C-Programmierung für Echtzeitanwendungen unter mCAT

Was ist mCAT?

- mCAT ist ein Echtzeitbetriebssystem für Mikrocontroller der Toshiba TLCS900-Familie (die Unterstützung anderer Prozessoren ist in Planung)

Ein Echtzeitkern wie mCAT wird hauptsächlich eingesetzt, um die Modularisierung von Programmen zu erleichtern. Im Sinne der leichteren Wartbarkeit sollte man versuchen, Aufgabengebiete einer Anwendung, die sich kapseln lassen, auch als separates Modul zu schreiben und von den anderen Teilen der Anwendung aufrufen zu lassen. Hierzu bieten sich zunächst alle hardwareabhängigen Probleme an, beispielsweise der Zugriff auf digitale E/A oder auf serielle Schnittstellen. Weiter kann man Programmteile wie Regler, Ablaufsteuerungen, Konverter oft mit wenig Mehraufwand isolieren, was bei der Fehlersuche eine große Erleichterung ist. Ausserdem werden die Module damit automatisch bei einem späteren Projekt wiederverwendbar bzw. austauschbar, wenn z.B. der Regelalgorithmus oder das Schnittstellenprotokoll auf Kundenwunsch geändert werden muss.

Echtzeit

Bei der industriellen Elektronik haben wir darüberhinaus häufig mit Aufgaben zu tun, die abhängig sind von der Zeit oder von äußeren Ereignissen. Ein Echtzeitkern wie mCAT bietet dazu die Möglichkeit, die Programmmodule, die wir aus "pflegerischen" Gründen ohnehin separiert haben, zeit- oder ereignisgesteuert zu starten. Das Betriebssystem regelt für uns die Kommunikation zwischen den Modulen und löst Konflikte auf, wenn zwei Module gleichzeitig laufen wollen (was bei nur einem Prozessor ja nicht geht).

Tasks und Interrupttreiber

Dazu unterscheidet mCAT die Module in Tasks und Interrupttreiber. Ein Interrupttreiber dient der schnellen Reaktion auf externe Ereignisse (zählt z.B. die Anzahl der Zähne eines Zahnrads, die an einem Initator vorbeigeführt werden), darf aber nur ganz kurz sein, um andere Interrupts auch zum Zuge kommen zu lassen. Eine Task muss so geschrieben werden, dass sie eine Aktion ausführt und dann in einen Wartezustand geht, aus dem sie durch eine andere Task, die die Zeitsteuerung vornimmt, oder durch ein asynchrones Ereignis wieder geweckt wird. Für die Zeitsteuerung liefert mCAT einen vorgefertigten Interrupttreiber "Ticker", den man mit einmaligen oder auch periodischen Weckaufträgen (z.B. alle 50ms) betrauen kann.

Nachrichten

Die Kommunikation findet bei mCAT mit Nachrichten (messages) statt. Ein Interrupttreiber schickt eine Nachricht an eine Task (z.B. wenn er 25 Zähne gezählt hat) und die Tasks untereinander schicken sich ebenfalls Nachrichten. Dem Ticker schickt man eine Nachricht, wie oft man geweckt werden will und der schickt dann z.B. jede Sekunde eine Wecknachricht. Dabei gibt man der Nachricht eine Priorität, die darüber entscheidet, wer gewinnt, wenn gleichzeitig eine andere Task aktiv wird oder noch läuft (wenn die Nachricht von einem Interrupttreiber kommt, der eine Task ja unterbricht).

Das Namensverzeichnis

Damit alle Module beliebig ausgetauscht werden können, ist es nicht zulässig, von festen Speicheradressen etc. auszugehen. Daher gibt es eine zentrale Instanz bei mCAT, bei der man unter Angabe eines Modulnamens eine Referenz auf das Modul anfragen kann. Die fiktive Reglertask spricht man also nicht direkt an, sondern über die Referenz, die man vom Namensverzeichnis erhält, wenn man nach z.B. "Elzet/Regler" fragt.

Details zu allen diesen Themen wollen wir später noch angehen, zuerst jetzt aber mal in die Praxis - zum ersten kleinen Programm unter mCAT2 - Hello World

Die Praxis: "Hello World"

Erstes Programm für TSMCPU900 mit mCAT.2

Dieses Miniprogramm schreibt "hello world" auf die serielle Schnittstelle. In fast allen Büchern ist es das erste Programm, also fangen wir hier auch so an. Hier kommt es hauptsächlich darauf an, wie man überhaupt ein Programm für mCAT.2 erstellt. Ein Programm, das "Hallo Welt" auf der Konsole ausgibt, eignet sich weder zur Modularisierung, noch würde man es üblicherweise auf ein Echtzeitbetriebssystem aufsetzen. Dennoch muss man diesen ersten Schritt tun, um nicht gleich von der Komplexität paralleler Tasks erschlagen zu werden. Leider müssen wir aber das ganze Drumherum der Taskerzeugung etc. auch für ein "Hallo Welt" erzeugen, damit mCAT damit regulär umgehen kann.

MS-Windows sei als Entwicklungsumgebung vorausgesetzt, es geht aber auch mit MS-DOS. Die Installation erfolgt durch das Programm auf der mCAT.2-CD, dabei wurde die TSMCPU900 als Standardbaugruppe bestimmt.

makefile erzeugen

Bei einem C-Cross-Compiler ist es vom Editieren des C-Programms bis zum Ausführen auf dem Zielrechner ein weiter Weg: compilieren, assemblieren, linken, in Hex-Download-File umwandlen, runterladen, starten. Einen Teil dieser Arbeit nimmt uns ein Hilfsprogramm namens VMAKE ab, das mit Hilfe der Konfigurationsdatei namens "makefile" Compiler, Assembler und Linker mit den richtigen Optionen füttert.

Am besten erstellt man sich aber zunächst für jedes Programm ein eigenes Verzeichnis, also auch für unser "Hello World":

```
c:\>cd mcat.2\cc

c:\mcat.2\cc>mkdir hello

c:\mcat.2\cc>cd hello
```

Da wir unter MS-DOS arbeiten und nur 8+3 Zeichen zur Verfügung haben, nennen wir die Dateien hello.*
Dann kopiert man sich ein beliebiges makefile aus den Beispielverzeichnissen, wir nehmen es aus c:\mcat.2\cc\ticker\ (am besten mit dem Explorer).
Dieses wird nun editiert, zum Beispiel mit edit makefile.
Die ersten Zeilen sind nur Kommentare, diese sind mit einem # gekennzeichnet.
Die nächsten beiden Zeilen sind wichtig:

```
PROJECT = hello

TARGET = std_ram
```

Hiermit sagt man dem makefile wie die Dateien für das Projekt heißen (PROJECT = hello) und wo man das Programm ablegen will. In diesem Beispiel im Standard RAM (TARGET = std_ram) der TSMCPU900.

.c File erzeugen

MCAT kann mit Tasks in Flash-Eprom und im RAM umgehen, wenn wir auch zunächst (in der Testphase) unser Programm nur in das RAM laden. Damit eine Task aber auch im EPROM beim Systemstart gefunden wird, braucht sie einen Vorspann, der Informationen über Stackgröße, Startpriorität, Name etc. enthält. Dieser "Initial Module Descriptor" oder kurz IMD, wird von einem Hilfsprogramm gleichen Namens erzeugt, dem man den Namen der zu erzeugenden C-Datei als Parameter anfügt.

Im Unterverzeichnis ..\mcat.2\cc\hallo geben wir also am DOS-Prompt ein:

```
c:\mcat.2\cc\hello>imd hello.c
```

Es erscheint ein Dialogfensterchen, in das der Name der Task einzutragen ist, unter der sie später in der mCAT.2 Tasklist auftaucht, also sorgfältig wählen. Hier empfiehlt es sich, seinen Firmennamen und mit "/" abgetrennt, den Modulnamen zu verwenden, für unser Beispiel verwenden wir **first/hello** .
Damit wird die Datei **HELLO.C** erzeugt, die wir nun für unser Programm erweitern.

HELLO.C editieren

Die Datei HELLO.c wird nun mit einem Texteditor geöffnet. Man kann grob 5 Bereiche erkennen:

"include"
Hier werden die C-typischen "Header-Files", die in ".h" enden und die die Aufrufprototypen externer Module enthalten, in die aktuelle Datei eingeschlossen.

"define"
Vorbesetzung von Konstanten, die weiter unten verwendet werden. Hier sind Stackgrösse etc. zu ändern, wenn die Vorgabewerte unzureichend sind, für unsere Testprogramme müssen hier keine Änderungen vorgenommen werden.

"Header"
Der auf den "define"s basierende eigentliche Header-Code, beginnend mit 55AAh oder 21930 dezimal, das ist der Code für den Modulanfang, nach dem mCAT im Speicher sucht. Damit zufällige Vorkommen nicht fehlinterpretiert werden, gibt es am Ende des Headers eine Prüfsumme.

"Init"
Anwendercode, der einmal bei Start der Task ausgeführt wird, ein gut geeigneter Platz für z.B. Hardwareinitialisierungen. Zumindest muss man sich in dieser Phase beim mCAT anmelden und sich eine Referenz auf sich selbst geben lassen. Der dazu nötige Code ist bereits vorhanden - die Selbstreferenz wird der Variable "Self" durch die Funktion "TaskStartup" zugewiesen. Die Lage zwischen Protect() und Unprotect() verhindert, dass die Funktion unterbrochen wird.

