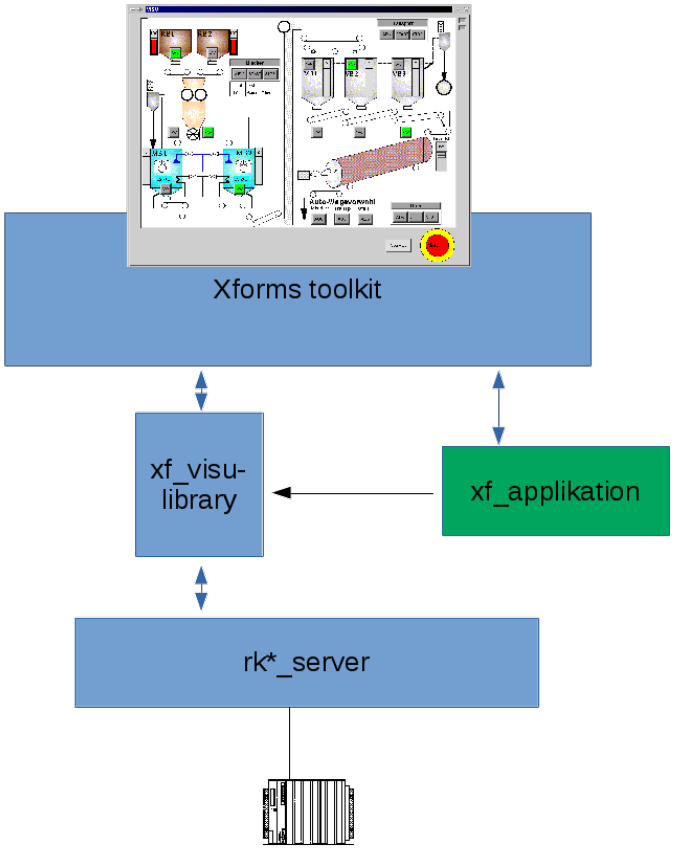


xf_visu

Visualisierungsystem auf Basis von Xforms (toolkit)

