dann wird er unmittelbar nach dem Laden einer neuen Anzeigeseite aufgerufen. Dies erfolgt noch *vor* dem 'Page-Enter'- und dem 'Page-Update'-Event, d.h. bevor die Seite zum ersten Mal auf dem Display sichtbar wird.
Dieser Handler wurde ursprünglich für die "Internationalisierung" (i18n) einer bestehenden (einsprachigen) Applikation verwendet. Dazu wurden in OnPageLoaded() alle Zeichenketten auf einer Seite *zur Laufzeit* in die vom Benutzer gewählte Sprache übersetzt (anhand einer String-Tabelle). Ein Bespiel dazu finden Sie im 'Internationalisierungs-Demo' (programs/script_demos/i18nDemo.cvt).

1.2 4.11.1.2 OnPageEnter(int page_nr, string page_name)

Dieser (ebenfalls optionale) Event-Handler wird beim ersten Aufruf einer Seite aufgerufen (kurz vor dem Sichtbar-Werden, und kurz *nach* dem Laden der Seitendefinition aus dem Flash (-> OnePageLoaded).

1.3 4.11.1.3 OnPageQuit(int page_nr, string page_name)

Dieser (optionale) Event-Handler wird -wenn im Script vorhanden- unmittelbar vor dem *Verlassen* einer Anzeigeseite aufgerufen. Die Funktionsargumente 'page_nr' und 'page_name' beziehen sich auf die beim Aufruf noch aktuelle, "alte" Seite. Kurz danach wird dann der OnePageEnter-Handler für die "neue" Seite aufgerufen.

1.4 4.11.1.4 OnPageUpdate( )

Dieser optionale Handler wird bei jeder Seiten-Aktualisierung *mehrfach* aufgerufen. Zweck: Beim Aufbau einer Display-Seite im Framebuffer können per OnPageUpdate() noch weitere Grafikelemente gezeichnet werden, die mit den Standard-Anzeige-Elementen nicht zu realisieren wären (z.B. bewegte "Sprite"-Grafik im 'MacPan'-Demo).
Details standen bei der Erstellung dieser Beschreibung noch nicht fest. Verwenden Sie bei Interesse bitte die englische Variante dieser Beschreibung.

2 4.11.2 Verarbeitung von Ereignissen, die von UPT-Anzeige-Elementen ausgelöst wurden ('Control Events')

Wurde ein Ereignis *nicht* auf dem 'untersten Level' abgefangen (siehe vorhergehendes Kapitel), dann wird es vom System i.A. an den nächsthöheren Verarbeitungsschritt weitergegeben. Dieser nächsthöhere Level hat oftmals mit den auf einer 'programmierbaren' Anzeigeseite befindlichen, für

den Bediener sichtbaren Bedienelementen (englisch: 'Controls') zu tun. Dazu gehören (beim UPT und ähnlichen Geräten) z.B. die programmierbaren 'Buttons', Menüzeilen, Editierfelder auf der aktuellen Anzeigeseite.

Ereignisse, die mit diesen 'sichtbaren Steuerelementen' (engl. Controls) zu tun haben, können in der Script-Sprache mit dem folgenden Event-Handler verarbeitet werden:

```
func OnControlEvent( int event, int controlID, int param1, int param2 ) // event from a 'visible control element'
```

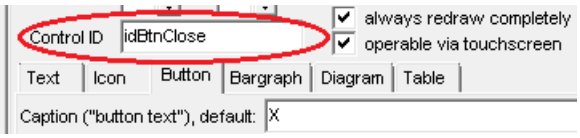
Dieser Handler kann für verschiedene Typen von Ereignissen aufgerufen werden. Das erste Funktionsargument teilt dem Handler mit, *welches Ereignis* zur Verarbeitung ansteht :

`event` : Typ des Ereignisses. Dies kann eine der folgenden, in der Script-Sprache fest eingebauten Konstanten sein (Integer):

`evClick` : "das Element wurde angeklickt, oder die ENTER-Taste wurde betätigt als es den Fokus hatte"
`evPenDown` : "der Touchscreen wurde gedrückt, und die Koordinate lag innerhalb der Fläche dieses Elements"
`evPenMove` : "Stift oder Finger wurde auf dem Touchscreen bewegt, Koordinate weiterhin innerhalb des Elements"
`evPenUp` : "Stift oder Finger wurde vom Touchscreen abgehoben" (nachdem die Koordinate vorher im Bereich des Elements lag)
`evKey` : "Ein Tastencode wird an das Element gesendet, welches zur Zeit den Eingabefokus hat"
`evBeginEdit`: "Ein Editierfeld wurde grade in den Modus 'Editieren' geschaltet". Zweck: Siehe display.EditValueMin / Max.
`evEndEdit` : "Das Editieren wurde soeben beendet". Zweck und Details hier.
Dieses Ereignis wird auch an den Handler für die virtuelle Tastatur (OnVirtualKeyboardEvent) übergeben.

`controlID` : Mit diesem Integer-Wert wird das Steuerelement adressiert, für das das Ereignis ausgelöst wurde.

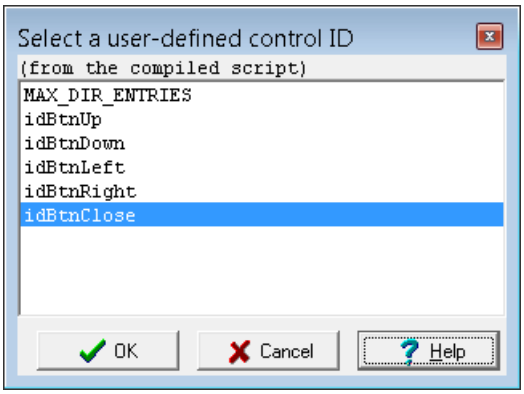
Als als Control-ID sollten nur benutzerdefinierte Konstanten mit 'sprechenden' Namen verwendet werden - KEINE ZAHLEN !
Der Wert wird (als symbolische Konstante mit maximal 20 Zeichen) im Feld namens 'Control ID' auf der Registerkarte display line properties bei der Definition einer UPT-Anzeigeseite im Programmiertool eingegeben.



Definition eines Buttons mit Control-ID

Hat ein Steuerelement keinen Control-ID (d.h. das oben erwähnte Feld ist leer), dann kann es **kein** Control-Event auslösen.
Es wird empfohlen, als Control-ID 'sprechende' Konstanten-Namen mit dem Vorsatz 'id' zu verwenden, z.B. idBtnUp, idBtnDown, idBtnLeft. Bei der Definition eines Steuerelements auf

der Registerkarte 'Seite #X' kann der dem Element zugeordnete Identifier dann per Doppelklick in das Eingabefeld 'Control ID' aus einer Liste ausgewählt werden, z.B.:



Liste mit selbstdefinierten Script-Konstanten,
nach Doppelklick in das Feld 'Control ID' im Programmiertool.

Der Control-ID kann auch zur Adressierung des entsprechenden Display-Elements verwendet werden, z.B.
display.elem_by_id[controlID].xyz (xyz=Komponente des Anzeige-Elements).

param1 : Erster Message-Parameter. Die Bedeutung dieses Parameters hängt vom Typ des Ereignisses (event) ab :

Bei event= **evPenDown**, **evPenMove** oder **evPenUp** enthalten param1 und param2 die *Grafik*-Koordinate (param1 = 'X', param2 = 'Y')
im 'Client-Koordinatensystem' (x=0, y=0 ist die linke obere Ecke des Steuerelementes).
Um diese Pixelkoordinaten im Touchscreen-Event-Handler ggf. in "Text-Koordinaten" umzuwandeln,
dividieren Sie X (gemessen in Pixel) durch tscreen.cell_width,
und Y (gemessen in Pixel) durch tscreen.cell_height.

Bei event=**evKey** enthält param1 den Tastencode der Taste, die an das Steuerelement gesendet wurde.

Bei event=**evClick**, **evBeginEdit** oder **evEndEdit** enthält param1 den *Index* des Elements, welches das Event ausgelöst hat.
So können auch *mehrere* gleichartige Steuerelemente mit *identischem Control-ID* voneinander unterschieden werden. Zum Zugriff auf das auslösende Element eignet sich in diesem Fall die Funktion display.elem[param1].xyz (darin ist xyz die Komponente des Anzeige-Elements).

param2 : Zweiter Message-Parameter. Bedeutung: s.O.

Ein einfaches Beispiel zur Verwendung des Handlers 'OnControlEvent' finden Sie in der Applikation 'EventTest' .
Ein Beispiel für Fortgeschrittene finden Sie im 'selbstdefinierten Popup-Menü', in dem ein Text-Panel mit entsprechenden Touchscreen-Event-Handlern als vom Bediener aufklappbares Menü (bzw Auswahlliste) verwendet wird.

Siehe auch:
Vom Anzeige-Element 'Tabelle' (table) ausgelöste Ereignisse,
vom Anzeige-Element 'Diagramm' (diagram) ausgelöste Ereignisse,
von der virtuellen Tastatur ausgelöste Ereignisse,
Aktivieren von Event-Handlern nach der Initialisierung per Kommando 'init_done'.

3 4.11.3 Timer-Events

Wie bereits im Kapitel 'Weitere Funktionen und Kommandos' beschrieben, wird mit dem Script-Kommando **setTimer** ein Timer gestartet.
Als drittes (optionales) Funktionsargument kann dabei der Name eines Timer-Event-Handlers angegeben werden, z.B.:

```
      var
        tTimer timer1;  // Deklaration einer Timer-Instanz als globale Variable
des Typs 'tTimer'
      endvar;
         ...
      setTimer( timer1, 100, addr(OnTimer1) ); // Starte 'timer1'
         // mit einem Intervall von 100 Millisekunden für periodischen Aufruf
des Handlers 'OnTimer1'
         ...
```

Der Funktionsname des Timer-Event-Handlers ist prinzipiell frei wählbar. Wir empfehlen, wie in allen Event-Handlern, dem Funktionsnamen die Silbe 'On' voranzustellen. Bei mehreren Timer-Event-Handlern innerhalb eines Scripts müssen natürlich unterschiedliche Funktionsnamen (und unterschiedliche Timer-Namen im Variablendeklarationsteil) verwendet werden.

In der Script-Sprache arbeiten Timer *immer* periodisch (solange sie nicht *gestoppt* werden). Nach jedem Ablauf des Timers wird das Flag 'expired' (engl. 'abgelaufen') in der Timer-Variablen gesetzt. Wenn (wie im obigen Beispiel) der Name eines Timer-Event-Handlers angegeben wurde, wird ***beim Setzen*** des **expired**-Flags auch der dazugehörige Timer-Event-Handler aufgerufen.
Beim Aufruf des Timer-Event-Handlers wird die Adresse des dafür verwendeten Timers übergeben. Die per Timer aufgerufene Funktion muss eine dazu passende Argumentenliste enthalten (tTimer ptr ..), wie im folgenden Beispiel für einen Timer-Event-Handler:

```
      func OnTimer1( tTimer ptr pMyTimer )  // Periodisch aufgerufener Timer-
Event-Handler
         local tCANmsg msg;  // Deklaration einer CAN-Message als lokale
Variable
         msg.id := 0x334;    // zu sendenden CAN-Message-Identifier setzen
         msg.len:= 8;        // CAN data length code setzen (max. 8 Bytes = 2
doublewords)
```

```
        msg.dw[0] := 0x11223344; // die ersten vier Bytes als 'double-word' im
CAN-Datenfeld setzen
        msg.dw[1] := 0x55667788; // die letzten vier Bytes als 'double-word'
setzen
        can_transmit( msg ); // CAN-Message senden
        return TRUE; // TRUE = 'Ereignis wurde verarbeitet' (FALSE würde kein
neues Event auslösen)
      endfunc; // end OnTimer1
```

Kehrt der Timer-Event-Handler mit dem Rückgabewert 'TRUE' zurück, wird das 'expired'-Flag des entsprechenden Timers gelöscht. Beim nächsten Ablauf des Timers wird das 'expired'-Flag dann wieder (vom System) gesetzt, was zum erneuten Aufruf des Timer-Event-Handlers führt.

Kehrt der Timer-Event-Handler dagegen mit dem Rückgabewert 'FALSE' zurück (d.h. "Event wurde NICHT verarbeitet"), dann bleibt das 'expired'-Flag in der Timer-Struktur gesetzt. Dadurch ist beim beim nächsten Ablauf des Timers aus Sicht des Laufzeitsystems dann keine 'steigende Flanke' mehr gegeben, was dazu führt, daß der Timer-Event-Handler *nicht* erneut aufgerufen wird (weil das vorherige Ereignis aus Sicht des Systems noch nicht verarbeitet wurde).

Weitere Beispiele für Timer-Events (im Script) finden Sie in Kapitel 5.
Siehe auch: wait_ms(0) (zum sofortigen Abarbeiten aller momentan anstehenden Timer-Events).

4 4.11.4 CAN-Empfangs-Handler

Zusätzlich zur im Kapitel CAN beschriebenen Methode zum Empfang von CAN-Telegrammen per Polling (zyklischer Aufruf von can_receive in der Hauptschleife) kann für bestimmte Telegramme ein CAN- bzw. LIN-Empfangs-Handler *in der Script-Sprache* implementiert werden.
Damit kann die Reaktionszeit für 'wichtige' CAN-Telegramme, auf die mit hoher Priorität besonders schnell reagiert werden muss, deutlich verkürzt werden.

Implementieren Sie dazu wie im folgenden Beispiel eine Handler-Funktion, und registrieren Sie diese (zusammen mit dem Message-ID oder -Filter) mit der Prozedur
can_add_id(<CAN-ID>, <Name des Handlers>) bzw **CAN.add_filter**(<Filter>, <Maske>, <Name des Handlers>).

```
      can_add_id( 0x0000123, addr(CAN_Handler_ID123) ); // register CAN-ID and a
receive-handler
      can_add_id( 0x0000124, addr(CAN_Handler_ID123) );  // another CAN-ID for
the same handler

       ...
```

Soll der Handler immer dann aufgerufen werden, wenn ein bestimmtes per CAN-Datenbank (*.dbc) definiertes Signal empfangen wurde, dann kann der CAN-Message-Identifier wie im folgenden Beispiel (mit "Oeldruck") mit der Funktion display.GetVarDefiniton() ermittelt werden. Statt des numerischen CAN-Message-Identifiers muss dann nur der Name des zu empfangenen Signals (bzw. der damit verknüpften Display-Variable) bekannt sein:

```
      var
```

```
        tDisplayVarDef ptr pVarDef;  // pointer to the definition of a display-
variable
      endvar;

       ...

      pVarDef := display.GetVarDefinition( "Oeldruck" ); // get database entry
for "Oeldruck"
      can_add_id( pVarDef.CAN_Msg_ID, addr(CAN_Handler_Oeldruck) ); // register
handler for "Oeldruck"

       ...
```

Ein CAN-Message-Handler braucht nicht auf einen einzigen Message-Identifier beschränkt zu sein. Verwenden Sie z.B. einen select-case-Block um abhängig vom ID eine 'Sonderbehandlung' zu realisieren:

```
  func CAN_Handler_ID123( tCANmsg ptr pRcvdCANmsg )
     // CAN-Receive-Handler for certain 'important' CAN message identifiers.
     // Interrupts normal processing, and must return to the caller a.s.a.p. !
     select( pRcvdCANmsg.id )  // Take a look at the CAN message identifier...
       case 0x123:  // message ID 0x123 (hex)
           CAN.DecodeMessage( pRcvdCANmsg ); // per CAN übertragene Variablen
sofort aktualisieren
           if( display.ValueFromMessageID123 > 123.0 ) then
              IllegalValueCounter := IllegalValueCounter + 1;
           endif;
           return TRUE; // 'handled' here; do NOT place this message in the CAN-
RX-Fifo
       case 0x124:  // message ID 0x124 (hex)
       case 0x125:  // message ID 0x125 (hex)
           return TRUE; // 'handled' here; do NOT place this message in the CAN-
RX-Fifo
     endselect;
     return FALSE; // did NOT handle this message here; let the system place it
in the CAN-RX-Fifo
   endfunc; // end CAN_Handler_ID123
```

Der so definierte CAN-Empfangs-Handler wird dann kurz nach dem Empfang eines entsprechenden Telegramms aufgerufen, wozu die normale Abarbeitung des Scripts (und anderer Funktionen des programmierbaren Terminals) kurz *unterbrochen* wird. Diese Unterbrechung darf aus technischen Gründen nur einige Millisekunden dauern (Details im gelb hinterlegten Warnhinweis im Kapitel 'Event-Handling in der Script-Sprache'). Der Aufruf des Handlers erfolgt noch *vor dem Decodieren der im Telegramm eventuell vorhandenen 'Signale' für das Display* (daher muss ggf. CAN.DecodeMessage aus dem Handler aufgerufen werden; siehe Hinweise am Ende dieses Abschnitts).
Während der Handler ausgeführt wird, kann das Gerät keine anderen Aufgaben erledigen ! Kehrt der CAN-Empfangs-Handler nicht 'freiwillig' innerhalb von 500 Millisekunden zum Aufrufer zurück, so wird die Ausführung des Handlers zunächst unterbrochen, damit das Gerät bedienbar bleibt. Als Rückgabewert wird dann der Wert 0 (Null) angenommen, was dazu führt, dass das

empfangene CAN-Telegramm vom Default-Handler des Systems (Firmware) verarbeitet wird.

Bedeutung des vom CAN-Empfangs-Handler zurückgelieferten Rückgabewertes ('Return-Wert') :

- **FALSE** (0) : Der Handler hat dieses Telegramm *nicht* verarbeitet.
  Der Default-Handler (Firmware) wird das Telegramm daraufhin in den CAN-Empfangs-FIFO eintragen,
  aus der es per Polling (zyklischer Aufruf von can_receive aus der Hauptschleife des Scripts) ausgelesen werden kann.
- **TRUE** (1) : Der Handler hat dieses Telegramm verarbeitet.
  Das empfangene Telegramm soll *nicht* in den RX-FIFO für die 'normale' Weiterverarbeitung von CAN-Signalen eingetragen werden.

Ein *komplettes* Beispiel mit CAN-Empfangs-Handler finden Sie in der Applikation ScriptTest3.cvt .