"Main"
Der Bereich für das Haupt-Anwenderprogramm. Hier startet man fast immer mit "loop {...}" eine endlose Schleife, die mit dem Warten auf eine hereinkommende Nachricht (WaitMsg) beginnt, da es in Prozesssteuerungen wenig Sinn macht, Tasks immer wieder neu zu erzeugen oder zu entfernen.

Nun editieren wir die Datei wie folgt:
In die 3.Zeile schreiben wir:

```
#include <simpleio.h>
```

Das bedeuted, daß wir die Header-Datei simpleio.h benutzen wollen. Dort stehen die Funktionen drin, die uns den Zugriff auf die seriellen Schnitstelle vereinfachen.

Nun schreiben wir (endlich) unser Hauptprogramm und zwar unter TaskMain **HINTER** die "{" Klammer und **VOR** "TaskDelete" (etwa Zeile 66) :

```
SIOWrStr(0,"hello world");
```

Zur bessern Übersicht hier die komplette TaskMain:

```
void TaskMain ()

{
        SIOWrStr(0,"hello world\n");

    TaskDelete(Self);
```

```
}
```

Mit SIOWrStr(0,"hello world\n"); sagen wir dem Compiler, dass wir den Text "hello world" (mit Zeilenumbruch (\n) auf die Schnittstelle Seriell 0 schreiben wollen.
Einen Überblick über alle Funktionen der simpleio.h gibt es HIER.
Gut, nun schliessen wir den Editor und jagen das Ganze durch den Compiler ;-)

Compilieren

Das Tool **vmake** nimmt uns hier viel Arbeit ab (siehe makefile erzeugen). Einfach:

```
c:\mcat.2\cc\hello>vmake -r
```

aufrufen. Der Parameter bewirkt **-r**, daß HELLO.C auf jeden Fall neu compiliert wird. Tauscht man nämlich nur einen Buchstaben in HELLO.C, sieht vmake keine Änderung (in der Dateigröße) und compiliert nicht neu.
Es sollten KEINE Fehler (No error) auftreten.

Am Ende zeigt vmake eine Tabelle an:

```
CPU  : TLCS900     Date : Mon May 22 19:15:10 2000
Input  file name : hello.abs
Output file name : hello.SHX   *** Motrola S format***


Section name        Start address      End address
------------        -------------     -------------
    .data             0x00402000        0x00402056
    .text             0x00402057        0x00402254
 .libltxt             0x00402255        0x00402308


COFF Convert end,No error
```

Was sagt uns das ?

Unser Ausgangsfile heißt **hello.SHX** (Output file name : hello.SHX *** Motrola S format***),
und unserer compiliertes Programm liegt im RAM von **0x00402000** bis **0x00402056** (.data 0x00402000 0x00402056).

Die Anfangsadresse **0x00402000** sollten wir uns merken, die brauchen wir gleich.

Download und ausführen

Nun verbinden wir eine serielle Schnitstelle des PCs mit **SER0** der TSMCPU900 mit einem seriellen Kabel.
Auf der TSMCPU900 muß der Terminator gesteckt werden, sonst befindet sich die TSMCPU900 im Bootmon und kann das Programm nicht ausführen!!!!

Nachdem wir dem Windows-Terminalprogramm von mCAT.2 (**wLgo**) die richtige Schnitstelle eingestellt haben, versorgen wir die TSMCPU900 mit Spannung (+24V). Nun sollten einige Meldungen auftauchen und der Prompt (2+>) sichtbar sein.
Wenn NICHT, stimmt etwas mit der seriellen Kommunikation nicht, also mal am Kabel wackeln oder so ;-)
Es kann auch sein, dass Ihre PC-Schnittstelle keine FIFOs hat, dann ist die Checkbox bei den erweiterten Schnittstellen-Einstellungen abzuwählen.

Mit /Datei/Datei herunterladen (oder F3) wählen wir die Datei aus, die wir downloaden wollen.
Es ist **hello.shx**.

Nachdem die Datei heruntergeladen wurde, geben wir am Prompt:

```
init 402000
```

ein (der Startwert, den wir nach dem Compilieren von vmake bekommen haben).

Und unser Text erscheint *freu*.

Das wars (fürs erste)!!!

Tipps

Damit das Programm im ROM und nicht im RAM steht, ändern wir im makefile die Zeile:

```
TARGET = std_ram
```

in

```
TARGET = std_rom
```

Nun führen wir wieder vmake aus, setzen die TSMCPU900 zurück (in wLgo *reset* eingeben) und laden das neue hello.shx runter.
Wir geben aber jetzt

```
init 800000
```

ein. Nach einem erneuten Reset der TSMCPU900 stellen wir fest, dass unser Programm im ROM liegt und bei jedem Start ausgeführt wird!!!!
Damit kann man sich zum Beispiel eine persönliche Begrüßung schreiben ;-))

Möchte man aber die **automatische Ausführung beim Start VERBIETEN**, muß man nur in der Zeile:

```
21930,     /* IMD.Pattern */
```

in der HELLO.C Datei den Wert **22015** eintragen. Hiermit liegt das Programm unter 800000, wird aber nicht beim Start ausgeführt.
Möchte man das Programm später doch automatisch starten lassen, gibt man im wLgo Prompt folgendes ein:

```
val 800000
```

Möchte man das Programm komplett **wieder löschen** gibt man am wLgo Prompt folgendes ein:

```
purge 800000
```

```
delpage 0
```

```
reset
```

Beim Neustart wird nun der Bereich KOMPLETT gelöscht, weiteres steht im "mCAT 2.06 & BOOTMON 1.02" Handbuch.

Autoren: Markus Ungermann, ELZET80, ungermann@elzet80.de ; Walter Giesler, ELZET80, giesler@elzet80.de
Erstellt: 2000-05-23
Last Update:2000-06-07
Status: Alpha

```
/*----[InitialTaskDescriptor]----------------------------------------------*/
#include <mcat.h>
#include <vtype.h>
#include <simpleio.h>

#define Version      '1'
#define Release      '0'
#define Interim      '0'
#define Priority     200
#define Mode         MODE_TASK
#define ID           255
#define Stack        1024
#define Heap         0
#define TaskId       short

#ifdef __STDC__
  extern void __cstart(IMD *imd);
  PUBLIC void TaskInit(IMD *imd);
  PRIVAT void TaskMain();
  PUBLIC TaskId Self;
#define CONST        const
#else
  PRIVAT void TaskInit();
  PRIVAT void TaskMain();
  PRIVAT TaskId Self;
#define CONST
#endif

#define ip(p)        (void(*)())(p)

PRIVAT IMD CONST task_imd = {
    21930,                  /* IMD.Pattern */
#ifdef __STDC__
    ip(__cstart),           /* IMD.Init */
#else
    ip(TaskInit),           /* IMD.Init */
#endif
    ip(TaskMain),           /* IMD.PC */
    Stack,                  /* IMD.StackSize */
    Heap,                   /* IMD.StackSize */
    0x112d2d11,             /* IMD.Build */
    Priority,               /* IMD.Priority */
    Mode,                   /* IMD.Mode */
    ID,                     /* IMD.ID */
    Version,                /* IMD.Version : VERSION */
    '.',                    /* ! */
    Release,                /* IMD.Version : RELEASE */
    Interim,                /* IMD.Version : INTERIM */
    '\0',                   /* IMD.Version */
    "elzet/hello\0\0\0\0\0",        /* IMD.Name */
    24811                   /* IMD.Check */
};

void TaskInit (imd)
IMD *imd;
{
    short error;

    Protect();
    Self = TaskStartup(imd,FromTop,&error,0);
    UnProtect();
}

void TaskMain ()
{                           SIOWrStr(-1,"Hello wie gehtïs ?");
    TaskDelete(Self);
}
/*----[END]-----------------------------------------------------------------*/
```

Programmbeispiele mit der seriellen Schnittstelle

Hier möchte ich einige Beispiele zur Benutzung der seriellen Schnitstelle geben. Ich gebe hier nur den relevanten Teil des C-Programmes an, wie man das Programm erzeugt oder compiliert, steht ausführlich im hello world Beispiel.

In der simpleio.h stehen alle notwendigen Funktionen.

Werteausgabe

Ausgabe eines Wertes (celsius) auf Schnittstelle 0 ohne führende Nullen:

```
int celsius;
celsius = 1;

SIOWrDecShort (0, celsius, 0);
```

Warten auf Tastendruck

Warte solange bis eine beliebige Taste gedrückt wird:

```
while (!SIOkbhit(0))
                  {
                                 /* Warte */
                   };
```

Zeichenabfrage

Wartet auf Zeicheneingabe von Schnittstelle 0 und gibt dann das Zeichen auf Schnittstelle 0 aus:

```
char rdc;

rdc = SIORdChar(0);
SIOWrChar(0, rdc);
```

mCAT 2 Bibliothek : simpleio.h

Der **_p1 Parameter** steht für:

```
-1  =  Sysmon Schnittstelle
 0  =  Seriell 0
 1  =  Seriell 1
 ...
   solange Schnittstellen vorhanden
```

Fast jede Funktion hat auch einen Rückgabewert. Wenn kein Wert zurückgegeben wird, ist der Rückgabewert TRUE, wenn die Schnittstelle gefunden wurde und FALSE, wenn die definierte Schnittstelle auf diesem System nicht vorhanden ist.

SIOWrLn(_p1)
int SIOWrLn(word chan);
Erzeugt Zeilenumbruch auf Schnittstelle
_p1 = Schnittstelle *Typ: word*

SIOWrDecShort(_p1,_p2,_p3)
int SIOWrDecShort(word chan, short val, word lead);
Schreibt Wert als kurzen Dezimalwert auf Schnittstelle
_p1 = Schnittstelle
_p2 = dezimalwert *Typ: short*
_p3 = Führende Nullen (0=Keine führende Nullen 1=Mit führenden Nullen)
Beispiel:

```
SIOWrDecShort(0, celsius, 0);
```

Schreibt den Wert von "celsius" ohne führende Nullen auf Schnittstelle Seriell 0.