Bedienungsanleitung / Projektierungsanleitung

Stand 07.10.2019

xf_visu ist ein Projekt von

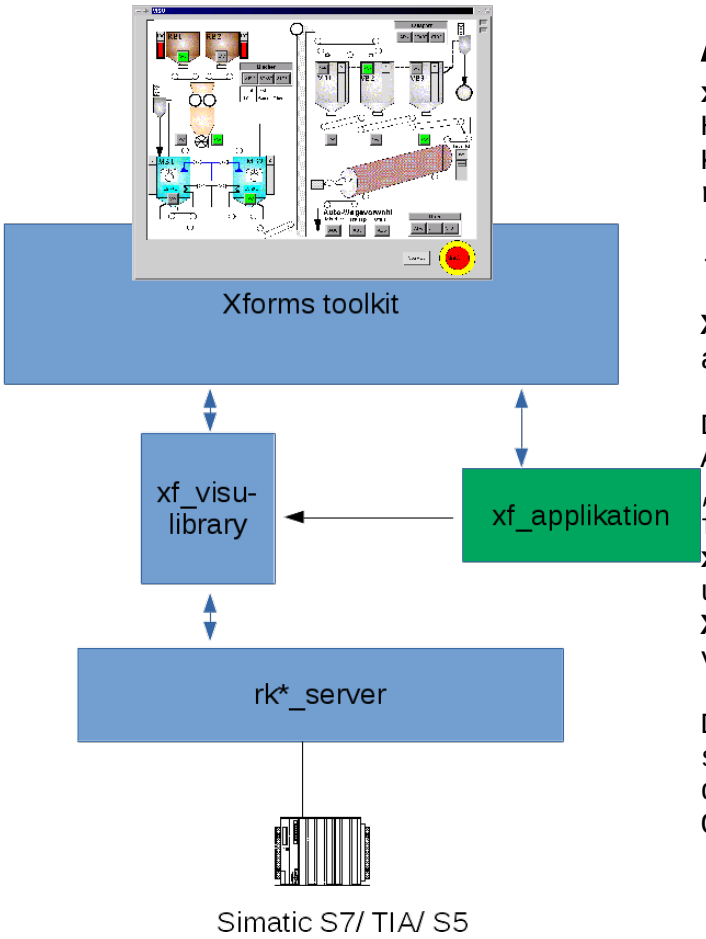
Heisch Automation

Ingenieurbüro für Industrieautomatisierung

Adresse: Im Vorderen Großthal 4 D 76857 Albersweiler /Pfalz
Tel: +49 6345 9496732 Mobil: +49 171 4311359
e-mail: sales@heisch-automation.de www.heisch-automation.de

Inhaltsverzeichnis

TODO : xf_set neuer Pointer ist *stru



Grundsätzlicher Aufbau

xf_visu basiert auf den beiden Hauptkomponenten **xforms** und der Kommunikationsserver-Familie **rk*_server**.

Xforms ist für die Darstellung auf dem Bildschirm zuständig

Die projektspezifische Applikation, hier „**xf_applikation**“ benutzt function-Aufrufe aus der **xf_library**,
um den Datenaustausch zwischen **Xforms** und den **rk*_server**n zu vermitteln.

Die **rk*_server** lesen und schreiben die Daten von oder zu den angeschlossenen Simatic-CPUs.

Für spezielle Aufgaben, z.B. Störmelde-Erfassung, Daten-Logging, Rezeptabruf mit SPS-seitiger Initiative können spezialisierte, externe Programme benutzt werden, die an den rk*_servern andocken.
Die visuellen Interfaces können sowohl direkt in der Applikation programmiert als auch als eigenständige Prozesse gestartet werden.

Für die Kommunikation externer Programme mit der Applikation bietet sich shared memory an, dazu gibt es bereits einige Code-Fragmente.
Natürlich können auch andere Kommunikatiosmöglichkeiten, z.B. tcp-Kommunikation genutzt werden, dazu ist allerdings nichts vorbereitet.

Die gesamten Aktivitäten der xf_visu-library werden von Xforms gesteuert:
Xforms verfügt über einen Aufruf „fl_set_idle_callback()“.
Die darin definierte function wird durchlaufen, wenn gerade nichts zu tun ist. .. und das ist fast immer der Fall.
Die zyklischen Aufgaben von xf_visu sind über diese function eingehängt.

(z.B: Anforderung eines Datenbereichs aus der SPS, Versorgung der Xforms-Objects mit neuen Daten etc.)

Ein Xforms-Objekt (FL_OBJECT) verfügt über 2 freie Pointer.
Diese sind laut Dokumentation für den Anwender reserviert und werden von Xforms nicht bearbeitet oder genutzt.

Die von „xf_visu“ benötigten FL_OBJECT-orientierten Daten werden dort in dem freien Pointer „u_vdata“ eingehängt. Es darf also von der sonstigen Xforms-Applikation nicht verwendet werden.

Bezeichnungsvereinbarungen

```
FL_OBJECT     =  ein Xforms-Objekt.
Xf_…          =  bezeichnen functions und structures, die zum xf-
                 Standard  gehören, dabei gilt:
xf_Object     =  ein Xforms-Objekt,inclusive der eingehängten
                  Structures, mit der xf_visu arbeitet.

Xfplc..       =  beziehen sich auf die Verbindung mit der SPS.
xf_inout..    =  stellen den Zusammenhang zwischen einem FL_OBJECT und
                 den zugehörigen PLC-Variablen her.
                 Sie erzeugen außerdem die notwendigen
                 Basisstructuren.
xf_add_..     =  definieren die Art der Bearbeitung der PLC-Variablen.
xf_set_..     =  Ändert die Vorbesetzung, die von den xf_inout ..
                 functions eingestellt werden.
xf_inout_type *stru  = die structure, die für jedes xf_Objekt angelegt
                 wird.
                 Sie beinhaltet alle Informationen über den Zusammenhang
                 zwischen dem FL_OBJECT und der adressierten Variablen
                 in der PLC.
```

Restriktionen gegenüber „freier“ Xforms-Programmierung:

Alle benutzen Forms müssen vor der zyklischen Bearbeitung durch fl_do_forms( ) bereits definiert sein „create_form_<name()“ und müssen während der gesamten Programmbearbeitung definiert bleiben : fl_free_form() ist nicht zulässig.