Hinweise zum CAN-Empfangs-Handler:

- Da der CAN-Empfangs-Handler vom System *so frühzeitig wie möglich* aufgerufen wird, wurden zu dem Zeitpunkt die im empfangenen CAN-Telegramm enthaltenen Signale *noch nicht* für das Display decodiert.
- Liefert der CAN-Empfangs-Handler den Return-Wert TRUE, dann signalisiert er damit, dass das empfangene Telegramm bereits 'vollständig' im Handler selbst verarbeitet wurde. Wie bereits erwähnt, wird das CAN-Telegramm dann *nicht* in den Empfangs-FIFO für die Anzeige eingetragen (zu diesem Zweck *muss* der CAN-Empfangs-Handler auch *vor* der Decodierung der im Telegramm enthaltenen Signale aufgerufen werden).
- Um im CAN-Empfangs-Handler die im empfangenen Telegramm enthaltenen Signalen zu verarbeiten, können die Signale direkt im Handler decodiert werden, z.B.:
  ```
  Drehzahl := pRcvdCANmsg.bitfield[0,16] * 0.3333; // empfangenes Signal decodieren
  ```
  (darin ist 'Drehzahl' eine im Script deklarierte Variable)
- ... oder (Alternative): Der CAN-Empfangs-Handler kann mit dem folgenden Befehl das sofortige Decodieren des empfangenen Telegramms veranlassen (inkl. aller darin enthaltenen Signale):
  ```
  CAN.DecodeMessage( pRcvdCANmsg ); // alle im empfangenen Telegramm enthaltenen Signale decodieren
  ```
  (die in 'pRcvdCANmsg' enthaltenen Signale stehen danach in den per DBC-Datei importierten Display-Variablen).
- Wurde der Empfang von CAN-Telegrammen mit Hilfe des CAN-Logfile-Players 'simuliert', dann wird der Player automatisch gestoppt sobald das Script durch einen Haltepunkt im Debugger gestoppt wird.
  Das letzte per Player 'abgespielte' CAN-Telegramm, welches z.B. zum Aufruf des Handlers führte, kann danach im Programmiertool auf der Registerkarte 'Fehler und Meldungen' untersucht werden.

- Stammen die empfangenen CAN-Telegrammen vom programmierbaren CAN-Simulator, dann kann beim Auflaufen des Scripts auf einen Breakpoint auch der *CAN-Simulator* pausiert werden. Diese Option war beim Entwickeln von Script-Applikationen, z.B. bei der Implementierung eigener Übertragungsprotokolle, sehr hilfreich.
- *Ein* einzelnes empfangenes Telegramm kann nacheinander zum Aufruf *mehrerer* registrierter CAN-Message-Handler führen. Beispiel: Ein Telegramm mit dem ID 0x123 könnte sowohl zum Aufruf eines per can_add_id(0x123,...) registrierten Handlers, und zusätzlich zum Aufruf des per CAN.add_filter(0x000, 0x000, ...) registrierten Handlers für "alle" Message-IDs führen.
  Dabei ist es sogar möglich, *denselben* Handler mehrfach zu registrieren. Entsprechend würde dieser *mehrfach registrierte* Handler dann auch bei passenden CAN-Messag-IDs *mehrfach* aufgerufen, obwohl vom CAN-Bus nur *ein* Telegramm empfangen wurde.
  Dabei werden zuerst die 'individuell' per can_add_id() registrierten Handler aufgerufen (bis *einer* dieser Handler den Wert TRUE liefert, d.h. "ich habe das empfangene Telegramm verarbeitet"), und (wenn keiner der per can_add_id registrierten Handler den Wert TRUE lieferte) zuletzt der per CAN.add_filter() registrierte Handler.

5 4.11.5 Ereignisse von der virtuellen Tastatur

Ereignisse, die von der virtuellen Tastatur ausgelöst werden, können ähnlich wie Ereignisse von *benutzerdefinierten* Anzeige-Elementen (OnControlEvent) auch per Script 'abgefangen' oder verarbeitet werden. Dazu kann die folgende Funktion im Script implementiert werden (bitte auch hier die genaue Schreibweise, mit Groß- und Kleinbuchstaben, beachten):

```
func OnVirtualKeyboardEvent( int event, int param1, int param2 ) // event from the 'virtual keyboard'
```

Im Gegensatz zum OnControlEvent-Handler fehlt hier der Parameter 'controlID', da von der virtuellen Tastatur immer nur *eine Instanz* existieren kann.
Die Bedeutung der Parameter 'event', 'param1' und 'param2' entnehmen Sie bitte der Beschreibung von OnControlEvent.
Vom Event-Handler für die virtuelle Tastatur bis dato (2020-06-07) unterstützte Ereignisse (erstes Argument, Parameter 'event'):

evChar
: Betätigung einer (virtuellen?) Taste mit dem Code (ASCII) in param1.

evBeginEdit
: Virtuelle Tastatur wurde soeben in den Modus 'Editieren' geschaltet.

evEndEdit
: Die Eingabe im Editierfeld der virtuellen Tastatur wurde soeben abgeschlossen.
Von besonderem Interesse im Zusammenhang mit dem Kommando vkey.connect:
Der editierte Text kann nun per vkey.text ausgelesen werden.
Parameter:
iparam1 : Zeilennummer ('row' aus dem Aufruf von vkey.connect),
iparam2 : Spaltennummer ('col' aus dem Aufruf von vkey.connect).

Wird OnVirtualKeyboardEvent() bei diesem Event mit dem Return-Wert TRUE verlassen, d.h. "das Ereignis wurde *vom Script* verarbeitet bzw. abgefangen, dann wird das automatische Zurückkopieren des Textes aus dem Editierfeld in die bei vkey.connect() angegebene Variable unterbunden.

evClick
Klick 'irgendwo in die Client-Fläche' der virtuellen Tastatur,
je nach Hardware auch Auswahl per Drehknopf oder 'Enter'-Taste.

Beispiele zur Nutzung der virtuellen Tastatur (z.T. mit Event-Handlern) finden Sie in den Applikationen script_demos/VKeyTest.cvt und script_demos/TableTest.cvt .

Siehe auch:
Befehle zum Steuern der virtuellen Tastatur per Script

6 4.11.6 Erweiterte Funktionen zum Message-Handling

Zum Zeitpunkt der Erstellung dieses Dokuments waren erweiterte Message-Funktionen zwar geplant, aber **noch nicht implementiert**.

Details zu den *geplanten* Funktionen (zum Senden und Empfangen von System-Messages) finden Sie z.Z. nur in der englischen Variante dieser Datei.

4.12 Präprozessor-Direktiven

1 4.12.1 #pragma

Mit der #pragma-Direktive werden optionale Anweisungen an den Script-Compiler übergeben. Bislang implementiert sind:

#pragma strict : *Nur* deklarierte Variablen verwenden.

2 4.12.2 #include

Diese Präprozessor-Direktive dient zum Einfügen einer Include-Datei in den Script-Quelltext.

Syntax:
#include "<Dateiname ohne Pfad>"

Da das Zielsystem (z.B. MKT-View) keinen Zugriff auf die auf der Festplatte des PCs gespeicherten Include-Dateien hat, wird der Quelltext aus der angegebenen Include-Datei bereits im Programmiertool *in die geladene Applikation kopiert*, und anschließend als Teil des Scripts *in der CVT- bzw. UPT-Datei abgespeichert* bzw bei der Übertragung der Applikation per CAN übertragen. So wird gewährleistet, dass das Script auch ohne Speicherkarte komplett im Zielsystem zur Verfügung steht.

Bei jedem Laden und Übersetzen (Compilieren) des Scripts *im Programmiertool* wird auch der Inhalt der Include-Datei neu aus der Datei geladen, und der eventuell bereits in der Applikation vorhandene Inhalt (aus einer früher geladenen Include-Datei) überschrieben.

Der aus der Include-Datei geladene Text ist aus technischen Gründen auch im Script-Editor / Debugger sichtbar.
Editieren des aus der Include-Datei geladenen Textes ist zwecklos ! Änderungen müssen *in der Include-Datei* durchgeführt werden, nicht im Script *in dem die Include-Datei geladen wurde*. Sinn und Zweck von Include-Dateien ist, dass diese mehrfach verwendet werden können, aber Änderungen nur zentral an einer Stelle (nämlich in der Include-Datei selbst) durchgeführt werden.

Der im Programmiertool verwendete 'Rich Text'-Editor ist leider nicht in der Lage, die 'inkludierten' Abschnitte gegen das Editieren zu schützen. Als Ersatz wird der aus der Include-Datei geladene Abschnitt farblich markiert, um ihn von 'normalem' Quelltext zu unterscheiden. Die Farbe des Hintergrunds zeigt auch an, ob die Include-Datei beim letzten Compilieren erfolgreich geladen wurde:

Gelb
: Die Include-Datei konnte vom Compiler nicht geladen werden, der gelb markierte Text im Editor stammt aus einer *möglicherweise veralteten* Datei (die sich irgendwo auf einem anderen PC befindet, oder außerhalb des auf der Registerkarte 'Verzeichnisse' eingestellten Include-Pfades).

Hellgrau
: Die Include-Datei wurde beim Compilieren des Scripts erfolgreich geladen.

Um eigene Include-Dateien zu erzeugen, können Teile aus existierenden Scripten herauskopiert werden und mit einem Text-Editor wie z.B. Notepad++ **als einfacher ANSI-Text** im einstellbaren Include-Verzeichnis abgespeichert werden. Als Dateinamenserweiterung empfehlen wir .inc (für 'include'), für den Script-Compiler spielt die Erweiterung aber (noch) keine Rolle.

Beim oben erwähnten "Neu-Laden" der Include-Datei markiert der Script-Compiler den Anfang und das Ende des inkludierten Textes mit zwei speziellen Quelltext-Zeilen, die *ebenfalls nicht im Script-Editor editiert werden dürfen* (da der Inhalt der 'alten' Include-Datei beim nächsten Kompilieren sonst nicht wieder 'entladen' werden könnte):

```
##begin_include "Test.inc" date=2016-08-04_16:24:10 // DO NOT EDIT THIS PART !
```

Diese Zeile steht direkt **vor** der ersten, aus der Include-Datei geladenen Zeile.
Zu Kontrollzwecken enthält diese Zeile den Namen und das letzte Änderungsdatum der inkludierten Datei.

```
##end_include "Test.inc"
```

Diese Zeile folgt direkt **nach** der letzten aus der Include-Datei geladenen Zeile.
Sie markiert (auch für den Compiler selbst) das Ende des aus der Include-Datei geladenen Abschnitts.
Zu Kontrollzwecken enthält auch diese Zeile den Namen der inkludierten Datei.

Ein einfaches Beipiel zur Verwendung von Include-Dateien finden sie hier.

4.13 Schlüsselwörter

Hinweis: Die folgende Liste ist unvollständig ! Sie wird laufend erweitert. Kommandos und Schlüsselwörter ohne Hyperlink sind möglicherweise noch nicht im Compiler, oder/und in der Laufzeitumgebung enthalten.
Siehe auch: Kurzreferenz, Operatoren, Konstanten .

Schlüsselwort	Parameterliste bzw. Syntax	Rück-gabe-wert	Beschreibung
~~abs~~			-> Math.abs() Liefert den Absc bzw (mit zwei Argumenten) di
addr	(variable)	pointer	Liefert die *Adresse* der Variable
append	(dest,source[,index])	-	Anhängen eines Strings (source)
AND	A AND B	int	boolean AND operator, same as The result is 1 (one, TRUE) if **b**
anytype			Datentyp für eine Variable, die s Zum Ermitteln des aktuell verw
~~atan~~			->Math.atan2 arctangent functio
atof	(string)	float	"ascii to float". Wandelt eine Ze
atoi	(string)	int	"ascii to integer". Converts a de
bit_and	A bit_and B	int	Performs a bit-wise AND operat For example, 5 (101 binary) **bit**
bit_or	A **bit_or** B	int	Performs a bit-wise OR combina For example, 5 (101 binary) **bit**
bit_not	(unary operator)	int	Bitweises Invertieren des Werte Auch als Einerkomplement beka Zur Kompatibilität mit "C" kann Bitweise InvNOT operator is of
BytesToFloat	(exp, m2, m1, m0)	float	Wandelt eine 4-Byte-Sequenz in Das erste 8-bit Argument (exp) Das letzte 8-bit Argument (m0)
BinaryToFloat	(32-bit 'DWORD')	float	Wie BytesToFloat, hier werden Wird als Argument eine 32-Bit- Ein Test für diese Funktion find
BytesToDouble	(exp1,exp0,m5..m0)	double	Wandelt eine 8-Byte-Sequenz in Das erste 8-bit Argument (exp1) Das letzte 8-bit Argument (m0)
can_receive		int	Tries to read the next received C When successful, the message is Otherwise (empty FIFO), can_r

CAN. (..)			Präfix und Namensraum für and
case	integer CONSTANT		part of a select .. case .. else .. en
chr	(integer code)		Converts an integer character co
cls			clears the text-mode screen (her
cop. (..)			Präfix (und Namensraum) für C Nur für Geräte mit CANopen.
~~cos~~			->Math.cos cosine function
~~dec~~			(reserved for an optimized 'decr
display.xyz			Commands and functions contro
dtInteger, etc			Datentypen-Codes, die in Verbi
endif			Ende eines **if .. then .. [elif ...] e**
endwhile			Ende einer **while** - Schleife
endselect			Ende einer select .. case - Liste
endproc			markiert das Ende einer selbstde
endfunc			markiert das Ende einer selbstde
elif			Teil eines **if .. then .. elif .. [elif** ("else if", wodurch ein zusätlic
else			Teil eines **if .. then .. else .. end** oder einer select .. case .. **else** ..
EXOR	A EXOR B (binary operator)	int	This operator performs a bitwise Often used to toggle (invert) one For example, 5 (0101 binary) E Notes: • The "^" operator is NOT (A ^ B is reserved for " • There is no BOOLEAN as the 'not equal' operato • The bitwise EXOR opera
file.			file I/O functions
float			data type for floating point valu
FloatToBinary	(32-bit 'float')	32-bit int (binär)	Umkehrfunktion zu 'BinaryToF Fliesskommawertes als 32-Bit-I Das Bitmuster selbst wird dabei von 'float' auf 'int' gesetzt. Die Funktion kann z.B. verwend nichtflüchtig gespeichert werden bei der automatischen Typumwa
for			begins a for .. to .. step .. next lo
ftoa	(value, nDigitsBeforeDot, nDigitsAfterDot)		'float to ASCII'.

			Wandelt einen Fliesskomma-We mit der spezifizierten Anzahl vo
func			marks the begin of a user-define
~~GOTO~~			stoneage jump instruction, try to
~~GOSUB~~			old subroutine calls, try to avoi
gotoxy	(x,y)		sets the text output cursor in the
hex alias HexString	(value, nDigits)		Wandelt einen Integer-Wert in e mit der im zweiten Argument sp
BinaryString	(value, nDigits)		Wandelt einen Integer-Wert in e mit der im zweiten Argument sp
if	(condition)		statement begins an IF .. THEN
in			defines the following argument
int			integer (data type; 32 bit signed
isin	(argument:0...1023)		Schnelle Integer-Sinus-Funktion Ausgangsbereich -32767 to +32
itoa	(value, nDigits)		'integer to ASCII'. Wandelt einen Integer-Wert (Ga mit der im zweiten Argument sp
limit	(variable,min_value,max_value)	-	Begrenzt den in einer Variablen
local			definiert lokale Variablen (die n
~~log~~			->Math.log natural logarithm
Math.xyz	Math.pow(x,y); Math.sin(x); ..		Mathematische Funktionen wie
next			ends any for .. to .. step .. next lo
not	(unary operator)		boolscher 'NOT' operator (Nega Ist der Eingang 0 (Null, FALSE andernfalls (mit Eingang unglei
OR			boolean OR operator, same as th The result is 1 (one, TRUE) if **a**
out			Kennzeichnet das folgende Argu in der Parameterliste einer Proze
print			Prints values into a multi-line "t
ptr			Keyword for a typeless or fixed-
cPI			constant value "PI" (3.14159....)
proc			Marks the begin of a user-define
ramdom	(N)		erzeugt eine Zufallszahl zwische
return	(Return-Wert)		Rückkehr von einer benutzerdefi

repeat			begins a repeat..until loop. This
REM			begins a remark in BASIC. Bett
rgb	(red, green, blue)		Composes a colour from red, gr
select	(integer expression)		begins a select .. case .. else .. en
setcolor			Sets the foreground- and backgr
~~sin~~			->Math.sin sine function
SHL	(binary operator)		Bitwise shift left. Example N :=
SHR	(binary operator)		Bitwise shift right. Example N Note: SHR is considered an 'arit and positive numbers remain po 0x80000000 SHR 31 gives the r
~~sqrt~~			->Math.sqrt Quadratwurzel
step			defines the counter's stepwidth i
stop			Stops execution of the script. Us
string			data type for a 'string' of charact
system	.component-name		Unit-Name für systemnahe Vari
~~TAN~~			
tCANmsg			Datentyp für ein CAN-Telegram
time			Funktionen zum Konvertieren v
tMessage			Datentyp für eine System-Messa im Script verarbeitet werden kön
to			defines the counter's end value i
trace	.print, .enable, ..		Steuerung der Trace History
tscreen	.component-name		text-screen buffer object
tScreenCell			data type name for a 'text screen
typedef			defines a new data type (usually
typeof			Liefert den *momentanen* Datenty die z.B. als 'anytype' deklariert Das Ergebnis ist i.A. eine Daten
until	(end criterion)		ends a repeat..until loop
wait_ms	(milliseconds)		Wartet die angegebene Zahl vor mit der nächsten Anweisung for
while			

Durchgestrichene Schlüsselwörter sind nicht (bzw. nicht mehr, oder noch nicht) in der Script-Sprache enthalten - sie stammen aus der privaten Entwicklung des Autors, die stark an die Programmiersprache BASIC (QBASIC) angelehnt war.

Beachten Sie, daß aus historischen Gründen (ähnlich wie in PASCAL) Groß/Kleinschreibung bei fast allen Schlüsselwörtern ignoriert wird. Siehe Hinweise zur Groß/Kleinschreibung in den Empfehlungen zum Programmierstil .