SIOWrDecLong(_p1,_p2,_p3)
int SIOWrDecLong(word chan, long val, word lead);
Schreibt Wert als langen Dezimalwert auf Schnittstelle
_p1 = Schnittstelle
_p2 = dezimalwert *Typ: long*
_p3 = Führende Nullen (0=Keine führende Nullen 1=Mit führenden Nullen)
Beispiel:

```
SIOWrDecLong(0, celsius, 0);
```

Schreibt den Wert von "celsius" ohne führende Nullen auf Schnittstelle Seriell 0.

SIOWrDecWord(_p1,_p2,_p3)
int SIOWrDecWord(word chan, word val, word lead);
Schreibt Wert als kurzen positiven Dezimalwert auf Schnittstelle
_p1 = Schnittstelle
_p2 = dezimalwert *Typ: word*

_p3 = Führende Nullen (0=Keine führende Nullen 1=Mit führenden Nullen)
Beispiel:

```
SIOWrDecWord(0, celsius, 0);
```

Schreibt den Wert von "celsius" ohne führende Nullen auf Schnittstelle Seriell 0.

SIOWrDecLWord(_p1,_p2,_p3)
int SIOWrDecLWord(word chan, lword val, word lead);
Schreibt Wert als langen positiven Dezimalwert auf Schnittstelle
_p1 = Schnittstelle
_p2 = dezimalwert *Typ: long word*
_p3 = Führende Nullen (0=Keine führende Nullen 1=Mit führenden Nullen)
Beispiel:

```
SIOWrDecLWord(0, celsius, 0);
```

Schreibt den Wert von "celsius" ohne führende Nullen auf Schnittstelle Seriell 0.

SIOWrHexByte(_p1,_p2)
int SIOWrHexByte(word chan, byte val);
Schreibt Wert als kurzen HexaDezimalWert (2stellig) auf Schnittstelle
_p1 = Schnittstelle
_p2 = Hex - wert 2 stellig *Typ: byte*
Beispiel:

```
SIOWrHexByte(0, 10);
```

Schreibt den Hex-Wert von Dezimal 10 (= Hex 0a) auf Schnittstelle Seriell 0.

SIOWrHexWord(_p1,_p2)
int SIOWrHexWord(word chan, word val);
Schreibt Wert als positiven kurzen HexaDezimalWert (4 stellig) auf Schnittstelle
_p1 = Schnittstelle
_p2 = Hex - wert 4 stellig *Typ: word*
Beispiel:

```
SIOWrHexWord(0, hexi);
```

Schreibt den Wert von "hexi" (000a) auf Schnittstelle Seriell 0.

SIOWrHexLWord(_p1,_p2)
int SIOWrHexLWord(word chan, lword val);
Schreibt Wert als positiven langen HexaDezimalWert (8 stellig) auf Schnittstelle
_p1 = Schnittstelle
_p2 = Hex - wert 8 stellig *Typ: long word*
Beispiel:

```
SIOWrHexLWord(0, hexi);
```

Schreibt den Wert von "hexi" (0000000a) auf Schnittstelle Seriell 0.

SIOWrStr(_p1,_p2)

int SIOWrStr(word chan, char *str);
Schreibt String auf Schnittstelle
_p1 = Schnittstelle
_p2 = String *Typ: char*
Beispiel:

```
SIOWrStr(0,"Das ist Schnittstelle Seriell 0\n");
```

Schreibt "Das ist Schnittstelle Seriell 0" auf Schnittstelle Seriell 0 und macht dann einen Zeilenumbruch.

SIOWrChar(_p1,_p2)
void SIOWrChar(word chan, char c);
Schreibt Wert als Character (Angabe in dezimal) auf Schnittstelle
_p1 = Schnittstelle
_p2 = Dezimal des ASCII-Zeichens *Typ: char*
Beispiel:

```
SIOWrChar(0,77);
```

```
oder SIOWrChar(0,'M');
```

Schreibt ein "M" auf Schnittstelle Seriell 0.

SIORdChar(_p1)
char SIORdChar(word chan);
Liest Wert als Character von Schnittstelle
_p1 = Schnittstelle
Typ: char

Evtl. mit SIOkbhit kombinieren.
Beispiel:

```
rdc = SIORdChar(0);
```

Weist "rdc" das von Schnittstelle Seriell 0 gelesene Character zu.

ACHTUNG!! SIORdChar wartet, bis es ein Zeichen liest!! In Echtzeitsystemen ist dies besonders zu beachten!!

SIOkbhit(_p1)
int SIOkbhit(word chan);
Liest von Schnittstelle, wenn Taste gedrückt wird.
_p1 = Schnittstelle
Beispiel:

```
while (!SIOkbhit(0))
                {
                        /* Warte */
                };
```

Solange auf Schnittstelle Seriell 0 keine Zeichen kommen, wartet das Programm.

SIODumpHex(_p1,_p2,_p3)
int SIODumpHex(word chan, lword addr, byte *data);

Schreibt den Speicherinhalt in der Länge von 16 Bytes in der Form: Fikt.Adresse Daten(hex), Daten(hex), (16x)..., ASCII-Darstellung
(00800000 AA 55 89 00 80 00 D0 00 80 00 00 04 00 00 00 00 * .U.....)
auf die Schnittstelle.
_p1 = Schnittstelle
_p2 = in erster Spalte angezeigte fiktive Adresse (hex) *Typ: long word*
_p3 = Zeiger (auf Daten vom Typ Byte) auf wirkliche Startadresse *Typ: byte*
Beispiel:

```
SIODumpHex(0, 800, 80800);
```

Zeigt den Inhalt der Speicherstelle 0x80800 und folgende über Seriell 0 an, stellt es aber so dar, als wäre es Speicherstelle 0x000800.
Parameter _p2 ermöglicht eine von der tatsächlichen Position unabhängige Darstellung für Debugzwecke.

---

simpleio.h

```
/*----[LIBDEF SimpleIO]---------------------*/

/* **** LIB [SIO] */

#define SIOWrHexLWord(_p1,_p2)\

                (int) __exec(0x400000l,(word) (_p1),(lword) (_p2))

#define SIOWrHexWord(_p1,_p2)\

                (int) __exec(0x400004l,(word) (_p1),(word) (_p2))

#define SIOWrHexByte(_p1,_p2)\

                (int) __exec(0x400008l,(word) (_p1),(byte) (_p2))

#define SIOWrDecWord(_p1,_p2,_p3)\

                (int) __exec(0x40000cl,(word) (_p1),(word) (_p2),(word) (_p3))

#define SIOWrDecLWord(_p1,_p2,_p3)\

                (int) __exec(0x400010l,(word) (_p1),(lword) (_p2),(word) (_p3))

#define SIOWrDecShort(_p1,_p2,_p3)\

                (int) __exec(0x400014l,(word) (_p1),(short) (_p2),(word) (_p3))

#define SIOWrDecLong(_p1,_p2,_p3)\

                (int) __exec(0x400018l,(word) (_p1),(long) (_p2),(word) (_p3))

#define SIODumpHex(_p1,_p2,_p3)\

                (int) __exec(0x40001cl,(word) (_p1),(lword) (_p2),(byte*) (_p3))

#define SIOWrLn(_p1)\

                (int) __exec(0x400020l,(word) (_p1))

#define SIOWrStr(_p1,_p2)\

                (int) __exec(0x400024l,(word) (_p1),(char*) (_p2))

#define SIOWrChar(_p1,_p2)\
```

```
                (void) __exec(0x400028l,(word) (_p1),(char) (_p2))
#define SIOkbhit(_p1)\
                (int) __exec(0x40002cl,(word) (_p1))
#define SIORdChar(_p1)\
                (char) __exec(0x400030l,(word) (_p1))
```

Autor: Markus Ungermann ELZET80 ungermann@elzet80.de
Erstellt: 2000-05-22
Last Update: 2000-05-29
Status: Beta

Programmbeispiele mit ExpressIO

Hier möchte ich einige Beispiele zur Benutzung von ExpressIO geben. Ich gebe hier nur den relevanten Teil des C-Programmes an, wie man das Programm erzeugt oder compiliert, steht ausführlich im hello world Beispiel.

Speziell zu ExpressIO weise ich auf das Handbuch **mCAT ExpressIO for TSM** von Volker Goller hin. Es ist gut verständlich und ein Beispiel gibt es auf den mCAT Doku html Seiten.

Grundgerüst

Damit ExpressIO funktioniert, benötigt man folgende Header-Dateien:

```
#include <express.h>
#include <ioman.h>
```

Dann definiert man seine Namen:

```
IOOBJECT dig_1, rel_1;
```

Hiermit legt man die Variablen `dig_1` und `rel_1` fest.

Nun der schwierigste Teil:

```
SYSTEM ()

{
        IOObjCreate (&dig_1,NULL,BUS_TYPE_CPU,TSMCPU_DOUT,2, CLASS_DIGITAL,NULL);
        IOObjCreate (&rel_1,NULL,BUS_TYPE_CPU,TSMCPU_DOUT,0, CLASS_DIGITAL,NULL);
};
```

Zuerst weist man dig_1 (&dig_1) auf der TSMCPU (BUS_TYPE_CPU) den Ausgang 2(TSMCPU_DOUT,2) als digitalen IO (CLASS_DIGITAL) zu.
Dann weist man rel_1 den auf der TSMCPU den Ausgang 0 als digitalen IO zu.