Grund:
Es werden in xf_visu einige Listen aufgebaut, die auf Objekte verweisen.
Das Löschen der sie beinhaltenden Form würde also zu Absturz führen.

Außerdem:
Visualisierungen sind Dauerläufer, durch das einmalige Erzeugen der Forms, der Objekte und der zugehörigen Datenbereiche wird die Gefahr einer Speicherfragmentierung reduziert.

Aufbau „Xforms“ oder „Xforms mit xf_visu“

Der übliche Aufbau eines trivialen xforms-Programms, ohne die Nutzung von xf_visu, würde hinsichtlich <proj>_main.s so aussehen:
(<proj> repräsentiert im Folgenden den konkreten Projektnamen.)

```
#include "<proj>.h"

int
main( int    argc,
      char * argv[ ] )
{
    FD_<main_form> *fd_<main_form>;  // << forms definition inside the main program


    fl_initialize( &argc, argv, 0, 0, 0 );
    fd_<main_form> = create_form_<main_form> ( );

    /* Fill-in form initialization code */

    /* Show the first form */

    fl_show_form( fd_<main_form>-><main_form>, FL_PLACE_CENTERFREE, FL_FULLBORDER,
"t" );

    fl_do_forms( );

    ..
    fl_free( fd_<main_form> );
    fl_finish( );
    return 0;
}
```

Dieses Programm wird im Wesentlichen so geändert: (Details weiter unten)

```
#include "<proj>.h"
#include xf_..specific files (s.b.) (1)

definition of background function  (2)

int
main( int    argc,
      char * argv[ ] )
{
   // FD_<main_form> *fd_<main_form>;  // now defined in „<proj>_data.h“ (3)


    fl_initialize( &argc, argv, 0, 0, 0 );
    fd_<main_form> = create_form_<main_form> ( );
    .. and create all other forms  (4)

    /* Fill-in form initialization code */
    initialize the xf_..system and the definition of accessed data blocks  (5)
    define all xf_objects       (6)
    /* Show the first form */
    fl_show_form( fd_<main_form>-><main_form>, FL_PLACE_CENTERFREE, FL_FULLBORDER,
        "t" );

    fl_do_forms( );

    shutdown xf-related stuff  (7)
    fl_free( fd_<main_form> );
    fl_finish( );
    return 0;
```

```
}
```

(1) :

Die xf-spezifischen #includes sind:

```
#include <forms.h>
#include "<proj>_data.h"          // access to global project data, ie. forms
#include "xf_visu/xfplc_std.h"    // part of xf_visu

// project specific xpm-grafics, i.e.
#include "valve.xpm"
...
#include "Fan_up.xpm"
#include "Motor.xpm"
```

(2) :

Alle Operationen des xf-Systems werden später dann ausgeführt, wenn xforms gerade nichts zu tun hat (.. also meistens).
Dann wird, falls definiert, die xforms-function **idle_cb(XEvent *ev,void *b)** aufgerufen. In ihr wird die „background()“ aufgerufen.
In „background()“ sind alle xf-bezogenen Bearbeitungen eingehängt.
(Im Idealfall nur eine function. (s.u.)

```
/* --------------- prototyping for main loop ------------------------- */
int background (void);

/* --------------- idle_callback ------------------------- */
int idle_cb(XEvent *ev,void *b)

{
  return background();
}
```

(3) :
Da die Forms für xf_visu aus verschiedenen Gründen global zugänglich sein müssen, werden sie in die **Datei <proj>_data.h** ausgelagert.