Siehe auch:

- Übersicht der in der Script-Sprache (fest) integrierten Konstanten
- Übersicht aller fest integrierten Datentypen
- Übersicht der Operatoren (z.T. Namen (Schlüsselwörter), z.T. Sonderzeichen)

4.14 Fehlermeldungen

Die folgende Liste ist nicht vollständig. Die meisten beim Übersetzen oder während der Laufzeit (im Simulator) auftretenden Fehlermeldungen und Warnungen sprechen glücklicherweise 'für sich selbst'.
Sie werden im Programmiertool auf der Registerkarte Fehler & Meldungen angezeigt.

- syntax error
  Beim Übersetzen trat ein Syntax-Fehler auf, der vom Compiler nicht genauer bestimmt werden kann.
- missing argument
  Die Parameterliste beim Aufruf einer Funktion oder Prozedur enthält weniger Argumente als erwartet.
- Warning: Possibly incompatible (argument-) type near line NNN (2nd argument: expecting 'int', got 'array of int')
  Datentypen beim Aufruf einer Funktion oder Prozedur sind 'verdächtig inkompatibel'.
  In diesem Beispiel wurde ein komplettes Array übergeben, aber die aufgerufene Funktion erwartete im zweiten Argument nur einen Integer-Wert.
  Ähnliche Warnungen können auch in Kombination mit anderen Datentypen auftreten.
  Überprüfen Sie in solchen Fällen den von der aufgerufenen Funktion (oder Prozedur) erwarteten Datentyp mit dem beim Aufruf übergebenen Typ.
  Bei trivialen Typumwandlungen (z.B. von Byte nach Integer oder Integer nach Float) erhalten Sie diese Warnung nicht. Solche Konversionen werden vom Laufzeitsystem automatisch durchgeführt.
- missing left parenthesis
  Fehlende linke Klammer, evtl. komplett fehlende Parameterliste.
- missing right parenthesis
  Fehlende rechte Klammer ("Klammer zu"). Im Gegensatz zum Display-Interpreter sind fehlende schließende Klammern beim Compiler ein "k.o."-Kriterium.
- missing LEFT square bracket ([ )
- missing RIGHT square bracket ( ])
  Fehlende eckige Klammer, meistens an einer Stelle an der ein Array-Index erwartet wird.
- missing operand
- type conflict
- division by zero
- illegal value
- function unknown
- function permanently unavailable
- function temporarily unavailable
- illegal array index or similar
- missing component

- unknown component
- function failed
- bad array subscript
- illegal pointer or reference
- comma or closing parenthesis expected
- expecting a semicolon
- expecting a comma
- name expected
- var-name expected
- expecting an assignment
- expecting a data type
- expecting an integer value
- undefined variable
- out of memory
- structure or block too large
- illegal channel number
- label not found
- return without gosub
- call stack overflow
- call stack underflow
- call stack corrupted
  Diese Fehler können auftreten, wenn eine anwenderdefinierte Funktion zum Aufrufer zurückkehren soll, auf dem Stack aber keine geeignete Rücksprung-Adresse abgelegt ist (die Runtime-Library kann 'Programm-Adressen' von anderen Datentypen auf dem Stack unterscheiden).
- name or symbol too long
  Namen von Variables, Funktions, Prozeduren, Datentypen, Konstanten, usw. sind auf maximal 20 Zeichen begrenzt.
- subscript or indirection too long
  Bezieht sich auf Arrays oder/und ineinander geschaltelte Strukturen.
  Ist z.B. A ein eindimensionales array, dann ist A[1][2] ein unzulässiger Ausdruck (in diesem Fall "zu viele Indizes").
- bad input
  Eine Eingabefunktion (oder ein String-Parser) konnte die Eingabe nicht verarbeiten (z.B. konnte eine Zeichenkette nicht in einen numerischen Wert verwandelt werden).

- 'for' without 'next'
  Der Compiler fand ein 'for', ohne dazu passendes 'next'.
- 'next' without 'for'
  Der Compiler fand ein 'next', ohne dazu passendes 'for'.
- 'else' without 'if'
- 'endif' without 'if'
- 'case' without 'select'
- 'endselect' without 'select'
- 'endwhile' without 'while'
- 'until' without 'repeat'
- no loop to exit from
- only callable from SCRIPT
- simple variables only
- variable or element is READ-ONLY
- unknown script command
- function not implemented yet
- RPN eval stack overflow
- RPN eval stack underflow
- illegal code pointer
- illegal sub-token after opcode
- cannot use as LVALUE (in assignment)
  Das Element auf der linken Seite eines Zuweisungs-Operators kann nicht als ZIEL einer Zuweisung sein.
- not an allowed ARRAY type
  Das Element auf der linken Seite einer Array-Klammer kann nicht 'wie ein Array' verwendet werden.
- ARRAY type mismatch (dimensions, etc)
- name already defined
  Der Name, den Sie für die Deklaration einer Variablen / Funktion / Prozedur verwenden wollen, wird bereits für 'etwas Anderes' verwendet.
- missing 'struct' or 'endstruct'
- internal error - SORRY !
  Sollte dieser compilerinterne Fehler auftreten, melden Sie dies bitte dem Entwickler (zusammen mit dem Quelltext, bei dessem Übersetzen der Fehler auftrat).
- unknown error code (< number >)
  Es trat ein Fehlercode auf, für den noch kein entsprechender Eintrag in einer internen String-Tabelle existiert.

Bitte melden Sie dies dem Entwickler, zusammen mit dem ausnahmsweise numerisch angezeigten Fehlercode (<number>).

5. Beispiele

Im Installationsarchiv sind einige Beispielprogramme enthalten, die z.T. als TEST bei der Implementierung der Script-Sprache verwendet wurden, z.g.T. aber auch als DEMO-PROGRAMME vorgesehen sind.

Einige der folgenden Beispiele wurden als 'verlinkter Hypertext' mit dem im Script-Editor integrierten HTML-Export-Tool erzeugt. Im Quelltext erkannte Schlüsselwörter werden **fett** dargestellt, und enthalten Links zur Dokumentation. Im Quelltext erkannte Namen von Funktionen und Variablen werden ebenfalls **fett** dargestellt, sie funktionieren *hier* als Link aber i.A. nicht, weil in den abgedruckten Beispielen nur Ausschnitte des Scripts enthalten sind (aber nicht der Deklarationsteil).

Nach der Installation des Programmiertools (z.B. unter c:\MKT\CANdbTerminalProgTool) finden Sie die *kompletten* Beispielprogramme unter c:\MKT\CANdbTerminalProgTool\Programs\script_demos*.cvt .
Eine Liste der mitgelieferten Beispiele finden Sie in der Übersicht in diesem Dokument.

script_demos\CANgate1.cvt

Ein einfaches 'CAN-Gateway' : Registriert bestimmte CAN-Message-Identifier für den Empfang auf 'CAN1' (erster CAN-Port), und sendet die empfangenen Telegramme auf dem zweiten Port ('CAN2') wieder aus.
Hinweis: In den höchstwertigen 2 Bits des 32-Bit 'Identifiers' (tCANmsg.id) wird die CAN-Bus-Nummer codiert (als 2-Bit-Zahl).
Die Prozedur 'CANGatewayProcess' wird periodisch aus der Hauptschleife aufgerufen, um die Gateway-Funktion zu realisieren.

```
proc CANGatewayProcess() // Process received CAN messages in "gateway mode"
  local tCANmsg can_msg; // use local variable wherever possible
  local int can_port;    // can port index, 0..3, from a 2-bit-field of
rcvd message
  local int id_only;     // can message id, without bus-number-bits

  // As long as there are CAN messages waiting in the FIFO,
  // handle them (can_receive drains the CAN-RX-FIFO and
  // copies the next received message into the specified variable) :
  while( can_receive( &can_msg ) )

    // Get the two-bit, zero-based "CAN port index" from the upper two bits
in the ID:
    can_port:= (can_msg.id & (cCanIdBit_Bus2 | cCanIdBit_Bus3) ) /
cCanIdBit_Bus2;

    // Get the CAN-message-id without bus number (~ is bitwise "not") :
    id_only := can_msg.id & (~(cCanIdBit_Bus2| cCanIdBit_Bus3) ); // ID w/o
bus-number

    // Count received CAN frames.. only for debugging purposes
    nFramesRcvd := nFramesRcvd + 1;

    // Show received message if wanted, on the text panel:
```

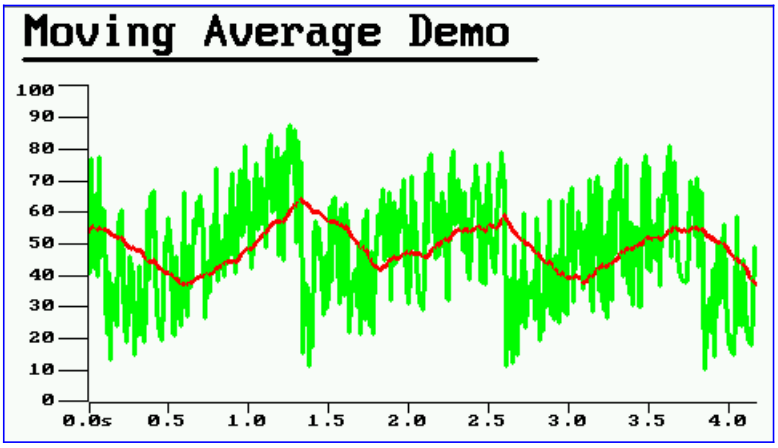
```
      if ( show_frames ) then
        print("CAN"+itoa(can_port+1)+" ",hex(id_only,4)," ", can_msg.len );
        for i:=0 to can_msg.len-1
           print(" ",hex(can_msg.b[i],2) );
        next i;
        print("\r\n");  // carriage return + new line
        if( tscreen.cy >= tscreen.vis_height ) then
           gotoxy(0,0); // screen 'full'; back to 'home' position
        endif;
        clreol;         // clear to end-of-line
      endif; // show_frames ?

      select( can_port ) // Note: can_port is an INDEX (2-bit number, can
count from 0 to 3) !
        case 0: // message received from FIRST CAN port ("CAN1")
          // At this point, we know the message was RECEIVED from the first
CAN port
          // because both 'bus number bits' in can_msg.id are CLEARED. For
details,
          // right-click on cCanIdBit_Bus2, then search it in the manual.
          // Modify the received message's 'id' field so it will be
          // tranmitted on the SECOND bus.
          //     From the help system: The CAN BUS NUMBER (!)
          //     is encoded in the most significant bits (bits 31..30) :
          can_msg.id := id_only | cCanIdBit_Bus2;  // modify ID-field(!) to
transmit on CAN2
          // We could also modify the ID (bits 10..0) or the data of the
          //    "echoed" CAN message, but in this simple demo, we don't:
          // can_msg.b[0] := 0; // set the first byte in the CAN message to
zero
          can_transmit( can_msg );  // send the response

        case 1: // Message received from the SECOND(!) CAN port ("CAN2")
          can_msg.id := id_only;  // modify ID-field(!) to transmit on CAN1
          // .. etc, add your own code here ..
          can_transmit( can_msg );  // send the response

        case 2: // Message received from the THIRD(!) CAN port ("CAN3")
          // (only available if "CAN-via-UDP" is enabled, MKT-View II/III/IV
only have TWO ports)
          break;   // don't send messages from THIS port on any other port

        case 3: // Message received from the FOURTH(!) CAN port ("CAN4")
          // (only available if "CAN-via-UDP" is enabled, MKT-View II/III/IV
only have TWO ports)
          break;   // don't send messages from THIS port on any other port

      endselect; // .. can_msg.id & (cCanIdBit_Bus2 | cCanIdBit_Bus3) ..
    endwhile; // end of CAN-message processing loop
  endproc; // CANGatewayProcess()
```

script_demos\MovgAvrg.cvt : Gleitende Mittelwertbildung

Dieses Beispiel berechnet einen gleitenden Mittelwert für ein Signal, über ein festes Zeitintervall. Dazu wird ein Timer und ein einfaches Array verwendet.
Gleitende Mittelwertbildung stellt ein wenig rechenintensives Verfahren zur Tiefpassfilterung

dar, z.B. zur Rauschunterdrückung in einem Messsignal.
Anno 2018 war eine brauchbare Beschreibung des Prinzips bei
de.wikipedia.org/wiki/Gleitender_Mittelwert zu finden.

Das ungefilterte Testsignal (grün) und das als gleitender Mittelwert 'gefilterte' Signal (rot) wird im Beispielprogramm als Y(t)-Diagramm dargestellt:



Screenshot aus Demo 'gleitende Mittelwertbildung'

Fragment aus dem Timer-Event-Handler, mit Berechung des gleitenden Mittelwertes:

```
   // Implementation of the moving average filter
   i := display.FilterIn;  // new filter input value
   if( Avrg1NumSummands<C_AVRG_BUF_SIZE ) then
      // Didn't reach the 'wanted' length (number of summands) yet :
      Avrg1NumSummands := Avrg1NumSummands+1;
      Avrg1Buffer[Avrg1BufIndex] := i;
      Avrg1Sum := Avrg1Sum + i;
   else // reached the "wanted" queue length : forget 'oldest' value
      Avrg1Sum := Avrg1Sum - Avrg1Buffer[Avrg1BufIndex]; // remove oldest
summand
      Avrg1Buffer[Avrg1BufIndex] := i; // store NEWEST summand in the
buffer
      Avrg1Sum := Avrg1Sum + i;        // add newest summand to sum
   endif;
   Avrg1BufIndex := (Avrg1BufIndex+1) % C_AVRG_BUF_SIZE; // new circular
buffer index
   display.FilterOut := Avrg1Sum / Avrg1NumSummands;
```

Ähnliche Themen (Signalverarbeitung per Script) : numerischer Integrator, gleitende Mittelwertbildung.

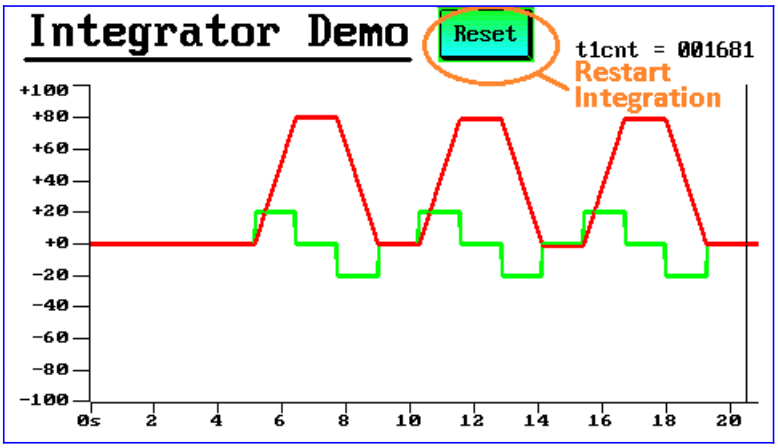
script_demos\Integrator.cvt : Numerische Integration

Dieses Beispiel führt eine numerische Integration diskreter Werte durch, nach dem Prinzip der Trapezregel. Ähnlich wie bei der gleitenden Mittelwertbildung (s.O.) wird dazu wird ein Timer verwendet. Ein Array zum Speichern der zu integrierenden Werte ist nicht nötig.

```
var
   int   Integr1Reset;         // flag to reset (clear) the integrator
   int   Integr1PrevTimestamp; // high-res timestamp of the previous sample
   float Integr1PrevValue;     // current "integrated" value (sum of areas)
   float Integr1Value;         // current "integrated" value (sum of areas)
   tTimer Timer1; // "periodic" timer for data acquisition (or simulation)
   int    Timer1Counter;
endvar;

  (...)

//---------------------------------------------------------------------------
func OnTimer1( tTimer ptr pMyTimer )  // periodically called Timer Event
Handler
   local int i, timestamp;
   local float f, delta_t;

   // Um dieses Demo auch ohne CAN-Bus 'in Aktion' zu sehen,
   // wird ein Testsignal mit positiven und negativen Pulsen erzeugt.
   // Das Testsignal wird auch als Y(t)-Diagramm auf dem Display geplottet.
   i := int(system.timestamp/(4*cTimestampFrequency)); // -> 4 seconds per
step
   select( i % 4 )       //      ___
      case 0: i := 0;    //     | + |
      case 1: i := 20;   // ___|   |___      ... (cycle repeats endlessly)
      case 2: i := 0;    //            | - |
      case 3: i := -20;  //            |___|
   endselect;
   display.IntegrIn := float(i);

   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   // Oben:  Erzeugen des Testsignals .
   // Unten: Implementierung des numerischen Integrators .
   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

   timestamp := system.timestamp; // read current timestamp (see manual)
   f := display.IntegrIn;         // get the new input value (to be
integrated)
   if( Integr1Reset ) then   // 'Reset' the integrator ?
      // (this must be done at least once after power-on) :
      Integr1Value := 0.0;  // no areas summed up (keine Flächen
aufaddiert)
      Integr1Reset := FALSE; // 'done' (integrator has been reset)
   else  // normal operation: integrate !
      // delta_t = time difference between current and previous sample
[sec],
      //           using the display's high-resolution timestamp
generator :
      delta_t := (timestamp-Integr1PrevTimestamp) / cTimestampFrequency;
      // Numerischer Integrator, siehe de.wikipedia.org/wiki/Trapezregel :
      Integr1Value := Integr1Value + delta_t * 0.5 * ( Integr1PrevValue + f
);
   endif;
   Integr1PrevValue  := f;             // save current input value for the
NEXT time
   Integr1PrevTimestamp := timestamp;  // save timestamp for the NEXT time
```

```
   display.IntegrOut := Integr1Value;  // copy integrated result into a
DISPLAY variable

   Timer1Counter := Timer1Counter + 1; // count number of timer events
(test)
   return TRUE;  // TRUE = "keep on firing timer events" (FALSE would stop)
endfunc; // end OnTimer1
```

Das vom Script erzeugte Testsignal (grün) und das Ausgangssignal des Integrators (rot) wird im Beispielprogramm als Y(t)-Diagramm dargestellt:



Screenshot aus Demo 'numerische Integration'

Durch Anklicken des 'Reset'-Buttons kann der Integrator an einem beliebigen Zeitpunkt neu gestartet werden. Dazu setzt der Button das Flag 'Integr1Reset' (Integrator 1 Reset), was im oben gezeigten Script zum Löschen der Summe ('Integr1Value') führt. Eine ähnliche Funktion könnte auch bei 'echten' Signalen nötig sein, z.B. wenn das zu integrierende Sensorsignal einen unerwünschten DC-Anteil enthält, der die Flächensumme nach langer Integrationszeit 'an den Poller' laufen lassen würde. Um dies zu vermeiden, wird in der Praxis statt eines idealen Integrators oft ein *'leaky integrator'* verwendet. Das komplette, im Installationsarchiv enthaltene Beispiel enthält eine entsprechende Option.