Ausgang setzen

Ausgang dig_1 einschalten:

```
SYSTEM();
OUT(&dig_1,1);
```

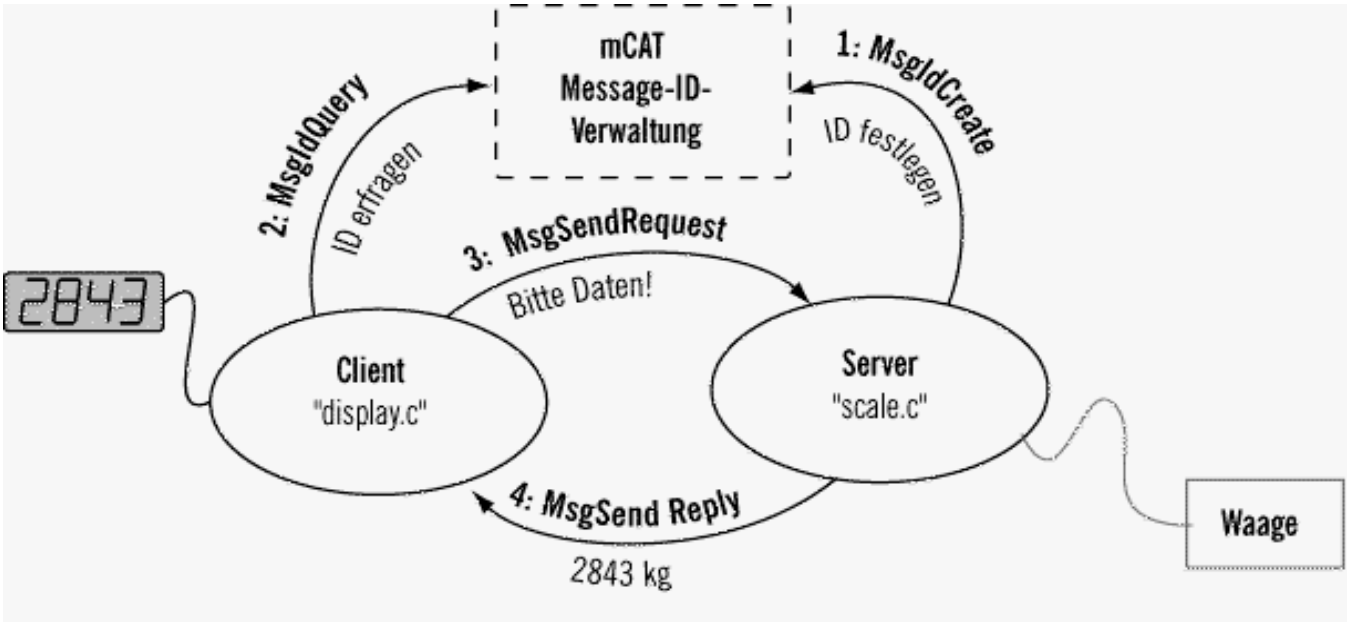
Ausgang dig_1 ausschalten:

```
 SYSTEM();
 OUT(&dig_1,0);
```

Nachrichten

Zur Wiederverwendbarkeit von Programmmodulen gehört auch eine möglichst universelle Weise der Verknüpfung untereinander. Idealerweise - im Sinne der Wiederverwendung - versucht man, eine Verantwortlichkeit für eine möglichst kleine Aufgabe zu definieren, beispielsweise die Abfrage einer Barcodepistole oder einer Waage, diese Aufgabe aber möglichst universell zu lösen. Dazu gehört auch, Module austauschbar zu machen, also beispielsweise die über RS232 angeschlossene Laborwaage gegen eine über RS485 angeschlossene Sackwaage zu tauschen, ohne eine Änderung an Partnermodulen zu verlangen, die das Gewicht abfragen wollen.

Eine ideale Möglichkeit, eine so weit gehende Isolation zu ermöglichen, ist die Verwendung von Nachrichten zur Kommunikation zwischen Prozessen (Tasks, Threads, Interrupttreibern), daher hat der Nachrichtenaustausch zentrale Bedeutung bei mCAT. Das mCAT Nachrichtenkonzept geht von einer Client/Server-Struktur der Kommunikationspartner aus: eine Task ist der Anbieter (Server) eines Dienstes, also in unserem Beispiel der Prozess, der die Waage abfragt. Der oder die anderen sind Konsumenten (Clients), beispielsweise eine Task, die das Gewicht auf einer 7-Segment-Anzeige darstellt - parallel dazu wäre eine andere Task denkbar, die eine Verknüpfung des Gewichts mit anderen Prozesswerten (z.B. Chargennummer, Wägegut, Temperatur) vornimmt und ein Etikett druckt. Entsprechend fragt der Client mit einer Anforderungs-Nachricht ("Request") beim Server nach einer Antwort-Nachricht ("Reply").



Aber wie verständigt man sich über den Nachrichteninhalt? Aus der Sicht des mCAT Nachrichtenkonzepts bestimmt die Wägetask (der Server) den Typ (die Struktur) der Nachricht, denn der Server weiss genau, welche Information er anbieten kann. Dabei ist es für einen Server durchaus üblich, verschiedene Nachrichtentypen zu definieren, wenn er unterschiedliche Informationen anzubieten hat. Für unser einfaches Beispiel nennen wir den Typ der Nachricht "sample/weight" (für die Benennung empfehlen wir die gleiche Konvention wie für die Namensgebung im IMD - entsprechend sind Nachrichten des Typs "mcat/..." reservierte Systemnachrichten). Diesen Nachrichtentyp melden wir nun bei mCAT offiziell an, dazu gibt es eine spezielle Systemfunktion, den Message-ID-Server, der von der Servertask über die Funktion MsgIdCreate angesprochen wird.

MsgIdCreate

Die Funktion erwartet:

- Den eigenen Task-ID
- Den Namen des Nachrichtentyps (im Beispiel "sample/weight")
- Einen NULL-Zeiger, damit ein ID-Code automatisch vergeben wird

und gibt einen Zeiger auf eine Struktur vom Typ MSGID zurück, die den Typcode für den eigenen Nachrichtentyp enthält. Aufruf:

```
MSGID *MsgIdCreate(short self, char *name, MSGID *msgid);
```

Diesen Zeiger auf eine Struktur vom Typ MSGID darf man nicht verwechseln mit der eigentlichen Nachricht, letztere wird erst später angelegt. Sinnvollerweise führt man die Funktion im TaskInit-Bereich der Server-Task aus, gleich nachdem man den eigenen Task-ID (self) vom TaskCreate erhalten hat, dann können mögliche Clients sofort darauf zugreifen. Wenn eine Konsumententask nun eine Antwortnachricht vom Typ (Beispiel) "sample/weight" haben will, braucht sie nur die Anforderungsnachricht dieses Typs mit MsgSendRequest loszuschicken. Dazu muss sie nicht wissen, welche Task der Server für die Anfrage ist, es reicht, wenn sie den Nachrichtentyp MSGID kennt (darüber wird eine größtmögliche Isolation und Austauschbarkeit von Server und Client erreicht).
Wie kommt der Client nun an den MsgID für "sample/weight"? Dafür gibt es die Systemfunktion:

MsgIdQuery

Wie der Name schon andeutet, kann man mit dieser Funktion bei mCAT nach einem MSGID fragen, man muss dazu nur den Namen des Nachrichtentyps kennen.

Aufruf:

```
MSGID *MsgIdQuery(char *name);
```

Und damit kommen wir nun endlich zum interessanten Teil des Vorhabens, dem eigentlichen Nachrichtenaustausch. Schritte 1 und 2, also MsgIdCreate und MsgIdQuery, haben wir schon oben besprochen. Diese Funktionen müssen die Kommunikationspartner ja auch nur einmal ausführen. Für den kontinuierlichen Nachrichtenaustausch sind die Schritte 3 und 4 der Grafik zuständig: MsgSendRequest und MsgSendReply.

MsgSendRequest

verschickt eine Anfragenachricht an die Servertask, die den Nachrichtentype definiert hat (mit MsgIdCreate) und benötigt 5 Parameter:

- Einen Zeiger auf die zu sendende Nachricht
- Den Task-ID der eigenen Task
- Zeiger auf MSGID (der eben mit MsgIdQuery erfragt wurde)
- Die Priorität der Anfrage
- Die gewünschte Priorität der Anwort

Die Funktion gibt einen Fehlercode ungleich Null zurück, wenn die Servertask nicht existent ist. Der Aufruf lautet formell so:

```
short MsgSendRequest (MSG *msg, short self, MSGID *msgid, word prio, word reply);
```

Wie die zu sendende Nachricht strukturiert wird, behandeln wir in Kürze, hier zunächst ein paar Worte zu:

Prioritäten

Traditionelle Echtzeitbetriebssysteme weisen jeder Task eine Priorität zu, die darüber entscheidet, welche von mehreren Tasks, die gleichzeitig lauffähig sind, denn tatsächlich den Prozessor bekommt. Bei mCAT wird auch das Prioritätenkonzept unter dem Aspekt der optimalen Zuständigkeit betrachtet und vorausgesetzt, dass derjenige, der einen Nachrichtenaustausch vornimmt, im Moment den besten Überblick über die Dringlichkeit hat. Man verschickt also mit einer Nachricht eine Priorität, die die **empfangende** Task auf eben diese Priorität setzt (was unter Umständen bedeutet, dass man sich selbst unterbricht, wenn man dem abgehenden Request eine höhere Priorität mitgibt als man selbst zur Zeit hat). Je nach Anforderung kann man auch die Priorität, die man selbst mit der Antwort des Servers zurückerhält, einstellen. Wer jetzt meint, damit sei Willkür und Wildwuchs Tür und Tor geöffnet, der soll sich bitte ins Gedächtnis rufen, dass wir es hier nicht mit konkurrierenden Benutzern zu tun haben wie bei einem Multi-User-Betriebssystem, sondern mit **einem** Programmierer, der sehr wohl weiss (es zumindest wissen sollte), wann welcher Prozess vorrangig ist.