Diese Datei **<proj>_data.h** könnte dann so aussehen:

Datei **„<proj>_data.h“** :

```
    #include "<proj>.h"

    FD_<main_form> *fd_<main_form>;
     //.. and all project specific forms ..
```

Es ist sinnvoll, für die zu bearbeitenden Datenbausteine in der Datei **„<proj>_data.h“** ausserdem symbolische Namen anzulegen:

```
// defines for plc_blocks / DBs !!numbers have to start at 1 !!
// DB 200 db_OP_IN
#define OP_IN 1
// DB 201 db_OP_OUT
#define OP_OUT 2
// … and others ...
```

Diese #defines sind nicht zwingend, verbessern aber später den

Überblick:
Hier: (Beispiel s.o.)
ein DB für die Ein-/Ausgabe zur PLC: **OP_IN** als Block nummer 1
ein DB für die Ausgabe von der PLC: **OP_OUT** als Block nummer 2

(4) :
Alle forms des Projects müssen hier erzeugt werden, da das xf-system die darin enthaltenen FL_OBJECTs auch zur Datenhaltung benutzt.

(5) :
Der idle_callback wird gestartet:

```
    // initialise idle callback, this is the main loop of the background program
    fl_set_idle_callback(idle_cb,&dummy_data); // see above
```

Diese folgende Variable `bg_init_run` informiert background(), dass dies der erste Lauf ist ..falls noch Initialsierungen durchgeführt werden müssen:

```
    bg_init_run = 1; // tell background() : this is your first cycle
```

Die zentrale Datenhaltung dex xf-Systems wird intitialisierT:

```
    xfplc_init(); // initialise the basic data
```

Für alle zu bearbeitenden Datenbausteine wird Platz reserviert und die Verwaltungsbereiche werden initialisiert:
(Symbolischer Name „OP_IN“ : siehe oben Datei <proj>_data.h)

```
    /* ==== define the plc_blocks ( project specific DBs to be read or written ) =====
*/
    xfplc_block_init( OP_IN, /* number of this block Numbers have to start with 1 */
                      CPU_NR, /* CPU number to process */
                      200,  /* db_nr to process */
                      0,    /* starting DBB inside the block */
                      52,   /* size of this block in Bytes */
                      900,  /* [ms] if update requested */
                      1     /* update_req */   // 1 = always update
    );

    xfplc_block_init( OP_OUT, /* number of this block Numbers have to start with 1 */
                      CPU_NR, /* CPU number to process */
                      201,  /* db_nr to process */
                      0,    /* starting DBB inside the block */
                      1024, /* size of this block in Bytes */
                      900,  /* [ms] if update requested */
                      0     /* update_req */  // 0 = update on request
    );
       // next plc_block ..
```

(6) :
Alle xf_Objects werden definiert: Die Verbindung zwischen dem PLC-Datenbaustein (`xfplc_block`) und dem FL_OBJECT wird hergestellt, der Datentyp wird definiert.
Dies kann direkt hier geschehen, in diesem Beispiel werden aber alle Definitionen in einer Function **`link_xf_to_object ()`** zusammengefasst.

```
    /* ==  definition of the io-variables  ( project specific function  )=== */
    link_xf_to_object () ; // user defined: linking objects to plc_blocks see below
```

(7) :
Vor dem Programmende muß der durch das xf-system allozierte Speicher wieder zurückgegeben werden:

```
/*   shutdown xfplc stuff */
xfplc_end ();
```

Die **Grundstruktur eines xf_visu Programms** wird dann so aussehen:

Main Programm: Datei **<proj>_main.c** :

```
#include <forms.h>
#include "<proj>_data.h"          // access to global project data, ie. forms
#include "xf_visu/xf_const.h"     // part of xf_visu
#include "xf_visu/xfplc_std.h"    // part of xf_visu
#include "xf_visu/cmdSend.h"      // part of xf_visu
#include "s7types.h"              // convert S7-data types to computer types

// project specific xpm-grafics, i.e.
#include "valve.xpm"
...
#include "Fan_up.xpm"
#include "Motor.xpm"


/* --------------- prototyping for main loop ------------------------- */
int background (void);

/* --------------- idle_callback ------------------------- */
int idle_cb(XEvent *ev,void *b)

{
  return background();
}
```