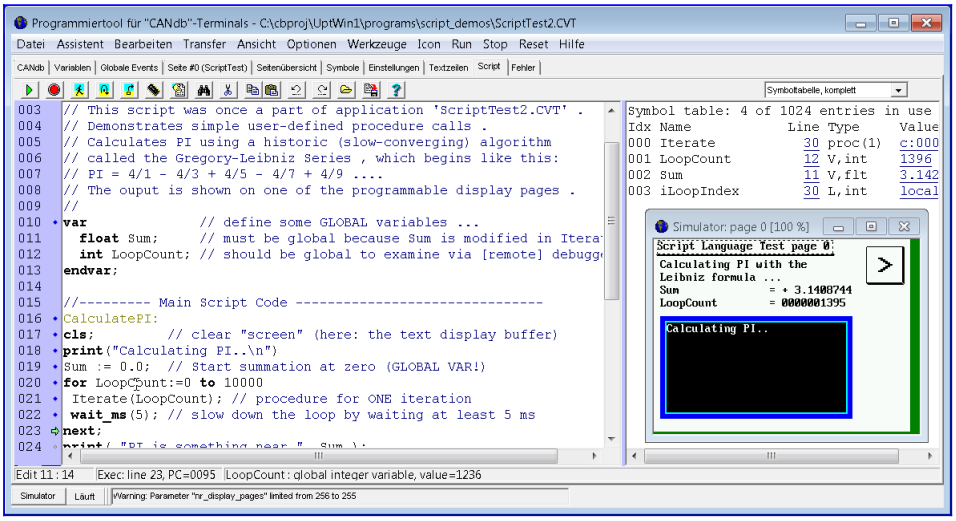
Ähnliche Themen (Signalverarbeitung per Script) : numerischer Integrator, gleitende Mittelwertbildung.

script_demos\ScriptTest1.cvt

Ein einfaches Testprogramm für die Ablaufsteuerung ("Programmfluss") in der Script-Sprache. Diente auch zum Testen der Fliesskomma-Konvertierungen.

script_demos\ScriptTest2.cvt

Ein weiteres Testprogramm für die Script-Sprache. Berechnet den Wert der Kreiszahl PI (circa 3.14159) mit einer Iterationsschleife.



Programmiertools beim Abarbeiten des Demos 'ScriptTest2.cvt'
Zum Vergrößern in das Bild klicken.

Zur Berechnung wird die langsam konvergierende Leibnitz-Serie verwendet, die hier (zu Demonstrationszwecken) für jede Iteration eine anwenderdefinierte Prozedur namens 'Iterate' aufruft:

```
//--------- Procedures and Functions (subroutines) ---------
proc Iterate(int iLoopIndex) // ONE iteration to calculate PI
 // Even loops: ADD, Odd loops: SUBTRACT from sum .
 // Note the BITWISE AND to check the least significant bit:
 if (iLoopIndex & 1) == 0
   // The division must use floats, so use 4.0 not 4,
   // otherwise 4 / ( 2 * (iLoopIndex + 1) would be calculated
   // with integer values, giving an INTEGER quotient !
   then Sum := Sum + 4.0 / (2 * iLoopIndex + 1);
   else Sum := Sum - 4.0 / (2 * iLoopIndex + 1);
 endif;
endproc; // end Iterate()
```

script_demos\ScriptTest3.cvt

Ein fortgeschritteneres Testprogramm für Arrays, Typdefinitionen, scrollbaren Text, und CAN-Bus-Funktionen.
Wegen Nutzung der *erweiterten* Script-Funktionen nur mit Freischaltung lauffähig.

script_demos\DisplayTest.cvt

Test- und Beispielprogramm, in dem Anzeige-Elemente per Script gesteuert werden, z.B. display.menu_mode, display.menu_index.
Mit Beispiel zum Aufruf einer benutzerdefinierten Prozedur ("GoToNextField") aus der Reaktionsmethode eines graphischen Buttons:

```
//-------------------------------------------------------------
proc GoToNextField // Called from the display (on button)
  display.menu_mode  := mmNavigate; // switch to "select"
(navigate) mode
```

```
  display.menu_index := (display.menu_index+1) % 8; // switch
to next field
endproc; // end GoToNextField
```

script_demos\diagrams.cvt

Test- und Beispielprogramm für per Script 'gefütterte' Diagramme und zur 'nahtlosen' Verarbeitung von Abtastwerten per Datenerfassungseinheit (DAQ).

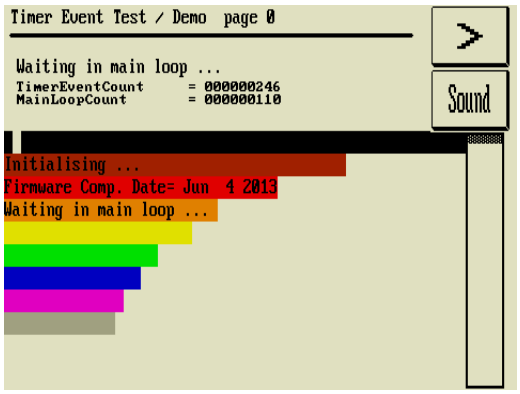
script_demos\TimerEvents.cvt

Test- und Demoprogramm für Timer-Events.
Das Script verwendet ein Array aus Timern, die per Script mit unterschiedlichen Zykluszeiten programmiert werden:

```
   const     // define a few constants...
      C_NUM_TIMERS = 10;         // number of simultaneously running timers
      int ResistorColours[10] =  // ten 'colour codes' [0..9]:
       { clBlack, clBrown, clRed,     clOrange, clYellow, // [0..4]
         clGreen, clBlue,  clMagenta, clDkGray, clLtGray  // [5..9]
       };
   endconst;
      ...
   typedef
      tMyTimerControl = struct
         int   iEventCount;
         float fltMeasuredFrequency;
      endstruct;
   endtypedef;
      ...
   var // declare a few GLOBAL variables...
     tTimer MyTimer[C_NUM_TIMERS]; // an array of timers for the stress-
test
     tMyTimerControl MyTimerCtrl[C_NUM_TIMERS]; // an array of self-defined
'timer controls'
   endvar;
      ...
   // Start the timers for the timer 'stress test'
   for i:=0 to #(C_NUM_TIMERS-1)
     MyTimer[i].user := i;  // use the index as 'user defined ID' for this
timer
     setTimer( addr(MyTimer[i]), 37+20*i/*ms*/, addr(OnMyTimer) );
   next i;
      ...
   //---------------------------------------------------------------------
   func OnMyTimer( tTimer ptr pMyTimer )  // another TIMER EVENT HANDLER...
     // Shared by  timers which run at different intervals.
     // The activity of each of these timers is visualized
     //     as a horizontal coloured bar on a text panel .
     local int i,x,y;
     local tMyTimerControl ptr pCtrl;
     debugTimer := pMyTimer[0]; // copy the argument into a global variable
(for debugging/"Watch")
     TimerEventCount := TimerEventCount + 1; // global counter for ALL
timer events
```

```
     i := pMyTimer.user;        // user defined index of this timer, here:
i = 0..9
     pCtrl := addr(MyTimerCtrl[i]); // address of a user-defined 'timer
control' struct
     x := pCtrl.iEventCount % tscreen.vis_width;
     y := i;
     tscreen.cell[y][x].bg_color := ResistorColours[i];
     tscreen.cell[y][x+1].bg_color := clWhite;
     tscreen.modified := TRUE;
     pCtrl.iEventCount := pCtrl.iEventCount + 1; // count the events fired
by THIS timer
      // ...
     return TRUE; // TRUE = 'the event has been handled here' (FALSE would
not fire more events)
   endfunc; // OnMyTimer()
```

Bei jedem Timer-Ereignis wird ein Zähler für den entsprechenden Timer erhöht, und der Zählerstand als farbiger Balken auf einem Text-Panel angezeigt:



Screenshot aus dem 'Timer-Event'-Demo

Der obere Balken zeigt den Stand des 'schnellsten' Timers an (Index 0, Farbcode Schwarz), der Untere den Stand des 'langsamsten' Timers (hier: Index 8, Farbcode Grau).

Zum Messen des Jitters diente ein weiterer Script-Timer, mit dem zyklisch CAN-Telegramme gesendet wurden:

```
  //---------------------------------------------------------------------
  func OnCANtxTimer( tTimer ptr pMyTimer )  // a TIMER EVENT HANDLER...
    local tCANmsg msg;  // use LOCAL variables (not globals) in event
handlers !
    msg.id := 0x335;    // set CAN message identifier for transmission
    msg.len:= 8;        // set CAN data length code (max. 8 bytes = 2
doublewords)
    msg.dw[0] := 0x11223344; // set four bytes in a single doubleword-move
(faster than 4 bytes)
    msg.dw[1] := 0x55667788; // set the last four bytes in the 8-byte CAN
data field
```

```
    can_transmit( msg ); // send the CAN message to the bus
    return TRUE; // TRUE = 'the event has been handled here' (FALSE would
not fire more events)
  endfunc;
```

Der entsprechende Timer wird im Initialisierungsteil des Scripts mit dem folgenden Befehl gestartet:

```
  // Start another timer for periodic CAN transmission .
  // Jitter can be checked with a CAN bus analyser.
  setTimer( CANtxTimer, 100/*ms*/, addr(OnCANtxTimer) ); // OnCANtxTimer
called every 100 ms
```

Bei einem Test mit dem MKT-View III ergab sich eine Abweichung von ca +/- 5 Millisekunden in den Zeitmarken eines CAN-Bus-Testers (bedingt durch das *kooperative*, nicht *präemptive*, Multitasking innerhalb der Script-Sprache). In zukünftigen Versionen der Firmware *könnte* der Jitter bei den Timer-Events noch reduziert werden (2013-06-05).

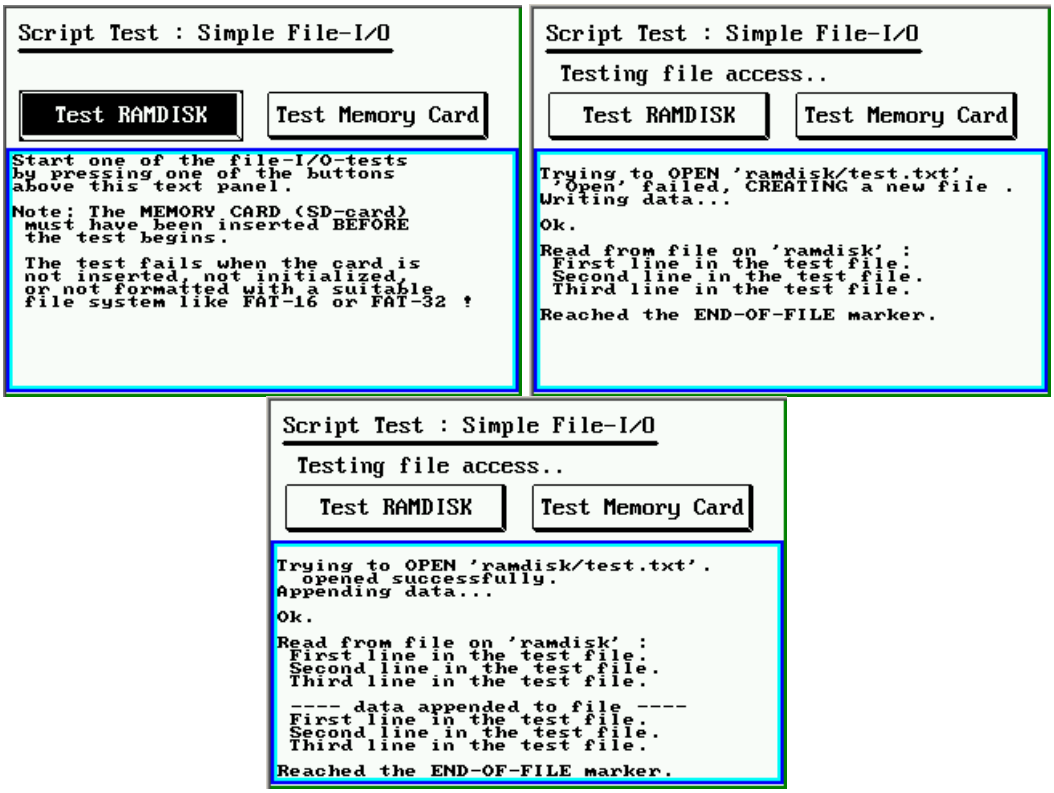
script_demos\FileTest.cvt

Test- und Demoprogramm für die Datei-I/O-Funktionen. Verschiedene Tests können durch Druck auf die graphischen Buttons auf der ersten Anzeigeseite gestartet werden:
'Test RAMDISK' schreibt und liest eine Datei auf der RAMDISK,
'Test Memory Card' verwendet für den gleichen Test die Speicherkarte.
Die in der Script-Sprache programmierte Funktion 'TestFileAccess' wird für beide Speichermedien verwendet. Das zu testende Medium wird durch das Funktionsargument 'pfs_path' spezifiziert; die erlaubten Werte sind 'ramdisk' (zum Testen der RAMDISK) oder 'memory_card' zum Testen der Zugriffe auf der Speicherkarte. Hier eine *gekürzte* Version der Testfunktion :

```
//-------------------------------------------------------------------
func TestFileAccess( string pfs_path )
  // Part of the test program for file I/O functions . Taken from
'FileTest.cvt' .
  // [in]  pfs_path : path for the pseudo-file-system like "ramdisk" or
"memory_card"
  local int fh;       // file handle
  local int i;
  local string fname;
  local string temp;

  // Build a complete filename, with a path:
  fname := pfs_path+"/test.txt";

  // First try to OPEN the file (but don't try to CREATE, i.e. OVERWRITE
it):
  fh := file.open(fname,O_RDWR); // try to open existing file, read- AND
write access
```

```
  if( fh>0 ) then
    file.seek(fh, 0, SEEK_END ); // Set file pointer to the END of the file
    // write a separator between the 'old' and the 'new' part of the file:
    file.write(fh,"\r\n---- data appended to file ----\r\n");
  else  // file.open failed, so try to CREATE a 'new' file:
    fh := file.create(pfs_path+"/test.txt",4096);  // create a file, with
pre-allocation
  endif;
  if( fh>0 ) then    // successfully opened or created the file ?
    file.write(fh,"First line in the test file.\r\n");
    file.write(fh,"Second line in the test file.\r\n");
    file.write(fh,"Third line in the test file.\r\n");
    file.close(fh);
  else               // neither file.open nor file.create were successful:
    print( "\r\nCould not open or create a file !" );
    return FALSE;
  endif;

  // After writing and closing the file (above), open it again, and READ
the contents:
  fh := file.open(pfs_path+"/test.txt", O_RDONLY | O_TEXT);  // try to open
the file for READING
  if( fh>0 ) then    // successfully created the file ?
    print( "\r\nRead from file on '",pfs_path,"' :" );
    while( ! file.eof(fh) ) // repeat until the end of the file.......
      temp := file.read_line(fh); // read one line of text from the file
      print( "\r\n ", temp );     // dump that line to the text panel
      Progress := Progress+1;
    endwhile;
    print( "\r\nReached the END-OF-FILE marker." );
    file.close(fh);
  else
    print( "\r\nCould not open file for reading !" );
    return FALSE;
  endif;
  return TRUE;
endfunc; // TestFileAccess
```

Durch zusätzliche 'print'-Anweisungen in der Applikation 'FileTest.cvt' erzeugt das Programm beim ersten und zweiten Druck auf den Button 'Test RAMDISK' folgende Ausgabe auf einem Text-Panel:

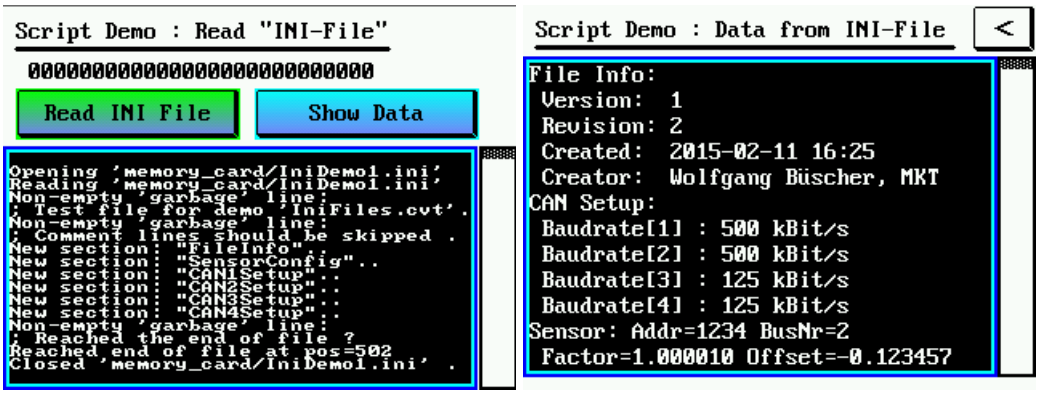


Testlauf des Datei-I/O-Demos (script_demos/FileTest.cvt)

Weitere Beispiele mit Datei-Ein/-Ausgabe: VT100/VT52-Emulator, Einlesen von INI-Dateien .

script_demos\IniFiles.cvt

Ein weiteres Beispiel zu die Datei-I/O-Funktionen, allerdings ***nur für Fortgeschrittene***. Liest eine 'Konfigurationsdatei', die ähnlich wie eine Windows-INI-Datei aufgebaut ist, zeilenweise ein.
Die aus der Datei gelesenen Parameter werden in Script-Variablen abgelegt, und können zur Kontrolle auf einem scrollbaren Text-Panel angezeigt werden:



Screenshots vom Demoprogramm zum Einlesen von Konfigurationsdateien ("INI-Files")

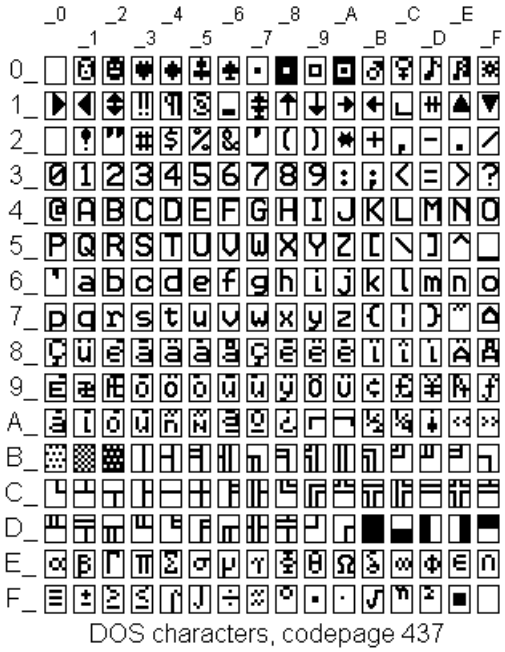
Darin wird die Funktion 'file.read' als Parser verwendet. Siehe Beschreibung von file.read.
Die Testdatei IniDemo1.ini ist im Installationsarchiv enthalten.
Zum Testen dieser Applikation im Simulator (Programmiertool) wird die INI-Datei aus dem einstellbaren Verzeichnis zur Simulation der Speicherkarte gelesen (ein Speicherkartenleser wird nicht benötigt).
Zum Testen auf einer 'echten' Hardware muss die Beispieldatei (IniDemo1.ini) aus dem Unterverzeichnis 'sim_mc' ("simulated memory card" im Programmiertool) in das Stammverzeichnis einer geeigneten Speicherkarte (mit FAT oder FAT32) kopiert werden.

script_demos\TScreenTest.cvt

Test für den emulierten 'Textbildschirm', mit Prozeduren zum Zeichnen von Linien und Rahmen ("tscreen") .
Diente als erster Test für lokale Variablen, Parameter-Übergabe in Prozeduren, und rekursive Prozeduraufrufe .

```
//-------------------------------------------------------------
proc FillScreen1(string sFillChar)
  // PROCEDURE to fill the screen with a colour test pattern .
  local int X,Y,old_pause_flag;
  old_pause_flag := display.pause;
  display.pause  := TRUE; // disable normal screen output
  for Y:=0 to tscreen.ymax
    for X:=0 to tscreen.xmax
      gotoxy(X,Y);
      setcolor( clWhite,
          rgb((11*(X+Y))&255, (9*(Y-X))&255, (3*X)&255 ) );
      print( sFillChar ); // print a single character
    next X;
  next Y;
  display.pause:=old_pause_flag; // resume display output ?
endproc; // end FillScreen1()
```

Darüberhinaus demonstriert TScreenTest .cvt, wie mit den speziellen DOS-"Grafikzeichen" (aus "Codepage 437" oder "Codepage 850") Linien, Rechtecke, und Tabellengitter auf dem Textbildschirm gezeichnet werden können. Auf der DOS-"Grafik" basiert auch ein einfaches Spielprogramm, bei dem per Tastatur der Kopf eines 'Wurms' (oder Schlange) durch ein Labyrint gesteuert wird. Auf ähnliche Weise können mit Hilfe des unten abgebildeten DOS-Zeichensatzes weitere bewegte oder statische Grafiken realisiert werden.



Zeichensatz mit 'Grafikzeichen' (DOS Codepage 437)

script_demos\LoopTest.cvt

Demonstriert die Verwendung der diversen Schleifen-Konstrukte in der Script-Sprache, z.B. for-to-next; darüberhinaus ein 'animiertes' Farb-Testprogramm für den Bildschirm, unter Verwendung von Schleifen, gotoxy, color, rgb, dem Kommando print, und dem Flag display.pause mit dem in diesem Beispiel verhindert wird, daß die Anzeige 'zur falschen Zeit' aktualisiert wird (hier: um Flackern beim Neu-Zeichnen des Textbildschirms zu vermeiden) .

script_demos\TimeTest.cvt

Test für die Funktionen zur Umwandlung von Datum und Uhrzeit, z.B. time.date_to_mjd und time.mjd_to_date .
Demonstriert auch, wie eine 'Unix-Zeit' (Anzahl von Sekunden seit der 'Geburt' von Unix im Januar 1970) in die entsprechende Anzahl von Jahren, Monaten, Tagen, Stunden, Minuten, und Sekunden aufgespaltet werden kann.

script_demos\StringTest.cvt

Testprogramm zur Verarbeitung von Zeichenketten, z.B. strlen(), strpos() und substr() .

script_demos\StructArrayTest.cvt

Test mit einem **Array**, in dem jedes Element aus einer anwenderdefinierten **Struktur** besteht. Jede Struktur besteht in diesem Beispiel aus **Ganzzahlen, Fließkommazahlen, und Zeichenketten** .
Die Schleife zum Anfüllen des Arrays dient seit Oktober 2010 auch als Benchmark für die Geschwindigkeit, mit der das Script auf dem Zielsystem(!) abgearbeitet werden kann .

script_demos\PageMenu.cvt

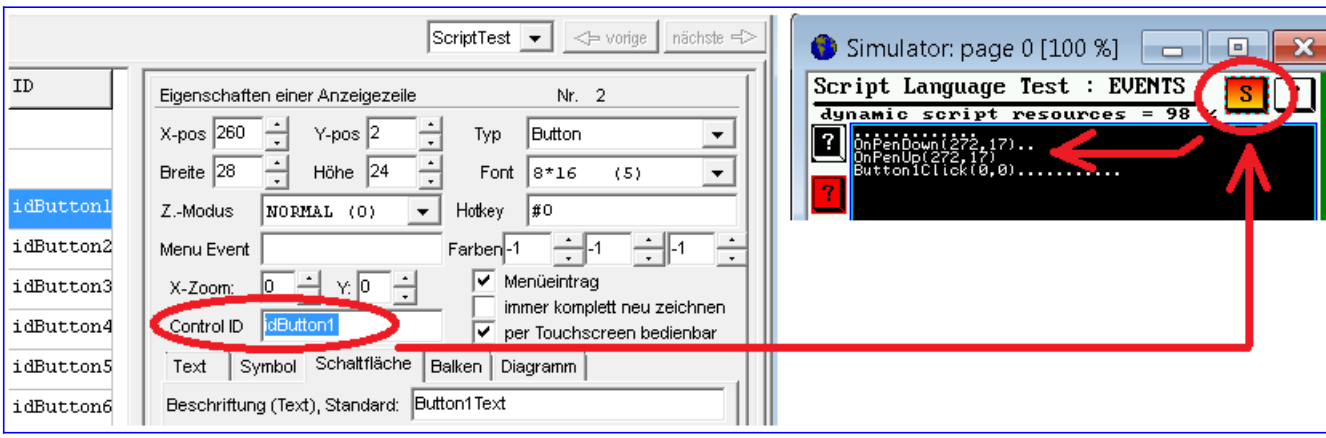
Dieses Beispielprogramm erstellt (per Script) zur Laufzeit eine 'Menü-Seite', auf der die Namen aller existierenden Anzeige-Seiten aufgelistet werden. Per Tastatur (Cursortasten und ENTER) kann der Bediener eine Seite im Menü anwählen und aufrufen.

script_demos\EventTest.cvt

Dies ist ein Test- und Beispielprogramm für Low-Level Event-Handler ("Systemereignisse" wie Tastendruck, Drehknopf, Touchscreen) und für Events, die von graphischen Bedienelementen ('Controls', z.B. Buttons) ausgelöst wurden.
Im Script-Quelltext werden zunächst symbolische Identifier definiert, mit denen mehrere Buttons (Schaltflächen) und ähnliche Bedienelemente voneinander unterschieden werden können:

```
const
  // Identifiers for certain DISPLAY ELEMENTS with event handling per
script.
  // These IDs are passed as 2nd parameter to the OnControlEvent() handler.
  // Note that the maximum length for these IDs (in the page definition
  // table) is limited to 11 (ELEVEN) characters. Control-ID ZERO is
invalid.
  idButton1 = 1; // first button
  idButton2 = 2; // second button, etc..
  idButton3 = 3;
endconst;
```

Die so definierten Identifier (z.B. **idButton1**) werden bei der Definition des entsprechenden Anzeige-Elements (auf der Registerkarte 'Seite N' unter 'Eigenschaften einer Anzeigezeile') im Feld 'Control ID' eingegeben (siehe Screenshot). Dieser Wert wird beim Aufruf des Handlers OnControlEvent (int event, int controlID, int param1, int param2) als Parameter 'controlID' übergeben.
Beim Drücken und Loslassen des ersten Buttons wird daher neben dem Event-Typ auch dessen Identifier (in diesem Beispiel **idButton1**) an den Event-Handler übergeben.
Dadurch können mit *einem* Event-Handler mehrere *gleichartige* Bedienelemente verwaltet werden (wie z.B. in der 'elektronischen Orgel').



Screenshots aus 'EventTest.cvt', rechts: Button mit Identifier **idButton1**

Im Event-Handler (OnControlEvent) werden üblicherweise select-case-Anweisungen verwendet, um zwischen verschiedenen 'Controls' (deutsch: 'Bedienelementen') zu unterscheiden.
In ähnlicher Form wird auch zwischen den Event-Typen (z.B. evPenDown, evPenMove, evPenUp) unterschieden, wenn z.B. beim Drücken und beim Loslassen eines Buttons unterschiedliche Reaktionen realisiert werden sollen. Hier das Grundgerüst für einen 'OnControlEvent'-Handler, der in ähnlicher Form auch im 'Event-Test'-Demo verwendet wird:

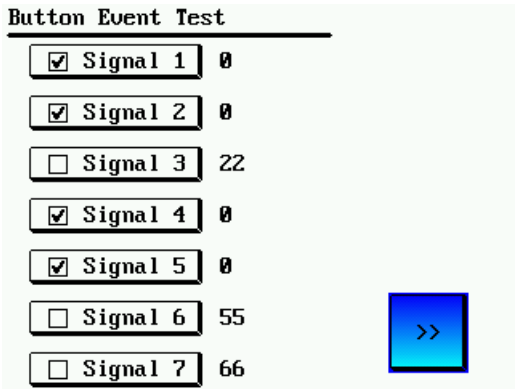
```
//--------------------------------------------------------------------
// Common handler for all 'visible elements which interact with the user'
//   on the current display page, aka "Control Elements"
//--------------------------------------------------------------------

func OnControlEvent(
         int event,     // [in] type of the event, like evClick, etc
         int controlID, // [in] control identifier (from page-def-table)
         int param1,    // [in] 1st message parameter, depends on event
         int param2 )   // [in] 2nd message parameter, depends on event
   // Called when 'something happens' with a certain control element
   // (button, menu item, edit field, etc) on the current display page .
   //  param1: client-X-coordinate or keyboard code (depends on event-type)
   //  param2: client-Y-coordinate (where applicable)
   local int x,y;
   select( event )   // what has happened (type of the event) ?
      case evClick:  // button, menu item, etc, was "clicked"...
         select( controlID ) // WHICH control element was "clicked" ?
            case idButton1:  // Button1 was 'clicked' ...
               // ... add your own code here :
               print( "Button 1 clicked.\r\n");

            case idButton2:  // Button2 was clicked
               // ... add your own code here ...
         endselect;  // end select controlID, for event "click" (via touch
or ENTER key)

      case evPenDown:        // the TOUCH-PEN was just pressed on a display
element
         select( controlID ) // WHICH control element ?
            case idButton1:  // Button1 has just been PRESSED (touch-pen
down)
               // ... add your own code here ...
            case idButton2:  // Button2 has just been pressed ->
               display.Wiper := 1; // windscreen wiper on (CAN)
               // ...
         endselect;  // end select controlID, for event "touch pen down"

      case evPenMove:        // finger or pen MOVED over a control (while
pressed)
         x := param1;        // client X coord
         y := param2;        // client Y coord
         select( controlID ) // WHICH control element ?
            case idButton1:  // finger or pen was moved while pressed on
Button1
               // ... add your own code here ...
```

```
            endselect; // end select controlID, for event "touch pen down"

         case evPenUp:           // the TOUCH-PEN was released over a display
element
            x := param1;         // client X coord
            y := param2;         // client Y coord
            select( controlID ) // WHICH control element ?
               case idButton1:  // Button1 has just been released (finger or
pen "up")
                  // ... add your own code here ...
               case idButton2:  // Button2 has just been released ->
                  display.Wiper := 0; // windscreen wiper off (CAN)
                  // ...
            endselect; // end select controlID, for event "touch pen released"
(just up again)
         endselect; // end select ( event )
      return 0;  // 0: let the system process this event, too
endfunc; // OnControlEvent
```

Hinweis: Der selbstdefinierte Identifier **idButton1** diente hier nur als **Beispiel**.
In Ihrer eigenen Anwendung sollten sie nach Möglichkeit 'sprechende' Symbole verwenden, z.B.
idStart, **idStop**, **idGearbox**, **idHorn**, **idWiper**, ...

script_demos\ButtonEventDemo.cvt

Dies ist ein weiteres (kleines) Beispielprogramm zum Verarbeiten von Button-Events, und darüberhinaus zum scriptgesteuerten Senden von CAN-Telegrammen (ohne DBC-Datei).
Beim Drücken eines Buttons wird ein Signal gesetzt, und per CAN-Bus (zusammen mit anderen Signalen) gesendet.
Beim Loslassen des Buttons wird das entsprechende Signal wieder gelöscht, und ein neues Telegramm per CAN-Bus gesendet.



Screenshot aus dem 'Button-Event-Test' (programs/script_demos/ButtonEventDemo.cvt)

Um die aktuellen Werte aller "CAN-Signale" neben den entsprechenden Buttons anzuzeigen, wird beim Aktualisieren der Anzeigeseite die benutzerdefinierte Funktion 'GetSignalByIndex' aufgerufen. Als Parameter ('index') wird dabei der Array-Index des entsprechenden CAN-Signals (CanSignals[index]) übergeben.
Die Reaktion auf das Drücken und Loslassen jedes der sieben (?) Buttons erfolgt in der

Prozedur OnControlEvent. Wie bereits im vorherigen Beispiel erläutert, wird diese Funktion (wenn im Script vorhanden) bei der Betätigung *jedes* Bedienelements (englisch: 'Controls') aufgerufen. Jedem Button ist ein eindeutiger Identifier ('control-ID', z.B. idButton1) zugeordnet, der im folgenden Beispiel auch als Array-Index für das dem Button zugeordnete CAN-Signal (CanSignals[0]...CanSignals[9]) verwendet wird:

```
const
  // Identifiers for certain DISPLAY ELEMENTS with event handling per
script.
  // These IDs are passed as 2nd parameter to the OnControlEvent() handler.
  // Note that the maximum length for these IDs (in the page definition
  // table) is limited to 11 (ELEVEN) characters. Control-ID ZERO is
invalid.
  idButton1 = 1; // first button
  idButton2 = 2; // second button, etc..
  idButton3 = 3;
  idButton4 = 4;
  idButton5 = 5;
  idButton6 = 6;
  idButton7 = 7;
    // (Calling the buttons "Button1" .. "Button7" etc is a no-brainer;
    //  but it emphasizes that these buttons may be used for "anything")
endconst;
```

Die Zuweisung eines symbolischen Identifiers an ein Bedienelement erfolgt wie im Beispiel 'Event Test'. Statt der nichtssagenden Namen 'idButton1'..'idButton7' könnten auch hier 'sprechende' Identifier verwendet werden (z.B. idStart, idStop, idUp, idDown, idLeft, idRight, ..).
Durch Subtraktion der Konstanten 'idButton1' kann aus jedem Button-ID ein passender Array-Index zum Zugriff auf die als Array deklarierten 'CAN-Signale' berechnet werden. Dadurch reicht *ein einziger* Event-Handler für *alle* Buttons aus:

```
//---------------------------------------------------------------------
---
func OnControlEvent(
         int event,     // [in] type of the event, like evClick, etc
         int controlID, // [in] control identifier (from page-def-table)
         int param1,    // [in] 1st message parameter, depends on event
         int param2 )   // [in] 2nd message parameter, depends on event
  // Called when 'something happens' with a certain control element
  // (button, menu item, edit field, etc) on the current display page .
  //  param1: client-X-coordinate or keyboard code (depends on event-type)
  //  param2: client-Y-coordinate (where applicable) .
  local int i;
  select( event )
      case evPenDown: // the TOUCH-PEN was just pressed over a display
element
         select( controlID ) // on WHICH control element was the touch pen
pressed down ?
            case idButton1:  // 1st button PRESSED, or...
            case idButton2:  // 2nd button PRESSED, or...
            case idButton3:
```

```
            case idButton4:
            case idButton5:
            case idButton6:
            case idButton7:
               // To keep it simple, we treat all these buttons the same
way.
               // Turn the buttons 'control ID' into a zero-based array
index:
               i := controlID - idButton1;
               CanSignals[i] = 1;
               SendCanSignals();
         endselect;  // controlID (for event 'PenDown')

      case evPenUp:   // the TOUCH-PEN was released from a display element
         select( controlID ) // from WHICH control element was the touch
pen lifted up ?
            case idButton1:  // 1st button RELEASED, or...
            case idButton2:  // 2nd button RELEASED, or...
            case idButton3:
            case idButton4:
            case idButton5:
            case idButton6:
            case idButton7:
               // To keep it simple, we treat all these buttons the same
way.
               // Turn the buttons 'control ID' into a zero-based array
index:
               i := controlID - idButton1;
               CanSignals[i] = 0;
               SendCanSignals();
         endselect;  // controlID (for event 'PenUp')

      endselect; // end select ( event )
   return 0;  // 0: let the system process this event, too
endfunc; // OnControlEvent
```