MsgWait

Dem Server wird die Nachricht durch das MsgSendRequest des Client' in seinen Briefkasten gelegt, der bei mCAT "Queue" heißt, und ganz im Sinne der feinen englischen Art bedeutet, dass auch mehrere Nachrichten hintereinander aufgereiht werden können.

Was hat nun der Empfänger der Nachricht, der Server, in der Zwischenzeit zu tun? Im Sinne des Gemeinwohls muss eine Task immer versuchen, möglichst wenig Prozessorzeit zu verbrauchen. Dies geschieht am einfachsten durch Aufruf der Systemfunktion MsgWait.

```
msg = MsgWait(short queuehdl, lword timeout);
```

Das erste Argument gibt die Warteschlange (Queue) an, 0 ist die Standardwarteschlange, das zweite Argument eine Ablaufzeit in Millisekunden. Rückgabewert ist ein Zeiger auf den Nachrichtenkopf. Diese Funktion wartet auf eine ankommende Nachricht und versetzt die Task solange in den Schlafzustand "sleep". Ein "normales" TaskMain in mCAT ist also eine Endlosschleife, die so aussieht:

- Warte auf Nachricht
- Tu, was die Nachricht verlangt
- Antworte
- .. und wieder von vorn mit MsgWait(0,0);

MsgSendReply

Nachdem nun der Server aus dem MsgWait die Anforderung entgegengenommen hat (Details später), beantwortet er dieselbe mit der Funktion "MsgSendReply". Diese Funktion benötigt die folgenden Parameter:

- Einen Zeiger auf die Antwortnachricht
- Den eigenen Task-ID
- Einen Quittungscode (ACK oder NAK), der in msg->error gefüllt wird

und gibt einen Fehlercode zurück, der normal 0 sein muss. Aufruf:

```
short MsgSendReply (MSG *msg, short self, word error);
```

Aus der Anfragenachricht kennt der Server die Task, an die die Antwortnachricht geschickt werden soll. Den eigenen Task-ID muss man nur deswegen einsetzen, damit das System verhindern kann, in eine Endlosschleife zu geraten - eine Nachricht an die eigene Queue ist aus diesem Grund nicht zulässig.

Damit wäre der erste Zyklus unseres Beispiels beendet, es können nun beliebig viele weitere MsgSendRequest-/MsgSendReply-Zyklen folgen. Was uns noch fehlt, ist eine Erklärung der Nachricht selbst, in den Aufrufbeispielen oben immer als MSG *msg angegeben:

MSG

Bei MSG handelt es sich um eine zweiteilige Datenstruktur, bestehend aus dem Nachrichtenkopf und dem Nachrichtenkörper. Der Kopf MSG ist in msg.h definiert und wird automatisch über mcat.h vom IMD-Programm per #include eingefügt. Generell wird der Kopf vom System verwaltet und ist vom Anwender nicht zu ändern. Der Anwender erzeugt jedoch, und zwar im Rahmen der Programmierung der Server-Task, eine Strukturdefinition der Spezialnachricht für "seinen" Nachrichtentyp, die in unserem Fall etwa wie folgt aussehen könnte:

```
typedef struct {
MSG msg;
lword weight;
} MSGWEIGHT;
```

Damit wird ein Datentyp MSGWEIGHT definiert, der eine Struktur beschreibt, bestehend aus der Struktur msg vom Typ MSG (dem Standard-Nachrichtenkopf) und dem 32-Bit-Wert weight (das ist das Gewicht, das der Server an den Client übermitteln will). **Während der Server diesen Datentyp definiert** (passend zu dem mit MsgIdCreate erzeugten MSGID), **sollte die Datenstruktur aber im Speicherbereich des Konsumenten (Client) angelegt werden**, der Server zeigt dann nur darauf. Dazu ist es hilfreich, die Msg-Strukturdefinition als Header-Datei anzulegen - in unserem Beispiel als scale.h.

Denn Nachrichten werden nicht wirklich physikalisch transportiert, sie bleiben an der gleichen Stelle im Speicher, es wird lediglich ein sogenanntes Signal (ThreadSignal) zusammen mit dem Prioritätswert an die Empfängertask geschickt. Der Client muss Sorge tragen, dass er den Speicherbereich der Nachrichten-Datenstruktur (also z.B. weight) nicht verändert, bevor der Server sie verarbeitet hat. Am einfachsten geschieht das dadurch, dass man vor Änderungen wartet, bis die Antwort (Reply) des Servers vorliegt.

In der Client-Task sieht das z.B. so aus:

```
MSG *msg;
MSGWEIGHT req;
```

Man beachte, dass req kein Zeiger auf einen Datentyp MSGWEIGHT ist, sondern dass tatsächlich Speicher für req angelegt wird. Dagegen ist *msg nur ein Zeiger, der normalerweise von MsgWait gefüllt wird und im Beispiel auf den msg-Teil der Struktur req zeigt - wenn die ankommende Nachricht vom Typ MSGWEIGHT ist!

Das Zusammenspiel zwischen den verschiedenen Elementen finden Sie im Anhang noch einmal grafisch aufbereitet.

Man ist übrigens nicht daran gebunden, nur eine Nachricht im Umlauf zu haben, man kann mehrere Datenstrukturen des gleichen Typs anlegen und in Umlauf bringen, nur die Unterscheidung ist dann "Handarbeit" - obwohl die Reihenfolge gewahrt bleibt, wenn man nicht einzelne mit anderen Prioritäten verschickt.

Die Praxis

Für unsere Versuche entsprechend der obigen Grafik mit Waage und Anzeige sind also zunächst einmal zwei C-Quelltexte für die beiden kommunizierenden Tasks anzulegen. Wir müssen das mit dem Dienstprogramm IMD tun, um unseren Task Descriptor korrekt aufgebaut zu bekommen. Für das Beispiel wollen wir den Server "SCALE" nennen und den Client "DISPLAY". Wir starten dazu in einem unter x:\mcat.2\cc\ neu erstellten Unterverzeichnis (z.B. \scale): imd scale.c. Im Dialog fügen wir als Name ein: sample/scale. Genauso verfahren wir mit dem Client: imd display.c mit dem Namen sample/display.

Ein Programm mit mehreren Tasks erfordert etwas mehr Überlegung als ein "Hello World", an einigen Stellen muss man schon die Standardvorgaben verlassen - man kann z.B. nicht zwei Tasks für die gleiche Startadresse im Zielsystem vorsehen. Also sind die "MAKEFILE"s anzupassen, die Steuerdateien für Compiler - Assembler - Linker. Der Standard MAKEFILE definiert neben dem Projektnamen - der gleichzeitig der Name unserer C-Quelldatei ist - den Speicher mit dem Schlüsselwort RAM. Im Normalfall (Hello World) steht da RAM = std_ram (bzw. std_rom für dauerhafte Speicherung im Flash-EPROM), was das vmake-Programm veranlasst, die Werte aus der Datei x:\mcat.2\etc\t9h_cor\std_ram.trg zu benutzen. Dort steht dann:

```
[ADDR]

ROMSTART=0x402000
RAMSTART=0x406000

ROMLENGTH=0x04000
RAMLENGTH=0x04000
```

und bedeutet, dass der Start des Speicherbereichs für den Programmcode (ROM) bei 402000 Hex liegen soll (dem Start des freien RAM-Bereichs auf der TSM-CPU) und der für die Variablen bei 406000 Hex. Beide Bereiche sind 4000 Hex, also 16kByte, groß (RxMLENGTH).

Wir müssen den Makefile für unsere Bedürfnisse anpassen, wenn unser Programmcode größer wird oder wenn z.B. auf den Standardadressen schon eine Task liegt. Wenn wir zwei Tasks in einem Unterverzeichnis haben, kann es keine zwei unterschiedlichen Makefiles gleichen Namens geben. VMAKE kann aber mit der Option -f filename.mak aufgerufen werden, dann wird nicht die Datei MAKEFILE ausgewertet, sondern die angegebene Datei. Also erstellen wir die Dateien scale.mak und display.mak (MAKEFILE kopieren, PROJECT und RAM wie unten gezeigt ändern, Rest unverändert lassen) und verweisen darin auf die Dateien scale.trg und display.trg mit den absoluten Speicherangaben:

In scale.mak ändern:

```
#
PROJECT = scale
#
RAM = scale
#
```

scale.trg:

```
[ADDR]

ROMSTART=0x402000
RAMSTART=0x406000

ROMLENGTH=0x02000
RAMLENGTH=0x02000
```

In display.mak ändern:

```
#
PROJECT = display
#
RAM = display
#
```

display.trg:

```
[ADDR]

ROMSTART=0x404000
RAMSTART=0x408000

ROMLENGTH=0x02000
RAMLENGTH=0x02000
```

Wir haben nur 8kByte jeweils für Code und Daten reserviert - mehr als ausreichend für diese einfache Anwendung. Die Bezeichnung ROMSTART ist natürlich insofern irreführend, als sich auch dieser Bereich im RAM befindet - aber es ist jeweils der Start des Codebereichs, der in einer Endversion in das Flash-EPROM verlegt würde.