Die function **main()** wird wie folgt erweitert:

```
int
main( int    argc,
      char * argv[ ] )
{
    fl_initialize( &argc, argv, 0, 0, 0 );

    /* ========== Create the forms ============ */
    fd_<main_form> = create_form_<main_form> ( );
        //.. and all project specific forms ..  // die bereits in <proj>_data.h
                                                // deklariert sind

    /* Fill-in form initialization code */


    // initialise idle callback, this is the main loop of the background program
    fl_set_idle_callback(idle_cb,&dummy_data); // see above
    bg_init_run = 1; // tell background() : this is your first cycle

    /* ========== initialise the xf_visu system ========== */
    xfplc_init(); // initialise the basic data



    /* ==== define the plc_blocks ( project specific DBs to be read or written ) =====
*/
    xfplc_block_init( OP_IN, /* number of this block Numbers have to start with 1 */
                      CPU_NR, /* CPU number to process */
                      200,  /* db_nr to process */
                      0,    /* starting DBB inside the block */
                      52,   /* size of this block in Bytes */
                      900,  /* [ms] if update requested */
                      1     /* update_req */   // 1 = always update
    );
```

```
    xfplc_block_init( OP_OUT, /* number of this block Numbers have to start with 1 */
                      CPU_NR, /* CPU number to process */
                      201,  /* db_nr to process */
                      0,    /* starting DBB inside the block */
                      1024, /* size of this block in Bytes */
                      900,  /* [ms] if update requested */
                      0     /* update_req */  // 0 = update on request
    );
       // next plc_block ..


    /* ==  definition of the io-variables  ( project specific function  )=== */
    link_xf_to_object () ; // user defined: linking objects to plc_blocks see below

    /* Show the first form */

    fl_show_form( fd_<main_form>-><main_form>,
            FL_PLACE_CENTERFREE, FL_FULLBORDER,  "t" );

    fl_do_forms( );

    ..
    /*   shutdown xfplc stuff */
    xfplc_end ();

    fl_free( fd_<main_form> );
    fl_finish( );
    return 0;
}


//-- * the background operations --*/
int background ( void )

{
    /* ==============  xfplc  operation ==================
    * MAIN PROGAM of xf_visu !!
    * plc_blocks (DBs) fetch and write data from ot to plc and display them
    * ================================================= */
    xfplc_main();

    // additional user defined background programs
    ….
    return 0;
}


// ============ user defined linkage to the FL_OBJECTS ====================

void link_xf_to_object ( void) ; // user defined: linking objects to plc_blocks

{
 xf_inout_type *stru;

    stru = xf_inout_create ( fd_f->box_1_1,    OP_OUT,174 );   (1)
       xf_add_show_fun ( stru ,  "%i", zeige_mat_bx1);         (2a)
   ...
     stru = xf_inout_create ( fd_f->x_soll,  OP_OUT, 4 );      (1)
       xf_add_show_auto ( stru, S7TYPE_INT,"%i" );             (2b)
  ...
}
```