Zum Senden (per CAN) dient hier die benutzerdefinierte Prozedur 'SendCanSignals'. Darin werden alle per Buttons steuerbaren Signale in ein einzelnes CAN-Telegramm "gemappt", und anschliessend per **can_transmit** gesendet:

```
//---------------------------------------------------------------------
---
proc SendCanSignals  // Called from the event handler (OnControlEvent)
                     // to send the current 'signals' via CAN .
  local tCANmsg msg; // use LOCAL variables (not globals) in event handlers
!
  msg.id := 0x335;   // set CAN message identifier for transmission
  msg.len:= 8;       // set CAN data length code (max. 8 bytes)
  // Map our 'CAN-Signals' into a CAN message.
  // To keep this demo simple, each signal occupies EIGHT BITS in the CAN
frame.
  msg.bitfield[ 0,8] = CanSignals[0];
  msg.bitfield[ 8,8] = CanSignals[1];
  msg.bitfield[16,8] = CanSignals[2];
  msg.bitfield[24,8] = CanSignals[3];
  msg.bitfield[32,8] = CanSignals[4];
```

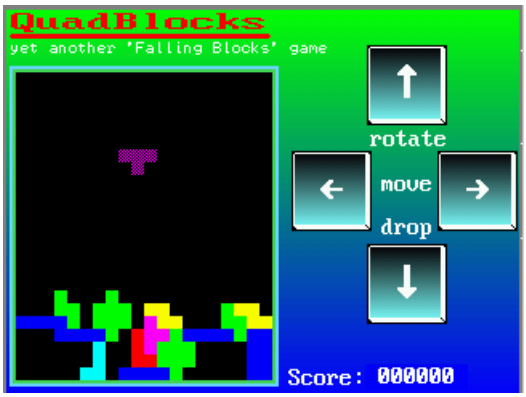
```
    msg.bitfield[40,8] = CanSignals[5];
    msg.bitfield[48,8] = CanSignals[6];
    can_transmit( msg ); // send message to the CAN network
endproc; // SendCanSignals()
```

script_demos\SendCANFromEditField.cvt

In diesem Beispiel wird zunächst ein numerischer Wert in einem Editierfeld eingegeben, per Script begrenzt (oder auch skaliert), und anschließend über den CAN-Bus verschickt.

script_demos\quadblox.cvt

Beispielprogramm zum Einsatz von **zweidimensionalen Variablen-Arrays**, **zweidimensionalen Konstanten-Arrays**, zum Abfragen der Tastatur, und von Ereignissen die mit den programmierbaren Buttons auf einer UPT-Anzeigeseite an das Script gesendet werden. Das Programm enthält die stark vereinfachte Implementierung eines früher weit verbreiteten "Puzzle-Spiels mit fallenden Blöcken", die während des Falls gedreht und verschoben werden können (der Name des Original-Spiels wird hier aus Copyright-Gründen nicht genannt). Bei Geräten mit Touchscreen können die Cursortasten mit den graphischen Buttons auf der rechten Seite des Bildschirms emuliert werden.



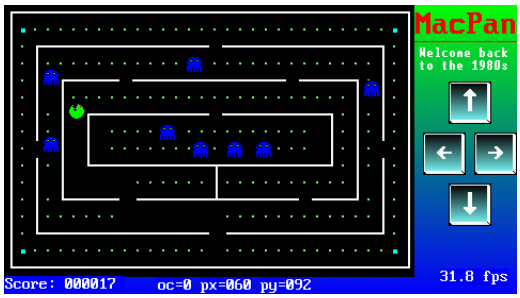
Screenshot aus programs/script_demos/quadblox.cvt

Das "QuadBlocks"-Demo wurde ursprünglich für Bildschirme mit 320*240 Pixeln entworfen. Per Script kann sich das Programm aber 'automatisch' auch für Geräte mit anderen Auflösungen anpassen. Dazu dient die Funktionen tscreen.vis_width und tscreen.vis_height . Bei Geräten mit 240*320 Pixel ("Portrait-Bildschirm" wie beim PDA) schaltet das Script zu einer anderen Seite um (statt der für den "Landscape"-Modus optimierten Seite). Dazu wird horizontale Auflösung des Bildschirms abgefragt (display.pixels_x, z.B. 128, 240, 320, oder 480 Pixels, je nach Ziel-Hardware).

script_demos\MacPan.cvt

Ein vom Autor in den 1990er Jahren als "C-Programm entwickeltes, einem Arcade-Klassiker aus den 1980ern nachempfundenes Demo. Um die auf damaligen Homecomputern verbreiteten "Sprite"-Grafiken nachzuempfinden, verwendet das Script einfache Bitmap-

Grafiken, die zum passenden Zeitpunkt (bei jedem Hauptschleifendurchlauf) im OnPageUpdate-Handler direkt in den Framebuffer gezeichnet werden.



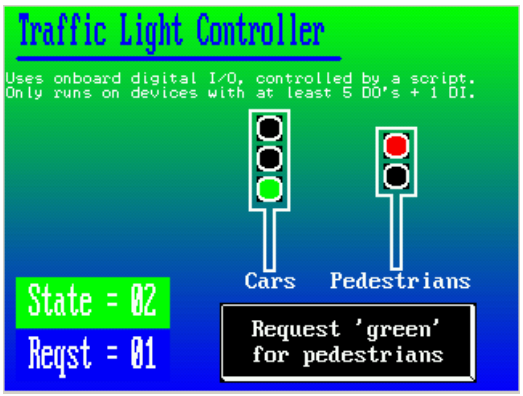
Screenshot aus programs/script_demos/MacPan.cvt

Zum Erzielen einer akzeptablen Geschwindigkeit (beim MKT-View III etwa 30 Frames/Sekunde) wird beim Übergeben von Arrays auf das zeitraubene *Kopieren* von Array-Daten verzichtet. Stattdessem werden bei Funktionsaufrufen nur *Referenzen* (Pointer) übergeben, z.B.:

```
for i:=0 to (MaxGhosts-1)
    if Ghost[i].alive and          // nur wenn Geist sichtbar und
       Ghost[i].on_center then     //  genau in der Mitte einer Zelle,
          FindYourWay( &Ghost[i] ); // dann neue Richtung "ausdenken"
    endif;
    MoveRunner(&Ghost[i]); // pixelweise Bewegung in JEDER Hauptschleife
next i;
```

script_demos\TrafficLight.cvt

Einfaches Script zur Steuerung einer Fußgängerampel. Demonstriert die Verwendung der Onboard-Digital-I/O (die nur in manchen Geräten in ausreichender Anzahl zur Verfügung steht, hier werden 5 digitale Ausgänge und 1 digitaler Eingang benötigt).



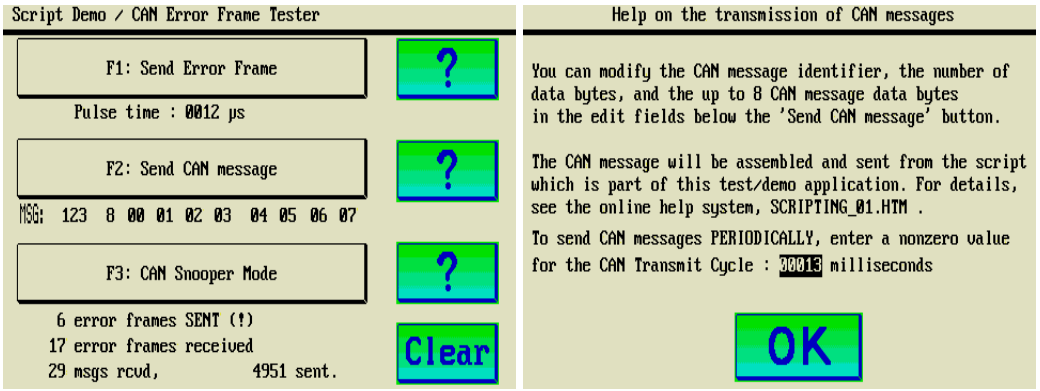
Screenshot aus programs/script_demos/TrafficLight.cvt

Fußgänger fordern per Tastendruck für sich 'Grün' an (erster digitaler Eingang). Als Alternative zum digitalen Eingang dient ein graphischer Button (für Geräte mit Touchscreen). Drei digitale Ausgänge steuern die Auto-Ampel (Rot, Gelb, Grün); zwei digitale Ausgänge

die Ampel für den Fußgängerübergang (nur Rot und Grün).
Der Zustand der beiden Ampeln wird als Grafik visualisiert. Unterschiedliche Bitmaps erhalten ihre Farbe in Abhängigkeit vom aktuellen Zustand der digitalen Ausgänge (aus diesem Grund funktioniert das Beispiel nicht auf dem MKT-View II : zu wenig digitale Outputs).

script_demos\ErrFrame.cvt

Mit diesem "sehr speziellen" Programm können sowohl CAN-Frames als auch CAN-Error-Frames(!) empfangen und gesendet(!!) werden.
Bei jedem empfangenen Error-Frame erfolgt ein akustisches Signal. Zur Erinnerung: Ein CAN-Error-Frame besteht aus sechs oder mehr dominanten Bits auf dem CAN-Bus, was im 'Normalfall' nie auftreten sollte). Einige schlecht designte Geräte bzw. Controller senden direkt beim Einschalten Error-Frames, was oft mit "nicht initialisierte Ports" (vor dem Initialisieren des CAN-Controllers) zusammenhing. Mit diesem Programm können solche Fehler im CAN-Netzwerk erkannt werden. Darüberhinaus kann mit diesem Programm die Robustheit eines aktives CAN-Netzwerk getestet werden: Senden Sie einen einzelnen Error-Frame (was mit diesem Programm möglich ist, allerdings nicht im Simulator). Ist mindestens ein aktiver CAN-Knoten im Netzwerk vorhanden, dann sollte dieser als Reaktion auf den Error-Frame einen weiteren Error-Frame senden. Ist dies nicht der Fall, ist das CAN-Netz möglicherweise "tot" (error-passive).
Die Länge der zu sendenden Error-Frames kann im Feld mit der Beschriftung 'Pulse Time' eingestellt werden. Der Default-Wert von 12 Mikrosekunden erzeugt nur in CAN-Netzen mit 1000 oder 500 kBit/Sekunde einen Error-Frame. Um in Netzen mit niedrigerer CAN-Baudrate Error-Frames zu provozieren, muss die Pulslänge entsprechend erhöht werden.



Screenshots vom 'CAN Error-Frame-Tester', programs/script_demos/ErrFrame.cvt

Um eine zusätzliche Buslast im zu testenden CAN-Netzwerk zu erzeugen, kann diese Applikation auch beliebige (zur Laufzeit editierbare) CAN-Telegramme senden. Per Touchscreen können die Felder unterhalb des Buttons 'Send CAN message' selektiert und editiert werden. Das Format ist:

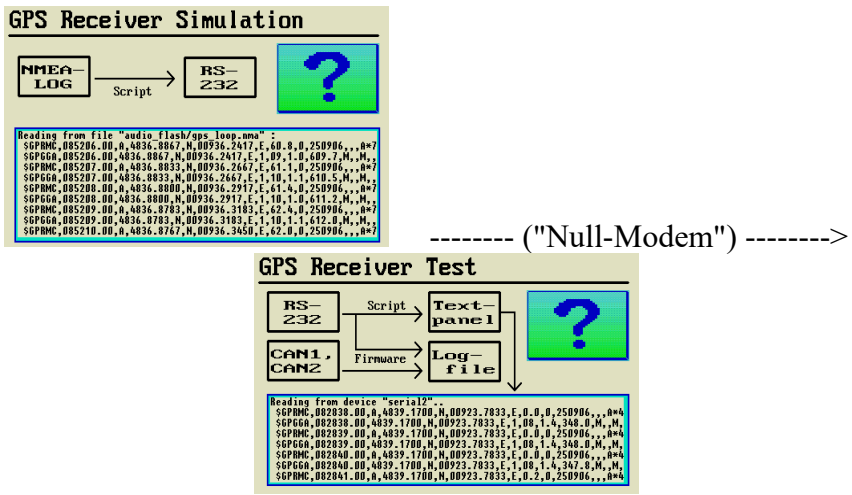
CAN-Message-ID (hexadezimal), Anzahl Datenbytes (0..8), und bis zu 8 Datenbytes (ebenfalls hexadezimal).

Telegramme können manuell gesendet werden (per Button 'Send CAN message', siehe

Screenshots), aber auch periodisch, mit Hilfe eines Timer-Events mit einstellbarer Zykluszeit (siehe Oben, dritter Screenshot mit 'CAN Transmit Cycle').

script_demos\SerialPt.cvt ; script_demos\GpsRcv01.cvt ; script_demos\GpsSim01.cvt

Verschiedene Tests für die serielle Schnittstelle, beim MKT-View II auch für *beide* serielle Schnittstellen). Die Applikation 'GpsSim01.cvt' wurde als Ersatz (d.h. zur "Simulation") eines GPS-Empfängers verwendet. Sie liest zeilenweise GPS-Daten aus einem NMEA-Logfile (Zeile für Zeile), und sendet diese über die serielle Schnittstelle. Die Applikation 'GpsRcv01.cvt' stellt (mehr oder weniger) das Gegenstück dar: Mit ihr werden empfangene NMEA-Sequenzen auf einem Text-Panel angezeigt. Jede Applikation wird zum Testen in ein MKT-View II geladen. Beiden Geräte müssen mit MODIFIZERTEN NULL-MODEM-KABEL verbunden werden, da der Anschluss für den GPS-Empfänger am MKT-View II nicht der Pinbelegung einer 'normalen' RS-232-Schnitstelle entspricht !
Hinweis: Da die serielle Schnittstelle in diesen Programmen per Datei-API angesprochen wird, funktionieren die Demos nur, wenn die *erweiterten Script-Funktionen* im Zielgeräte freigeschaltet sind.



Signalfluss zum Testen der seriellen Schnittstelle(n)

Zum Zugriff auf den seriellen Port dienen die Gerätenamen "serial1" bzw. "serial2" (= 2. Port, im MKT-View II ist dies die Schnittstelle für den GPS-Empfänger). Hier ein Auszug aus dem Script-Quelltext im 'GPS-Receiver'-Demo (GpsRcv01.CVT) :

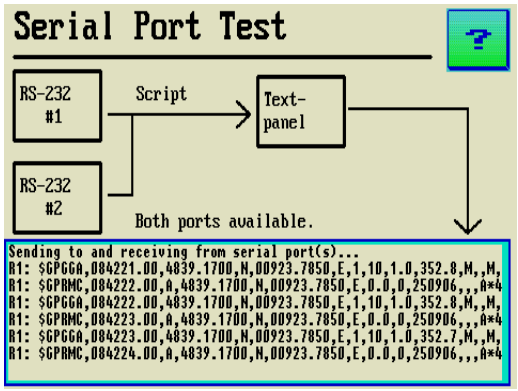
```
// Try to open the second serial port "like a file" .
// In the MKT-VIEW II, device "serial2" is the GPS port.
// Note: Do NOT modify the serial port's baudrate here.
// The system has already set it, according to the
// 'System Setup' / 'GPS Rcv Type' (4k8, 9k6, ...).
hSerial := file.open("serial2");
if( hSerial>0 ) then // successfully opened the GPS port
     display.PortInfo := "Port2";
else // Could not open the 2nd serial port !
     // This happens on a PC (which has no dedicated GPS port).
```

```
        // Try the FIRST serial port instead, with a fixed baudrate:
        hSerial := file.open("serial1/9600");
        display.PortInfo := "Port1";
    endif;
    if( hSerial<=0 ) then // could not open the serial port ?
        print( "\r\nCouldn't open the serial port !" );
        stop;
    endif;

    print( "Reading from device \"",file.name(hSerial),"\"..\r\n" );
    while(1) // endless loop to read and process received data...
        // Read the next bytes from the serial port
        // (not necessarily a complete LINE) :
        temp := file.read_line(hSerial);
        if( temp != "" ) then // something received ?
           // Dump the received character(s) to the text panel..
           if( tscreen.cy >= tscreen.vis_height ) then
               gotoxy(0,1); // wrap around
           endif;
           print( " ", temp );
           clreol;
           print( "\r\n" );
           clreol; // clear the next line (to show last entry)
        else // nothing received now ..
           wait_ms(50); // .. let the display program work
        endif;
    endwhile;
```

Das Script in der Applikation 'Serial Port Test' (programs\script_demos\SerialPt.CVT) verwendet ebenfalls die File-API, um *beide* seriellen Schnittstellen zu öffnen (funktioniert nur bei Geräten mit zwei Schnittstellen, z.B. MKT-View II/III/IV), und liest alles, was von beiden Ports zeilenweise empfangen wird, zeilenweise ein. Mit dem folgenden Code *versucht* das Script, beide Schnittstellen zu öffnen, mit der gleichen fest eingestellten Bitrate:

```
// Try to open the second serial ports "like files" .
// In the MKT-VIEW II, device "serial2" is the GPS port.
// To open the serial port with a fixed baudrate,
// append it after the device name as below .
hSerial1 := file.open("serial1/9600"); // open 1st serial port

hSerial2 := file.open("serial2/9600"); // open 2nd serial port
```

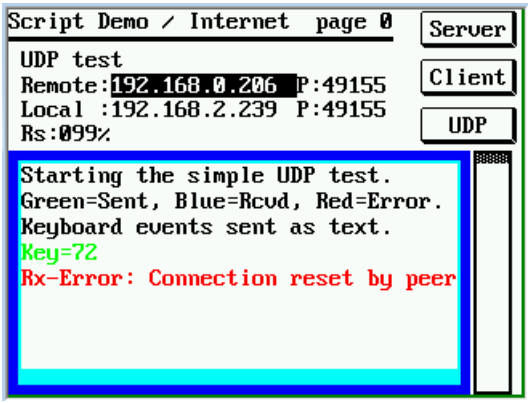
Die empfangenen Strings werden auf einem Text-Panel angezeigt. Die Anzeige sieht dabei z.B. wie folgt aus:



Screenshot 'Serial Port Test' (programs/script_demos/SerialPt.cvt)

script_demos\InetDemo.cvt

Test, Demo, und Beispielprogramm für die Internet-Funktionen (TCP und UDP) in der Script-Sprache.
Enthält einen simplen, TCP/IP-basierten Client und Server. Es ist auch im Simulator (Programmiertool) *lauffähig*, wenn Sie in der Windows-Firewall (bzw. dem "Security Center") dem Programm den Zugriff auf das Internet erlauben.