Und jetzt endlich an das eigentliche Programm, wir beginnen mit der Datei scale.c, deren Rumpf wir mit IMD erstellt haben. Dort wird zunächst oben bei den Includes eingefügt:

```
#include <simpleio.h>
#include "scale.h"
```

Mit simpleio.h ermöglichen wir den Zugriff auf die serielle Schnittstelle, die wir für die Ausgabe von Statusmeldungen brauchen wollen. In der Datei scale.h wird die Nachrichtenstruktur des Nachrichtentyps "weight" definiert (wie oben bei MSG beschrieben), die in eine ".h"-Datei ausgelagert wird, um sie auch in der Client-Task einschließen zu können. Mit den <spitzen> Klammern wird übrigens auf eine Datei im Verzeichnis \cc\include\ verwiesen, mit "Anführungszeichen" auf eine Datei im aktuellen Verzeichnis (\scale).

Weiter geht es hinter dem IMD-Bereich: hier ist zunächst funktionsübergreifend ein Zeiger auf eine Struktur vom Typ MSGID zu deklarieren, den wir msgidweight nennen:

```
MSGID *msgidweight;
```

Als Server Task sollte scale.c diesen Zeiger nun im Task-Init füllen lassen, wozu die Funktion MsgIdCreate nach dem Task-Create aufgerufen wird:

```
msgidweight = MsgIdCreate(Self,"sample/weight",NULL);
if (!msgidweight) TaskDelete(Self);
```

Nochmal: Die Struktur MSGID ist nicht die Nachricht oder der Nachrichtenkopf, sondern nur eine Struktur, die mCAT für die Verwaltung des Typs (der Art) der Nachricht benötigt. Innerhalb der Struktur ist "type" der eigentliche Typcode. Die Nachrichtenstruktur ist nun vereinbart, die Aufgabe der Task scale ist es nun, im TaskMain auf hereinkommende Nachrichtenanforderungen dieses Typs zu warten und diese zu beantworten. Da wir für das Beispiel keine wirklich vorhandene Waage voraussetzen können, gehen wir von einem Startgewicht von 1000g aus und addieren mit jedem Nachrichtenzyklus 10g. Dazu nutzen wir die 32-Bit-Int-Variable weight. Die Nachricht selbst wird ja im Client-Speicher angelegt, hier ist nur ein Zeiger darauf nötig, der als msg vereinbart wird:

```
MSGWEIGHT *msg;
```

Dann ist für die Statusausgaben noch die Kanalnummer der seriellen Schnittstelle anzulegen und vorzubesetzen mit
word ser = 0.

Mit loop machen wir eine Endlosschleife auf, in der auf die Nachrichten gewartet wird. Wenn die Funktion MsgWait aufgerufen wird, hält sie die Task an, bis eine Nachricht eintrifft. Dann füllt sie den Rückgabewert mit dem Zeiger auf die Nachricht - allerdings auf die generische Nachricht vom Typ MSG, also nur dem universellen Nachrichtenkopf. Wir wollen aber weiterarbeiten mit der speziellen Nachricht vom Typ MSGWEIGHT, also fordern wir eine zwangsweise Abbildung (Cast) darauf , was ja geht, da auch MSGWEIGHT mit einer Struktur vom Typ MSG anfängt.

```
msg = (MSGWEIGHT *) MsgWait(0,0L);
```

Wir müssen dann prüfen, ob wir auch wirklich eine Nachricht des gewünschten Typs bekommen haben. Dazu vergleichen wir den Struktureintrag type von MSG mit dem Struktureintrag id von MSGID (Teil von MSGIDWEIGHT).
if (msg->msg.type == msgidweight->id) {

Nur wenn die ID-Codes übereinstimmen, folgt die korrekte Nachrichtenbehandlung. Hier werden nun je Zyklus 10g addiert und der Wert in die Nachrichtenstruktur eingefüllt. Dann erfolgt eine Ausgabe darüber auf der Konsole.

```
weight = weight +10;
msg->weight = weight;
SIOWrStr(ser,"Server 'Scale': Gewicht = ");
SIOWrDecLWord(ser,weight,0);
```

Schließlich wird die Nachricht an den Anforderer zurückgesandt:

```
MsgSendReply(msg,Self,ACK);
```

Soweit also zur Servertask scale.c, nun zum Client display.c:

Auch hier sind zunächst wieder simpleio.h und scale.h einzuschließen, dagegen ist im Startup keine Erweiterung nötig. Unser Programm spielt sich also im TaskMain ab und beginnt mit den Deklarationen. Auch hier wird eine lokale Variable weight angelegt, es wird ein Zeiger auf den MSGID vereinbart und einer auf den Nachrichtenkopf msg. Wichtig für den Client ist nun, dass er die Nachrichtenstruktur vom Typ MSGWEIGHT (definiert in scale.h) in seinem Speicherbereich anlegt, hier unter dem Namen req. Schließlich wird der Kanal für die Konsolenausgabe vereinbart:

```
lword weight;
MSGID *msgidweight;
MSG *msg;
MSGWEIGHT req;
word ser = 0;
```

Als erste Aktion wird der ID der Nachricht "sample/weight" erfragt:

```
msgidweight = MsgIdQuery("sample/weight");
```

Dann beginnt auch hier eine Endlosschleife, diesmal mit dem Absenden einer Anforderung an den Server, der die Nachricht vom Typ "sample/weight" beantwortet.

```
MsgSendRequest(&req,Self,msgidweight,200,200);
```

Dann warten auf die Antwort des Servers und Überprüfung, ob der ID stimmt

```
msg = MsgWait(0,0L);
if (msg->type == msgidweight->id) {
```

Zuweisung aus dem Datenfeld der Nachricht, der Cast ist notwendig, weil aus MsgWait nur ein Zeiger auf eine MSG zurückgegeben wird, weight aber nur Teil der erweiterten Struktur req vom Typ MSGWEIGHT ist. Damit erhalten wir den Wert von Scale, der uns interessiert.

```
weight = ((MSGWEIGHT*)msg)->weight;
```

Statt der Ausgabe auf einem Display machen wir es uns einfacher und stellen den empfangenen Wert auf der Konsole dar:

```
SIOWrStr(ser,"\nClient 'Display': Empfang = ");
SIOWrDecLWord(ser,weight,0);
SIOWrLn(ser);
SIOWrLn(ser);
```

Formulierung einer Abbruchbedingung, wenn 100 Nachrichten ausgetauscht sind und Fehlerabfang für Nachrichten anderen Typs.

```
if (weight >= 2000) TaskDelete(Self);
} else {
SIOWrStr(ser,"Client Display: Unbekannte Nachricht!\n");
```

Damit ist das Ende der Schleife und des Programms display.c erreicht. Jetzt muss das Ganze noch compiliert und heruntergeladen werden. Dazu wird VMAKE mit den vorbereiteten Makefiles gestartet:

vmake -f display.mak -r und vmake -f scale.mak -r.

Das -r braucht man eigentlich nur, wenn man etwas geändert hat, aber die Anzahl der Zeichen nicht verändert wurde. Die Warnung des Compilers, das eine Zeile nicht erreicht wird o.ä. kann man in der Regel ignorieren. Wer Spaß daran hat, kann sich die ASM-Dateien ansehen, um festzustellen, wie der generierte Assembler-Quelltext aussieht, interessanter ist die Linker-Datei MAP, die die Speicherbereiche zeigt, für Scale z.B. einen Code-Bereich von 402000 bis 402445 Hex, also etwas mehr als ein kByte, dazu 24 Byte Daten. Zumindest sollte man gelegentlich überprüfen, ob die Speicherbereiche kommunizierender Tasks sich nicht überlappen - es gibt dafür keine Warnungen, der Compiler kann nicht wissen, dass die Tasks zusammengehören.

Schließlich wären die Hex-Dateien scale.shx und display.shx herunterzuladen. Es handelt sich um ASCII-Dateien, die Informationen über die Speicheradresse beinhalten, sie werden z.B. mit wLGO heruntergeladen: F3, dann Datei aussuchen und starten. Vorher sollte man die TSM-CPU900 angeschaltet haben - auf dem wLGO-Fenster sind dann die Ausgaben der Startsequenz zu sehen. Wenn da irgendetwas von Bootmon steht, ist der Terminator nicht gesteckt, wenn wirre Zeichen kommen, sind die Übertragungsparameter falsch eingestellt. Richtig wäre 19200 bit/s, 8N1. Und wenn gar nichts kommt, war es die falsche Schnittstelle oder ein Kabeldefekt.