Screenshot 'UDP Test' (aus programs/script_demos/InetDemo.cvt)

Die drei Schaltflächen in der obenern Hälfte des Displays (siehe Screenshot, oben) dienen zur Auswahl einer Testfunktion:
'Server' startet den per Script implementierten TCP-Server;
'Client' startet eine dazu passende Client-Funktion;
'UDP' startet einen kombinierten Test zum Senden und Empfangen von Datagrammen (UDP).
Beim 'Client'- und 'UDP'-Test werden Tastaturereignisse per Netzwerk gesendet, und empfangene Telegramme auf dem Text-Panel in der unteren Hälfte des Displays angezeigt.
Aus Sicht des Gerätes *gesendete* Nachrichten werden **grün** angezeigt,
empfangene Nachrichten in **blauer** Farbe,
und *Fehlermeldungen* in **roten** Buchstaben auf dem scrollbaren Text-Panel.

Im weiter oben gezeigten Screenshot wurde die Fehlermeldung 'Connection reset by peer' absichtlich hervorgerufen, indem beim UDP-Test beim *Empfänger* (am anderen Ende der LAN-Verbindung) zu spät gestartet wurde. Dadurch hatte der (UDP-) Empfänger noch keinen passenden UDP-Port geöffnet (IP-Adresse gültig aber Port nicht geöffnet), was den Protokoll-Stack (beim MKT-View) zum Senden einer Fehlermeldung (ICMP, s.U.) veranlasst.
Der auf dem PC laufende 'Simulator' (Programmiertool) erhielt danach, beim *nächsten* Aufruf einer Internet-Funktion, vom Netzwerk (Windows Socket Services) den im Folgenden erläuterten, für UDP etwas überraschenden 'Socket'-Fehlercode 'Connection reset by peer'.
Details:

Obwohl UDP ein 'verbindungsloses' Protokoll ist, meldete Winsock 'Connection Refused' oder auch 'Connection reset by peer'.
Im weiter oben gezeigten Screenshot wurde daraufhin die Meldung 'Connection reset by peer' (Verbindung durch den Partner unterbrochen) angezeigt. Grund: Der Partner war zwar (über dessen IP-Adresse) erreichbar, hatte aber noch keinen UDP-Port geöffnet, und der im MKT-View verwendete TCP/IP-Protokollstack (LwIP) sendete daraufhin -korrekterweise- einige Millisekunden später per ICMP die Fehlermeldung 'Port nicht erreichbar'.
Per Wireshark wurde der LAN-Verkehr wie folgt decodiert ("lesbar gemacht"):

```
No.  Time     Source         Destination  Prot. Len Info

26 11.561528 192.168.0.234 192.168.0.206 UDP  50 Src port: 49155  Dst
port: 49155

27 11.561904 MktSyste_12   Broadcast     ARP  60 Who has 192.168.0.234
? Tell 192.168.0.206

28 11.561927 192.168.0.206 MktSyste_12   ARP  42 192.168.0.234 is at
74:27:ea:e2:84:d8

29 11.564618 192.168.0.206 192.168.0.234 ICMP 71 Destination
unreachable (Port unreachable)
```

Da die (ICMP-) Fehlermeldung beim UDP-"Sender" erst Millisekunden später eintraf, hatte das dort laufende Script den Aufruf von inet.send bereits als 'erfolgreich' abgehandelt, und das Problem wurde erst bei einem später folgenden Aufruf einer Internet-Funktion im Script erkennbar (inet.recv), daher die Fehlermeldung 'Rx-Error', obwohl die Ursache nicht der UDP-*Empfang*, sondern der per ICMP als fehlerhaft quittierte *UDP-Sendeversuch* war.

Siehe auch: Testen der Internet / Ethernet - Kommunikation

script_demos\MultiLanguageTest.cvt

Diese Applikation war ursprünglich nur ein Testprogramm für diverse, "sehr spezielle" Funktionen in der Script-Sprache. Sie verwendet die Datei-I/O-Funktionen (file API) um Textdateien aus dem geräteeigenen FLASH-Speicher zu lesen (mit Hilfe des Pseudo-File-

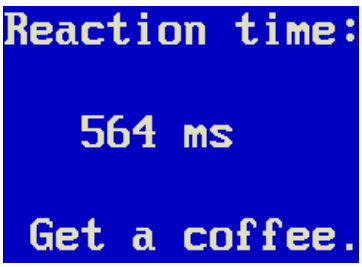
Systems), und eine benutzerdefinierte Funktion namens "GetText", um verschiedene Zeichenketten für die Anzeige per Backslash-Sequenz in die aktuell eingestellte Sprache (Englisch, Deutsch) umzuwandeln.
Sie demonstriert auch, wie Script-Prozeduren per Display-Interpreter-Kommando aufgerufen werden können.

script_demos\OperatorTest.cvt

Diese Applikation enthält ein Script zum Testen neuer Funktionen und Operatoren. Sie wurde während der Software-Entwicklung verwenden, und funktioniert höchstwahrscheinlich nur auf Geräten mit der aktuellen Firmware.

script_demos\ReactionTest.cvt

Ein kleines Beispielprogramm für die Script-Sprache mit einem 'Reaktionstest' (für den Bediener). Das Script wartet zunächst für 0.5 bis 5 Sekunden (gesteuert durch eine Zufallszahl). Danach ändert es die Farbe des Bildschirms, und misst die Zeit bis der Bediener den Farbwechsel per Tastendruck, Touchscreen, oder Drehen/Drücken des Bedienknopfes quittiert.
Abhängig von der gemessenen Reaktionszeit erfolgt danach eine 'Bewertung' wie z.B.:



Screenshot 'Reaktionstest' (programs/script_demos/ReactionTest.cvt)

script_demos\TraceTest.cvt

Diese Applikation enhält ein kurzes Script zum Testen der Trace-Historie, unter Verwendung der in Kapitel 4.10 beschriebenen Funktionen zum Steuern der Trace-Historie.
Darin enthalten:

- Ausgabe eigener Meldungen (per Script) im Trace mit dem Kommando trace.print
- Automatisches Stoppen der Trace-Historie per Script
- Unterdrücken bestimmter CAN-Messages bei der Anzeige in der Trace-Historie (trace.can_blacklist)
- Anzeigen des Inhaltes der Trace-Historie auf einem Text-Panel (per trace.entry[n])
- Löschen der Trace-Historie auf Wunsch des Bedieners (per graphischem Button / Touchscreen)
- Abspeichern der Trace-Historie auf der Speicherkarte des Gerätes

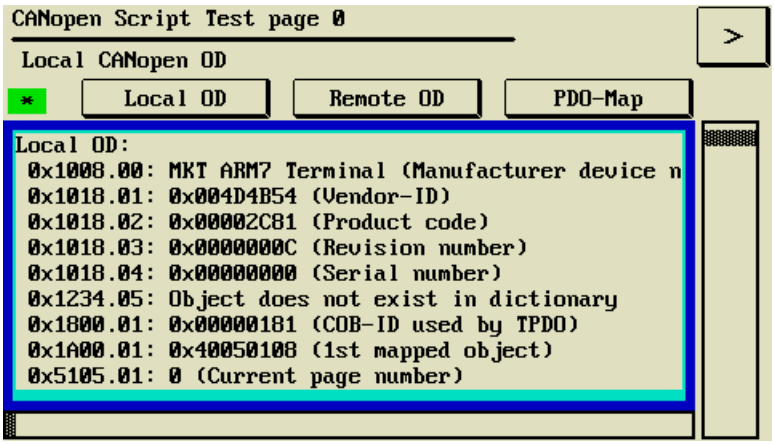
script_demos\CAN_ASC_Logger.cvt

Nur für Gerät mit Speicherkarte und freigeschalteten 'erweiterten Script-Funktionen'.
Das Script in dieser Applikation enthält einen *einfachen CAN-Logger*, der unabhängig vom *bei einigen Geräten* in der Firmware vorhandenen Logger empfangene CAN-Telegramme

direkt im Vector-ASC-Format auf der Speicherkarte speichern kann.
Da bei der Konvertierung nach 'ASCII' (Text) mindestens 64 Bytes pro CAN-Telegramm anfallen (bei 8 Bytes in CAN-Datenfeld), eignet sich dieses Format nicht zum Loggen eines 'kompletten' CAN-Busses mit hoher Bitrate und Buslast. Bei einem Test mit MKT-View IV und einer 'schnellen' Speicherkarte liessen sich maximal 1800 Messages pro Sekunde im Vector-ASC-Format abspeichern.

script_demos\CANopen1.upt

Nur für das 'UPT Programming Tool II' (Programmiertool für Geräte mit CANopen V4) und für Geräte mit integriertem CANopen-Protokoll-Stack.



Screenshot aus dem 'CANopen-Test' (programs/script_demos/CANopen1.upt)

Die Buttons 'Local OD', 'Remote OD', und 'PDO-Map' oberhalb des Text-Panels dienen zum Aufruf verschiedener Test-Routinen im Script. Die Scroll-Balken am rechten und unteren Rand des Text-Panels dienen ggf. zum Verschieben des 'virtuellen' Textbildschirms (per Touchscreen).
Hier eine leicht gekürzte Variante der Prozedur mit dem 'PDO-Mapping Test', mit der der erste Sende-PDO (TPDO1) während der Laufzeit umprogrammiert wird:

```
//---------------------------------------------------------------------
proc TestPdoMapping // demo to 'reprogram' this device's own PDO mapping
  // 'Reprogram' a CANopen slave's PDO mapping table...
  // How to do it CORRECTLY (from CiA 301, V4.2.0, page 142..143):
  // > The following procedure shall be used for re-mapping,
  // > which may take place during the NMT state Pre-operational
  // > and during the NMT state Operational, if supported:
  // >  1. Destroy TPDO by setting bit valid to 1b of sub-index 01h
  // >     of the according TPDO communication parameter.
  // >  2. Disable mapping by setting sub-index 00h to 00h.
  // >  3. Modify mapping by changing the values of the corresponding sub-
indices.
  // >  4. Enable mapping by setting sub-index 00h to the number mapped
objects.
  // >  5. Create TPDO by setting bit valid to 0b of sub-index 01h
  // >     of the according TPDO communication parameter.
```

```
  local dword dwCommParValue; // 32-bit unsigned integer

  cop.error_code := 0; // Clear old 'first' error code (aka SDO abort code)
.
   // If the following CANopen commands work as planned, cop.error_code
remains zero.
   // If something goes wrong, cop.error_code could tell us what, and why
it went wrong.
   // In the original script (in script_demos/CANopen1.upt), it is checked
after each obd-access.
  dwCommParValue := cop.obd(0x1800,0x01); // save original PDO comm.-param
  cop.obd(0x1800,0x01) := dwCommParValue | 0x80000000;  // make 1st TPDO
invalid; set bit 31
  cop.obd(0x1A00,0x00) := 0x00;        // clear TPDO1 mapping table (CiA:
"Disable mapping"..)
  cop.obd(0x1A00,0x01) := 0x40050108;  // 1st mapping entry: map object
0x4005, subindex 1, 8 bits
  cop.obd(0x1A00,0x02) := 0x50010108;  // 2nd mapping entry: map object
0x5001, subindex 1, 8 bits
  cop.obd(0x1A00,0x03) := 0x51050108;  // 3rd mapping entry: map object
0x5105, subindex 1, 8 bits
  cop.obd(0x1A00,0x00) := 0x03;        // enable mapping by setting the
number of mapped objects
  cop.obd(0x1800,0x01) := dwCommParValue & 0x7FFFFFFF; // make 1st TPDO
valid; clear bit 31

  print( "\r\nNew PDO mapping table:\r\n" );
  ShowPdoMap( cTPDO, 1/*PdoNumber*/ );  // show new PDO mapping table (in
CANopen1.upt)
endproc; // TestPdoMapping()
```

script_demos\J1939sim.cvt

Dieses Beispiel simuliert ein Steuergerät, aus dem sich einige Parameter per J1939-Protokoll auslesen lassen, und einen 'Diagnose-Tester', der einige (standardisierte) Parameter J1939 aus einem Steuergerät auslesen kann.

Ein ähnliches Script ist auch für ISO 15765-2 ("ISO-TP") verfügbar.

script_demos\ISO15765sim.cvt

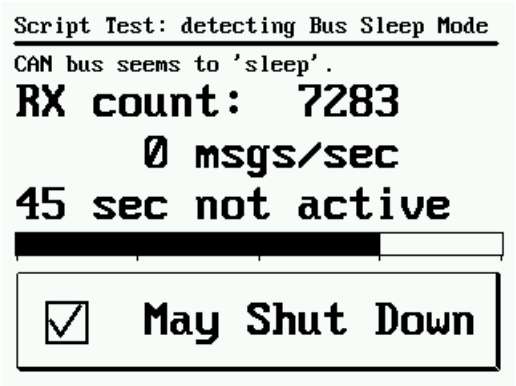
Mit diesem Script versuchte der Autor im Sommer 2015 ein Steuergerät mit ISO 15765-2 (aka "ISO-TP") zu emulieren.
Zum Zeitpunkt der Erstellung dieses Dokumentes (2015) war das Demo-Script vermutlich nicht fehlerfrei, denn für die Entwicklung stand keine geeignete CAN-Testumgebung zur Verfügung (z.B. "echte" ECU mit ISO 15765).

Ein ähnliches Script ist auch für J1939 verfügbar.

script_demos\BusSleepMode.cvt

Dieses Beispielprogramm versucht 'Busruhe' (CAN bus sleep mode) zu erkennen, d.h. Abwesenheit jeglicher CAN-Aktivität.

Dazu wird in einem Timer Event Handler der Zählerstand von CAN.rx_counter() ausgewertet. Ändert sich der Wert innerhalb einiger Sekunden nicht, wird 'Busruhe' signalisiert.
Das MKT-View (II,III,IV) könnte sich daraufhin auch selbst abschalten, was in diesem Demo erst nach 60 Sekunden erfolgt.



Screenshot aus Beispielprogramm 'Bus Sleep Mode',
nach 45 Sekunden *ohne Aktivität* auf dem CAN-Bus

Auszug aus dem Script in BusSleepMode.cvt :

```
var
   int    iPrevCanRxCount;   // CAN-Message-Zählerstand aus der vorherigen
Messung
   int    iCanMsgsPerSecond; // aktuelle CAN-Message-Rate (Messages pro
Sekunde)
   int    iTimeOfNoActivity; // Anzahl Sekunden ohne CAN-Aktivität /
Shutdown-Timer
   int    MayShutDown;       // vom oben abgebildeten Check-Button
gesteuertes Flag
   string Info;
   tTimer Timer1;
endvar;

    ...

setTimer( Timer1, 1000, addr(OnTimer1) ); // Start timer with event
handler, called every 1000 ms
init_done;  // let the system know "we're open for business" (enable event
handlers)

while(1)  // endless loop for the script's MAIN THREAD
  if (iTimeOfNoActivity>60) and MayShutDown then   // no bus activity for
over 60 seconds ?
     system.shutdown;           // turn this device off
  endif;
  wait_ms(50);  // give the CPU to someone else for 50 milliseconds
endwhile; // main thread

func OnTimer1( tTimer ptr pMyTimer )  // periodically called, once per
second
   local int iNewCanRxCount;
```

```
    iNewCanRxCount    := CAN.rx_counter( cPortCAN1 );
    iCanMsgsPerSecond := iNewCanRxCount - iPrevCanRxCount;
    if( iPrevCanRxCount == iNewCanRxCount ) then
      // Arrived here: No bus activity !
      iTimeOfNoActivity := iTimeOfNoActivity + 1;
      Info := "CAN bus seems to 'sleep'.";
    else
      iTimeOfNoActivity := 0;
      Info := "CAN bus is active.";
    endif;
    iPrevCanRxCount := iNewCanRxCount;
    return TRUE;
endfunc; // end OnTimer1
```

script_demos\VT100Emu.cvt

Mit diesem Beispielprogramm kann ein VT100- oder VT52-Terminal emuliert werden.
Die empfangenen Zeichen werden per Script auf einem Text Panel angezeigt.
Geeignet für CAN (mit bis zu 8 Zeichen pro CAN-Telegramm) und für die serielle Schnittstelle (RS-232).
Eine Übersicht der wichtigsten VT100- und VT52-Escape-Sequenzen finden Sie auf der mittlerweile 'online' gestellten "MKT-CD" in Dokument Nr. 85141, VT100/VT52-Emulation für MKT-Geräte .
Aussschnitt aus dem Script in der Applikation 'VT100Emu.cvt':

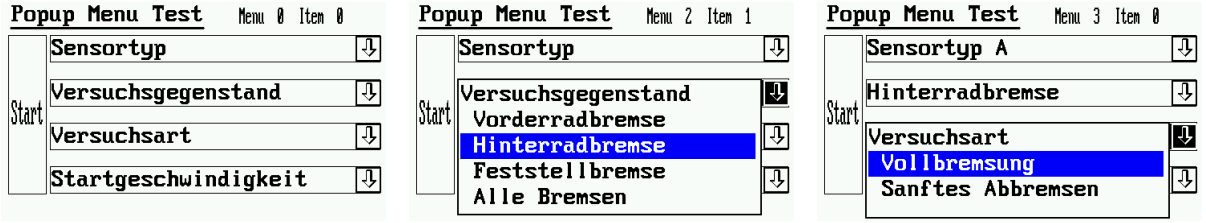
```
hSerialPort := 0; // use the serial port at all ?
if (iSerialBaud>=300) and (iSerialBaud<=115200) then
   // Try to open the serial port "like a file" .
   hSerialPort := file.open("serial1/"+itoa(iSerialBaud) );
endif;            // use the SERIAL PORT (RS-232) ?

 ........

init_done;        // let the system know "we're open for business"

 ........


while( TRUE )     // main loop .. only interrupted by event handlers and
similar
  wait_ms(50);    // give the CPU to whoever-may-need-it .

  if( hSerialPort>0 ) then // successfully opened the serial port (RS-232)
     if( file.read( hSerialPort, sRcvd ) > 0 ) then
        VT100_HandleRcvdString( sRcvd );
     endif;
  endif;

endwhile;         // end of 'endless' script loop
```

Die ebenfalls *im Script* implementierte Prozedur VT100_HandleRcvdString wird nicht nur aus der oben gezeigten Hauptschleife, sondern auch aus einem CAN-Empfangs-Handler

aufgerufen. Das emulierte "VT100-Terminal" lässt sich daher nicht nur für die serielle Schnittstelle, sondern auch als einfache Text-Anzeige *für den CAN-Bus* nutzen.
Der komplette Quelltext des VT100-Emulators würde den Rahmen dieses Dokumentes sprengen; Sie finden ihn in der Datei programs\script_demos\VT100Emu.cvt .

script_demos\popup1.cvt Selbstdefiniertes 'Menü' (pop-up, pull-down)

Dieses Beispiel zeigt selbstdefinierte Pop-Up-Menüs (hier: als Auswahllisten) auf dem Bildschirm an. Der Inhalt wird *zur Laufzeit* per Script erzeugt.
Für die Anzeige der Auswahllisten dient ein Text-Panel mit Rahmen, zum "Aufklappen" der Menüs normale Buttons (Schaltflächen):

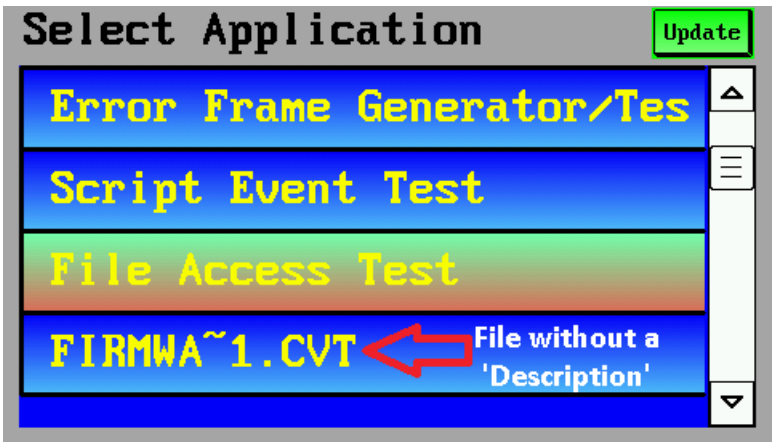


Selbstdefinierte Popup-Menüs (aus programs/script_demos/popup1.cvt)

Um ein Popup-Menü zu öffnen oder schliessen, wird es per display.elem[<Element-Name>].visible sichtbar bzw. unsichtbar gemacht.
Zur Bedienung per Tasten, Encoder (Drehknopf) oder Touchscreen dienen im Script enthaltene Event-Handler.

script_demos\AppSel_1.cvt : 'App(likations) - Selektor'

Dieses Beispielprogramm durchsucht das Wurzelverzeichnis der Speicherkarte nach *.cvt-Dateien (neudeutsch 'Apps'), und zeigt deren *Dateinamen*, oder -wenn vorhanden- die *Datei-Beschreibung* in einer Tabelle an. Der Bediener kann *eine* Datei per Touchscreen auswählen, um sie zu starten.



Screenshot aus dem 'Applikations-Selektor' mit Tabelle zur Auswahl der zu startenden Applikation.

Die Datei in der letzten Zeile enthält keine *Beschreibung*, darum wird dort nur der *Dateiname* angezeigt.

Wenn vorhanden, wird dem Bediener in der Auswahlliste statt des Dateinamens die *Dateibeschreibung* (s.U.) angezeigt. Andernfalls (ohne Dateibeschreibung) wird der normale Dateiname (mit maximal 8 Buchstaben für den Namen + 3 Buchstaben für die Dateinamenserweiterung, hier '.CVT') angezeigt. Der Benutzer wählt die zu startende Applikation per Touchscreen oder Drehknopf aus der Liste aus. In diesem Beispiel besteht die 'Liste' aus einer einspaltigen Tabelle mit vertikalem Scrollbalken.

Durch Antippen des Buttons 'Update' kann die Anzeige des Inhaltsverzeichnisses aktualisiert werden, z.B. wenn in der Simulation im Programmiertool der Inhalt des Simulations-Verzeichnisses geändert wurde, oder wenn eine neue Speicherkarte eingesetzt wurde.
Die per Script gesteuerten Farben des Update-Buttons haben folgende Bedeutung:

Gelb

Das Verzeichnis der Speicherkarte wird momentan eingelesen.

Grün

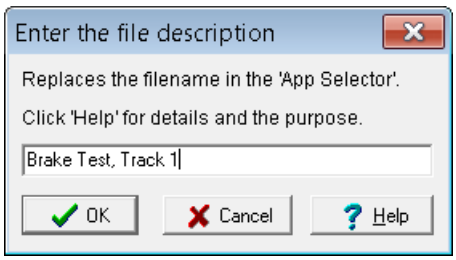
Das Verzeichnis der Speicherkarte wurde erfolgreich eingelesen, und mindestens eine 'nachladbare' Applikation gefunden.

Rot

Das Verzeichnis konnte nicht eingelesen werden, keine Speicherkarte eingesetzt, oder keine geeignete Datei auf der Speicherkarte vorhanden.

Solange keine Speicherkarte eingesetzt ist, oder auf dieser keine *.CVT-Dateien vorhanden sind, blinkt der 'Update'-Button gelb/rot, weil per Timer (im Script von AppSel_1.cvt) alle 200 Millisekunden neu versucht wird, das Verzeichnis einzulesen, bis zum Erfolg.

Die Dateibeschreibung einer UPT- oder CVT-Applikation (im obigen Screenshot z.B. die Zeile mit dem "Error Frame Generator") kann per Programmiertool im Menü 'Datei' .. 'Beschreibung' definiert werden:



Eingabe der 'Beschreibung' einer Applikation im Programmiertool

Details zur Ablage der Beschreibung innerhalb der UPT- bzw. CVT-Datei finden Sie im Handbuch zum Programmiertool .

Das folgende Fragment aus AppSel_1.cvt dient zum Einlesen des Verzeichnisses der Speicherkarte, und zum Extrahieren der Dateibeschreibung. Alle für die Anzeige in der

Tabelle relevanten Informationen werden im selbstdefinierten Array ' MyFileList[]' mit dem Typ tMyFileInfo gespeichert (Deklaration im folgenden Fragment nicht enthalten).

```
//---------------------------------------------------------------------------
---
func ReadDir( string sFileMask )
  // Reads the directory, and stores the result in MyFileList[].
  //   For *.cvt and *.upt files, tries to extract the 'file description'.
  // [in] sFileMask : defines which types of files to list, e.g. "*.*"
  // [return] number of entries found (0=none) .
  local int i,handle,n_entries;
  local string sPath;
  local tDirEntry dir_entry; // directory entry structure used by
directory.read()
  local int fh;
  local string sExtension;
  local string sGarbage, sDescription;
  n_entries := 0;

  // Isolate the 'path' from the search mask:
  i := strrpos( sFileMask, "/" );
  if( i>0 ) then
     sPath := substr( sFileMask, 0, i+1 );
  else
     sPath := sFileMask;
  endif;

  // Isolate the 'extension' from the search mask:
  i := strrpos( sFileMask, "." );
  if( i>0 ) then
     sExtension := substr( sFileMask, i+1, 3 );
  else
     sExtension := sFileMask;
  endif;


  // open, read, and close the DIRECTORY :
  handle := directory.open( sFileMask );
  if( handle>0 ) then    // successfully opened the directory for reading ?
    while( directory.read( handle, &dir_entry ) ) // repeat for all
matching entries...
      if((dir_entry.attributes & (~cFileAttrArch))==cFileAttrNormal)
             && (n_entries<MyFileList.size(0) ) then
         MyFileList[ n_entries ].sFilename := dir_entry.name;
         MyFileList[ n_entries ].sDescription := "";
         MyFileList[ n_entries ].sDate := itoa(dir_entry.year,4) + "-"
                                        + itoa(dir_entry.month,2)+ "-" +
itoa(dir_entry.mday,2);
         MyFileList[ n_entries ].sPath := sPath;
         // Look "into" the file. If it's an UPT or CVT application,
         // it may contain a 'file description' which provides
         // more information for the operator than the 8.3 filename.
         if ( sExtension=="cvt") or ( sExtension=="upt") then
            fh := file.open( sPath + dir_entry.name, O_RDONLY | O_TEXT);
```

```
            if( fh>0 ) then // Successfully opened the file ? Try to
extract its DESCRIPTION !
               if( file.read( fh, sGarbage, "Description=\"",
sDescription,"\"\r\n" ) > 0 ) then
                 MyFileList[ n_entries ].sDescription := sDescription; //
ok, found description
               endif;
               file.close(fh);
            endif;
         endif; // < *.cvt or *.upt ? >
         n_entries := n_entries+1;
      endif;   // < "normal" file ? >
    endwhile; // continue_reading ?
    directory.close(handle); // never forget to close files and directories
!
    return n_entries;
  else   // failed to open the directory, most likely there's no memory
card:
     return -1;
  endif;
endfunc; // ReadDir()
```

Beim Auswählen der zu startenden Datei wird im 'On-Click'-Handler der Tabelle (die in diesem Beispiel als Auswahlliste verwendet wird) lediglich der Index der angeklickten Tabellenzeile in eine Variable umkopiert ('iSelectedItem'). Diese wird in der Hauptschleife des Scripts abgefragt, und die vom Bediener gewählte Datei ("App") per system.exec gestartet:

```
  if( iSelectedItem >= 0 ) then // operator has selected an application.
Launch it.
    system.exec( MyFileList[iSelectedItem].sPath +
MyFileList[iSelectedItem].sFilename );
    // system.exec() should load the selected application INTO RAM(!)
    // and start it. When successful, system.exec() never returns.
    // If we ever get HERE, there's something wrong with the selected app
    // so let the operator select another:
    iSelectedItem := -1;  // "done" (until the user selects another file)
  endif; // iSelectedItem ?
```

Falls in der per App-Selektor nachgeladenen Applikation keine Vorkehrungen enthalten sind, um im Bedarfsfall wieder in den App-Selektor zurückzuschalten (z.B. system.reboot), kann dies notfalls durch Aus- und Wiedereinschalten, oder durch Neustart per Shutdown-Fenster ("Reset") erfolgen:

Screenshot des *in der Gerätefirmware* implementierten 'Shutdown'-Fensters.

script_demos\IncludeTest.cvt : Testprogramm für '#include'

Das Script in diesem kleinen Beispielprogramm verwendet eine Include-Datei, in dem ein Unterprogramm ('SayHello') enthalten ist.
Das Unterprogramm wird aus dem Script (Hauptprogramm) aufgerufen.
Gekürzter Ausschnitt aus dem Quelltext, mit dem vom Compiler eingefügten Text aus der Include-Datei (ähnlich wie im Editor/Debugger grau markiert):

```
// File  :  "IncludeTest.cvt"
#pragma strict
#include "Test.inc"
##begin_include "Test.inc" date=2016-08-04_16:24:10 // DO NOT EDIT THIS
PART !
proc SayHello()
  cls;
  print("Hello world, this is Test.inc speaking.\r\n");
endproc;
##end_include "Test.inc"

init_done;

SayHello(); // Call the procedure implemented in Test.inc
// End of the script's "main part".
```

Um eine der oben aufgeführten Beispiel-Applikationen zu laden, wählen Sie die Funktion 'Datei' .. ' Lade Programm' im Hauptmenü des Programmiertools. Hinweis: Scripte sind lediglich ein Teil einer Applikation, und in der Applikationsdatei (*.cvt bzw. *.upt) enthalten.
Die Beispiele finden Sie im *Unter*verzeichnis 'programs\script_demos' (nicht zu verwechseln mit dem von Windows verwalteten Verzeichnis namens "Programme" / "Program Files" / "Programmi" / etc).

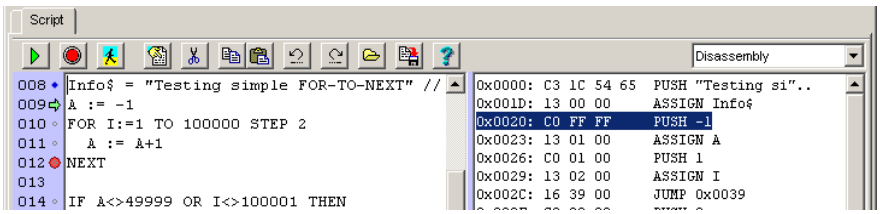
Ende des Kapitels 'Beispiele' - folgen Sie dem Link zur Übersicht.

6. Bytecode

Hinweis:

Dieses Kapitel wird vom Entwickler der Script-Sprache als 'lesenswert' eingestuft, ist aber zum Einsatz der Script-Sprache nicht unbedingt notwendig. Sind Sie daran interessiert, was in der Script-Sprache 'unter der Haube' passiert, lesen Sie bitte weiter...

Um die Geschwindigkeit beim Abarbeiten des Scripts zu beschleunigen, wird der Quelltext vor der Ausführung in einen maschinennahen Binärcode ('Bytecode') umgewandelt.
Der Mikrocontroller kann den Bytecode wesentlich schneller abarbeiten, als den Quelltext 'direkt' zu interpretieren (d.h. ohne Tokenisierung, und ohne Umwandlung der Ausdrücke in RPN).



Screenshot von Quelltext-Editor (links) mit Disassembler-Anzeige (rechts)

Tipp:

Der Bytecode kann (in disassemblierter Form) im Programmiertool sichtbar gemacht werden.

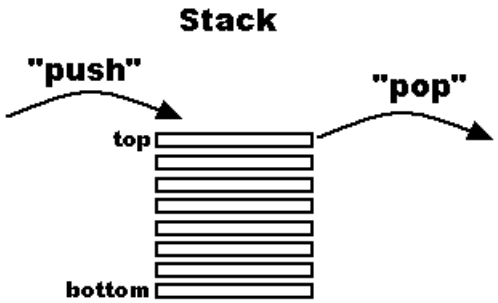
Klicken Sie dazu in der Toolbar des Script-Editors auf den links abgebildeten Button ('Chip'-Symbol).

1 6.1 Übersetzen des Quelltextes in Bytecode ("Kompilieren")

Dieses Kapitel wurde bislang noch nicht ins Deutsche übersetzt. Verwenden Sie bei Interesse bitte das englischsprachige Originaldokument.

2 6.2 Der Stack

Der Stack (auf Deutsch "Stapel", nicht "Keller" ... denn 'Keller' = engl. cellar, basement trifft die Bedeutung eines Stacks als 'Stapel' wesentlich schlechter !).



Verwendung des Stacks durch das Script-Laufzeitsystem

Der Stack ist ein klassischer LIFO-Speicher (Last-In / First-Out). Wird ein neuer Wert auf den Stack 'gepusht', bedeutet dies, der Wert wird auf der Spitze des Stacks **GESTAPELT (STACKED)** (und keineswegs darunter "**eingekellert**"), vergleichbar mit dem oberen Blatt auf einem

Papierstapel (siehe Abbildung). Nur das oberste Element auf dem Stack kann vom Stapel gelesen werden ("pop" : Auslesen und Entfernen des Elementes auf der Spitze des Stacks). Dies sind die einzigen Operationen, die mit dem 'klassischen' Stack erlaubt sind. Um den Stack auch als Speicher für lokale Variablen verwenden zu können, sind darüberhinaus noch weitere Operationen implementiert (Stichwörter 'stack frame' und 'base pointer').

Der Inhalt des Stacks kann zu Testzwecken im Programmiertool (Simulator) angezeigt werden. Details unter 'Stack Display' . Beim intensiven Einsatz von Unterprogrammen, speziell beim Einsatz von Rekursion, sollten Sie den Stack-Verbrauch im Script im Auge behalten.

In der Script-Sprache dient der Stack nämlich sowohl zum Berechnen von Ausdrücken (die vom Compiler in RPN umgewandelt wurden), zum Speichern von Adressen im Programmspeicher (beim Aufruf von Funktionen und Prozeduren), und als temporärer Speicher für lokale Variablen. Lokale Variablen werden auf dem Stack angelegt, indem eine Funktion zunächst eine der Anzahl von Variablen entsprechende Menge von Dummy-Werten (i.A. Null) auf dem Stack abgelegt wird. Die Adresse der ersten lokalen Variablen (auf dem Stack) wird danach in einem üblicherweise 'Base Pointer' genannten Register abgelegt. Mit Hilfe des Base-Pointers wird dann innerhalb der grade aktiven Funktion auf die lokalen Variablen zugegriffen, wodurch ein Teil des Stacks (der sog. 'Stack Frame') nicht mehr als LIFO, sondern als Speicher mit wahlfreiem Zugriff verwendet werden kann. Details dazu im nächsten Kapitel.

2.1 6.2.1 Stack Frames (für Funktionsargumente und lokale Variablen)

Dieses Kapitel wurde bislang noch nicht ins Deutsche übersetzt. Verwenden Sie bei Interesse bitte das englischsprachige Originaldokument.

3 6.3 Bytecode-Spezifikation

Dieses Kapitel wurde ebenfalls noch nicht ins Deutsche übersetzt. Verwenden Sie bei Interesse bitte das englischsprachige Originaldokument.

7. Letzte Änderungen (nach Datum)

Hinweis: Wurde dieses Dokument von einem lokalen Speichermedium geladen,
dann existiert vermutlich eine aktuellere Historie online !
Die oben verlinkte *Online*-Revisions-Historie bezieht sich auf das gesamte System (inkl. Programmiertool und Gerätefirmware). Die folgende, nach Datum sortierte Auflistung bezieht sich dagegen nur auf die *Script-Sprache*.

2021-05-05
Entfernung der 'figure'-Tags um die Umwandlung von HTML nach PDF zu verbessern.
Hinweise zum CAN-Simulator eingefügt.

2020-10-12
Neues Kommando: file.delete .

2019-12-17
Neue Kommandos und Datentypen für Text-Panels .

2019-02-22
Neue Datentypen (z.B. bool, tColor) und automatisch erzeugte Typ-Konversionen .