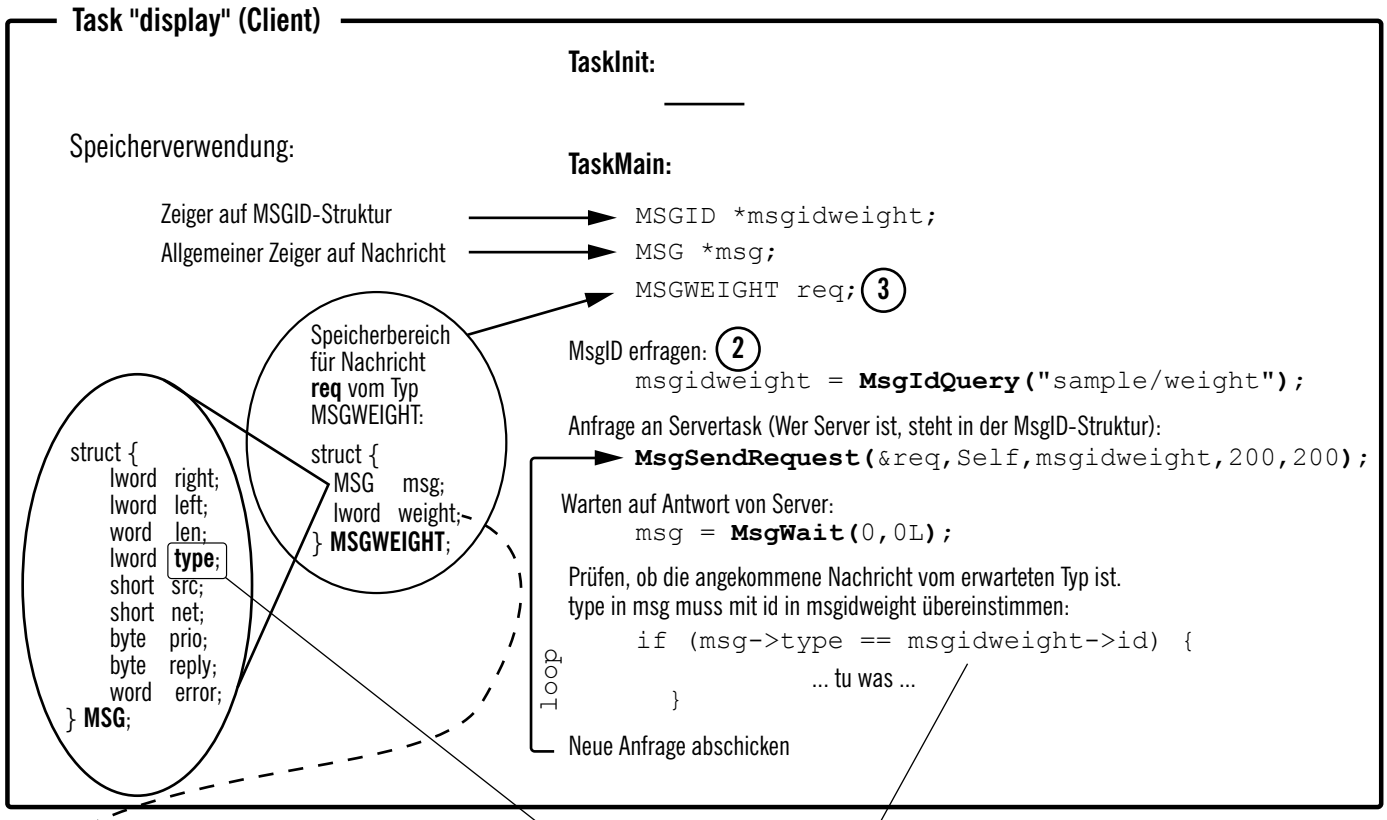
Nach dem Herunterladen werden die Programme auf Ihrer Code-Basisadresse (ROMSTART) gestartet mit INIT 402000 für SCALE (muss zuerst gestartet werden wg. MsgID) und mit INIT 404000 für DISPLAY. Der Bildschirm läuft nun über mit "Gewichts"-Ausgaben, wenn alles ok war. Herzlichen Glückwunsch!

Allgemein noch ein paar Hinweise zu Nachrichten: Ankommende Nachrichten werden in Warteschlangen (Queues) bei der jeweiligen Task eingereiht. Wenn die Nachrichten alle die gleiche Priorität haben, behalten sie die Reihenfolge, in der sie ankommen. Hat eine ankommende Nachricht eine höhere Priorität, wird sie vorne in der Schlange einsortiert, d.h. sie überholt weniger wichtige Nachrichten. Dabei wird die Empfängertask auf diese hohe Priorität gestellt, selbst wenn die Task nicht im MsgWait steht, so dass sie eine laufende Aufgabe schnell abarbeiten kann um baldmöglichst die wichtige Nachricht abzunehmen. Man muss Nachrichten übrigens nicht mit dem MsgWait-Aufruf abholen, man kann auch pollen, d.h. während des Programmablaufs nachschauen, ob eine Nachricht vorliegt. Dazu gibt es den Systemaufruf MsgGet.

In späteren Kapiteln kann man dann noch lernen, dass man Nachrichtenwartenschlange (Queues) erzeugen kann, die nur Nachrichten eines bestimmten Typs annehmen und einzelne Threads abstellen kann, die diese Warteschlangen bearbeiten.

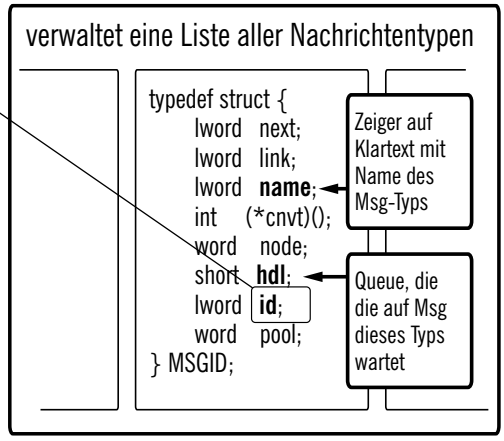
Autor: Walter L. Giesler, ELZET80, giesler@elzet80.de
Erstellt: 2000-05-29
Last Update:2000-06-15
Status: Alpha

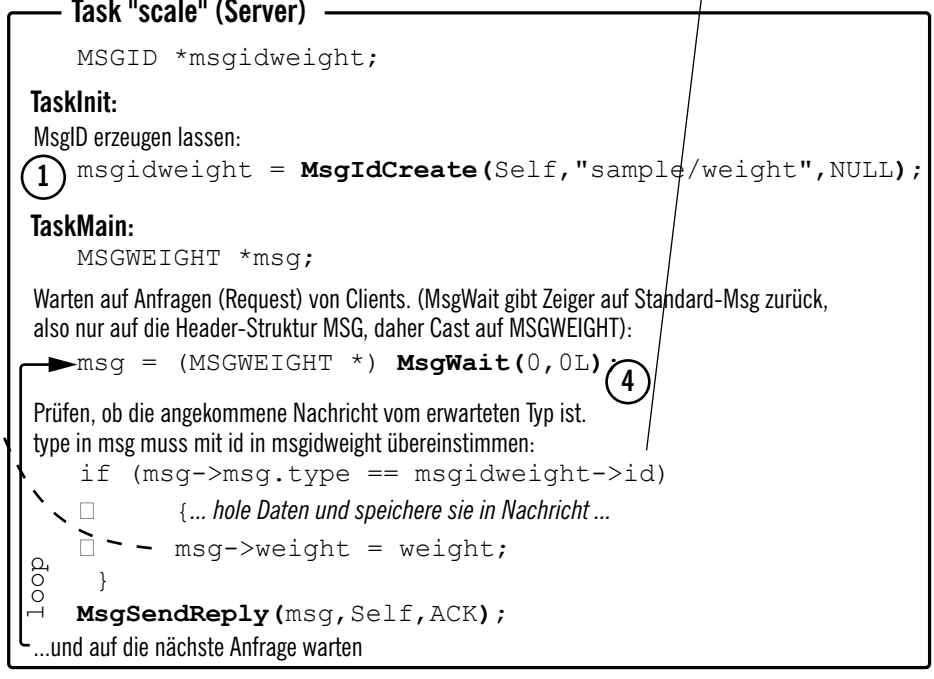
Datenstrukturen und Nachrichtenaustausch im Beispielprogramm scale/display



1. Ein Anbieter (Server), hier scale.c, definiert eine Nachricht mit (z.B.) Waagendaten vom Typ MSGWEIGHT und gibt sie über MsgID-Create bekannt unter dem Namen (z.B.) "sample/weight". Die MsgID-Verwaltung weist einen MsgID zu.
2. Ein Konsument (Client), hier display.c, erfragt mit MsgIdQuery den MsgID der Nachricht mit Namen "sample/weight".
3. Der Client legt in **seinem** Speicherbereich die eigentliche Nachricht, hier "req", an - nach der vom Server definierten Struktur MSGWEIGHT (in scale.h).
4. Der Server bekommt mit der Request-Nachricht einen Zeiger auf den Nachrichtenkopf im Speicherbereich des Client. (Erst nach dem Reply darf der Client wieder auf die Nachricht zugreifen.)

"id" in der MSGID-Struktur entspricht "type" im Nachrichtenheader MSG.





Die Struktur MSGID wird vom System angelegt und ist **unabhängig** von der eigentlichen Nachricht.

Über MSGID macht das System die Task (Queue) ausfindig, die eine Nachricht dieses Typs bearbeiten kann.

Die Struktur MSGID:

```
typedef struct {
          lword     next;
          lword     link;
          lword     name;
          int       (*cnvt)();
          word      node;
          short     hdl;
          lword     id;
          word      pool;
} MSGID;
```

Die Struktur MSG:

```
typedef struct {
          lword     right;
          lword     left;
          word      len;
          lword     type;
          short     src;
          short     net;
          byte      prio;
          byte      reply;
          word      error;
} MSG;
```

Die Struktur MSGWEIGHT (scale.h):

```
typedef struct {
    MSG     msg;
    lword   weight;
} MSGWEIGHT;
```

Target-Speicherbereiche in scale.trg:

```
[ADDR]
romstart=0x402000
ramstart=0x406000
romlength=0x02000
ramlength=0x02000
```

Der MAKEFILE scale.mak:

```
PRINT
PRINT       ** Es wird compiliert: scale.c **
PRINT
PRINT       (c) 1999 mocom software GmbH & Co KG
PRINT       Author:     Volker Goller
PRINT
#           Deutsche Aenderungen: Walter L. Giesler 11.6.2000

#           The project macro is the name of
#                       * the main source file
#                       * the output file
#                       * the map file
```

PROJECT = scale

```
#
# TARGET: The address definition. We use the std_r?m macros here
#
#           std_rom     = Flash-EPROM
#           std_ram     = RAM
#           meinprog = Speicherfestlegungen in meinprog.trg verwenden
#
```

TARGET = scale

```
#
# CMD MACRO:            passing extra arguments to the c-compilers
#                       command line. We use this command to pass the
#                       $(CORE) macro as a C macro!
#
CMD = -D$(CORE)

#
# LIST OF YOUR OBJECT FILES.
# The list will include $(PROJECT) at least.
#
OBJFILES = $(PROJECT).rel

#
# LIST YOUR INCLUDE FILES
#
INCFILES =

#
# HOW TO BUILD THE TARGET?
#
#           * compile C files (pre-defined rule)
#           * create linker control file from template (MKLNK.EXE)
#           * call linker
#           * tag output file (TAG.EXE)
#           * convert output (abs) top SHX
#
$(PROJECT).shx: $(OBJFILES)
                        $(MKLNK) $(TARGET) $(PROJECT) $(OBJFILES)
                        $(LNK) $(PROJECT).LNK -o$(PROJECT).abs
                        $(TAG)
                        $(CONVERT) $(SHX_OUTPUT)

#
# individual dependencies
#
$(PROJECT).rel:         $(PROJECT).c $(INCFILES)
```

Der MAKEFILE display.mak:

```
PRINT
PRINT       ** Es wird compiliert: display.c **
PRINT
PRINT       (c) 1999 mocom software GmbH & Co KG
PRINT       Author:     Volker Goller
PRINT
#           Deutsche Aenderungen: Walter L. Giesler 11.6.2000

#           The project macro is the name of
#                       * the main source file
#                       * the output file
#                       * the map file
```

PROJECT = display

```
#
# TARGET: The address definition. We use the std_r?m macros here
#
#           std_rom     = Flash-EPROM
#           std_ram     = RAM
#           meinprog = Speicherfestlegungen in meinprog.trg verwenden
#
```

TARGET = display

```
#
# CMD MACRO:            passing extra arguments to the c-compilers
#                       command line. We use this command to pass the
#                       $(CORE) macro as a C macro!
#
CMD = -D$(CORE)

#
# LIST OF YOUR OBJECT FILES.
# The list will include $(PROJECT) at least.
#
OBJFILES = $(PROJECT).rel

#
# LIST YOUR INCLUDE FILES
#
INCFILES =

#
# HOW TO BUILD THE TARGET?
#
#           * compile C files (pre-defined rule)
#           * create linker control file from template (MKLNK.EXE)
#           * call linker
#           * tag output file (TAG.EXE)
#           * convert output (abs) top SHX
#
$(PROJECT).shx: $(OBJFILES)
                        $(MKLNK) $(TARGET) $(PROJECT) $(OBJFILES)
                        $(LNK) $(PROJECT).LNK -o$(PROJECT).abs
                        $(TAG)
                        $(CONVERT) $(SHX_OUTPUT)

#
# individual dependencies
#
$(PROJECT).rel:         $(PROJECT).c $(INCFILES)
```

Target-Speicherbereiche in display.trg:

```
[ADDR]
romstart=0x404000
ramstart=0x408000
romlength=0x02000
ramlength=0x02000
```

Das Programm scale.c:

```
/*----[InitialTaskDescriptor]----------------------------------------------*/
#include <mcat.h>
#include <vtype.h>
                                                                /* Für Zugriff auf Schnittstelle */
#include <Simpleio.h>
                                                                /* Message-Struktur */
#include "scale.h"


#define Version      '1'
#define Release      '0'
#define Interim      '0'
#define Priority     200
#define Mode         MODE_TASK
#define ID           255
#define Stack        1024
#define Heap         0
#define TaskId       short

#ifdef __STDC__
  extern void __cstart(IMD *imd);
  PUBLIC void TaskInit(IMD *imd);
  PRIVAT void TaskMain();
  PUBLIC TaskId Self;
#define CONST        const
#else
  PRIVAT void TaskInit();
  PRIVAT void TaskMain();
  PRIVAT TaskId Self;
#define CONST
#endif

#define ip(p)        (void(*)())(p)

PRIVAT IMD CONST task_imd = {
          21930,                        /* IMD.Pattern */
#ifdef __STDC__
          ip(__cstart),                 /* IMD.Init */
#else
          ip(TaskInit),                 /* IMD.Init */
#endif
          ip(TaskMain),                 /* IMD.PC */
          Stack,                        /* IMD.StackSize */
          Heap,                         /* IMD.StackSize */
          0x112d2d11,                   /* IMD.Build */
          Priority,             /* IMD.Priority */
          Mode,                         /* IMD.Mode */
          ID,                           /* IMD.ID */
          Version,              /* IMD.Version : VERSION */
          '.',                          /* ! */
          Release,              /* IMD.Version : RELEASE */
          Interim,              /* IMD.Version : INTERIM */
          '\0',                         /* IMD.Version */
          "sample/scale\0\0\0\0",       /* IMD.Name */
          26412                         /* IMD.Check */
};
```

```
                                                        /* Zeiger auf Message-ID vereinbaren */
MSGID *msgidweight;

void TaskInit (imd)
IMD *imd;
{
    short error;

    Protect();
    Self = TaskStartup(imd,FromTop,&error,0);
                                                        /* Nachrichtentyp (MsgID-Struktur) erzeugen */
          msgidweight = MsgIdCreate(Self,"sample/weight",NULL);
          if (!msgidweight) return;
    UnProtect();
}

void TaskMain ()
{
                                                        /* Anlegen der Gewichts-Dummy-Variable weight */
          lword weight = 1000l;
                                                        /* Zeiger auf erwartete Nachricht vereinbaren */
          MSGWEIGHT *msg;
                                                        /* Kanalnummer der seriellen Schnittstelle */
          word ser = 0;

          loop {
                                                        /*        Warten auf Nachricht, dabei Cast auf Zeiger
                                                                  auf Nachricht vom Typ MSGWEIGHT (sonst würde
                                                                  msg nur auf den Headerteil MSG zeigen) */
                    msg = (MSGWEIGHT *) MsgWait(0,0l);
                                                        /* Prüfung, ob der Nachrichtentyp ok ist */
                    if (msg->msg.type == msgidweight->id) {
                                                        /* Dummy-Waage heraufzählen lassen */
                              weight = weight +10;
                                                        /* Lokale Variable in Nachricht kopieren */
                              msg->weight = weight;
                                                        /* Ausgabe auf Konsole */
                              SIOWrStr(ser,"Server 'Scale': Gewicht = ");
                              SIOWrDecLWord(ser,weight,0);
                                                        /* Positive Bestätigung an Anforderer */
                              MsgSendReply(msg,Self,ACK);
                    } else {
                              SIOWrStr(ser,"Server 'Scale': Nachrichtentyp unbekannt!\n");
                                                        /* Negative Bestätigung an Anforderer */
                              MsgSendReply(msg,Self,NAK);
                    }
          }

                                                        /* Hier kommt das Programm nie hin */
    TaskDelete(Self);
}
/*----[END]---------------------------------------------------------------*/
```

Das Programm display.c:

```
/*----[InitialTaskDescriptor]---------------------------------------------*/
#include <mcat.h>
#include <vtype.h>
#include <Simpleio.h>
#include "scale.h"

#define Version      '1'
```

... Rest Header wie bei scale.c

```
...
          "sample/display\0\0", /* IMD.Name */
          31982                          /* IMD.Check */
};
```

```
void TaskInit (imd)
IMD *imd;
{
    short error;

    Protect();
    Self = TaskStartup(imd,FromTop,&error,0);
    UnProtect();
}

void TaskMain ()
{
                                                        /* Gewicht */
          lword weight;
                                                        /* Zeiger auf MSGID*/
          MSGID *msgidweight;
                                                        /* Zeiger auf Nachrichtenkopf */
          MSG *msg;
                                                        /* Speicherbereich für Nachricht vom Typ MSGWEIGHT */
          MSGWEIGHT req;
                                                        /* Konsolenkanal */
          word ser = 0;

                                                        /* Zeiger auf Nachrichtentyp abfragen und prüfen */
          msgidweight = MsgIdQuery("sample/weight");
          if (msgidweight == NULL) {
                    SIOWrStr(ser,"Client 'Display': Kein MsgID 'weight'!\n");
                    TaskDelete(Self);
          }

          loop {
                                                        /* Nachricht versenden, 1. Argument muss Zeiger sein*/
                    MsgSendRequest(&req,Self,msgidweight,200,200);
                                                        /* Warten auf Nachricht */
                    msg = MsgWait(0,0);
                                                        /* uns interessieren nur Antworten des Typs sample/weight */
                    if (msg->type == msgidweight->id) {
                                                        /* Gewicht kopieren */
                              weight = ((MSGWEIGHT*)msg)->weight;
                                                        /* Ausgabe auf Konsole */
                              SIOWrStr(ser,"\nClient 'Display': Empfang = ");
                              SIOWrDecLWord(ser,weight,0);
                              SIOWrLn(ser);
                              SIOWrLn(ser);
                                                        /* Abbruchbedingung */
                              if (weight >= 2000) TaskDelete(Self);
                    } else {
                              SIOWrStr(ser,"Client Display: Unbekannte Nachricht!\n");
                    }
          }

                                                        /* Hier kommt das Programm nicht an */
    TaskDelete(Self);
}
/*----[END]------------------------------------------------------------*/
```