Datenaustausch zwischen PLC und xf_visu – Grundstruktur --

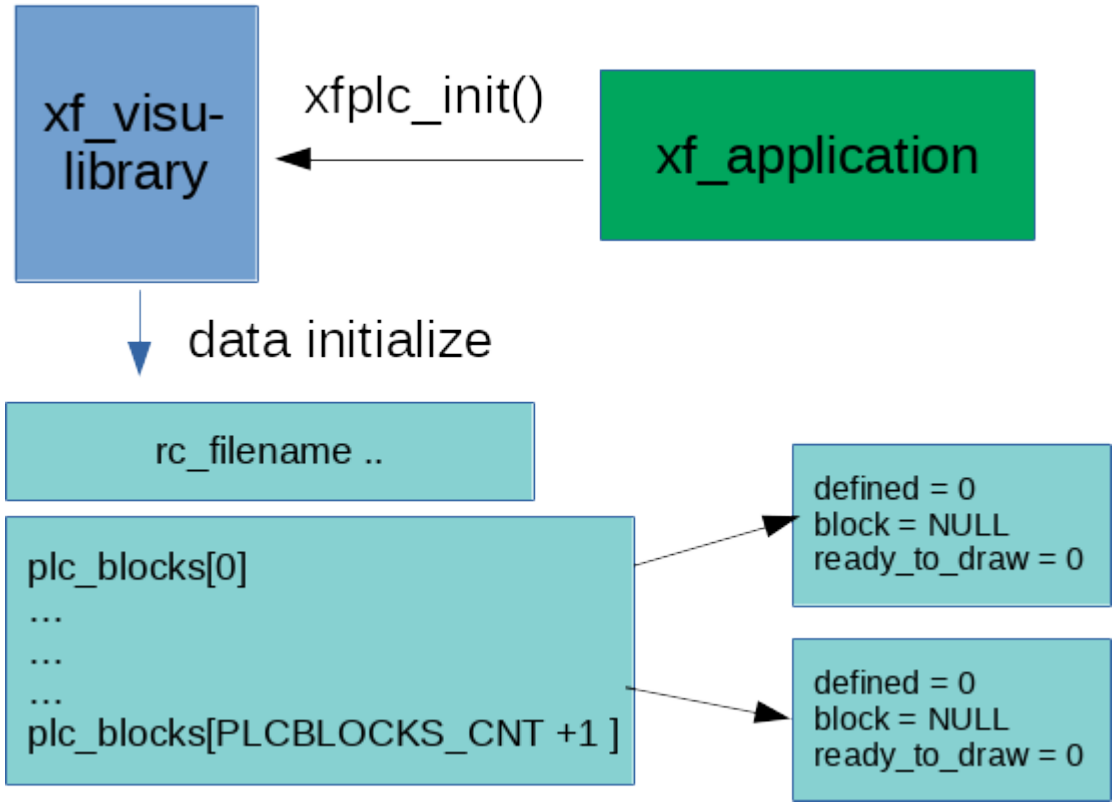
xfplc_init()

Zur Datenkommunikation wird ein rk*_server benutzt, die Auswahl erfolgt entsprechend den Kommunikationsmöglichkeiten der SPS.
Die Kommunikation des servers zur Simatic / den Simaticen wird im rc-file des Servers definiert. (siehe Handbuch des entsprechenden Servers)

Damit ihn „xf_visu“ zur Kommunikation benutzen kann, muss „xf_visu“ das rc-file mitgeteilt werden.

In der Datei „xfplc_std.c“ wird eine Definition „RCFILENAME“ erwartet, sie muss den vollständigen Namen mit Pfad des rc-files beinhalten.

Beim Initialisieren des „xf_visu“-Systems
(function : **void xfplc_init (void)**) wird der rc-filename gespeichert.



xfplc_init () initialisiert auch gleichzeitig die Verwaltung der zu bearbeitenden Datenbausteine. („plc_blocks“).
Es sind mit der Konstanten „**PLCBLOCKS_CNT**“ derzeit 100 „plc_blocks“ möglich, die Konstante kann in der Datei „xfplc_std.h“ geändert werden.

```
typedef struct {
   int block_nr;          /* number of this block */
   int cpu_nr;            /* CPU number to process, see rk*_server */
   int db_nr;             /* db_nr to process */
   int dbb_nr;            /* starting DBB inside the block */
   int dbb_len;           /* size of this block in Bytes */
```

```
   int defined;           /* block is defined : if 0 ignore */
   int update_cycle;      /* [ms] delay between 2 updates, if update requested */
   int update_time;       /* time for requested updates [ms] internal counter */
   int update_req_auto;       /* requested to update :
                                 set, if one of the forms, which contains an object
                                 which gets it's data from here is active */
   int trigger_upd;       /* trigger updating one time */
   unsigned char *byte;   /* pointer to the data inside this block */
   int rdy_to_draw;       /* block is updated = 1 , idle = 0 , error = < 0 */
   int user_update_req;       /* requested to update :
                                 normally 0 but can be manipulated by user program */
   FL_OBJECT  **oblist_ptr;   /* pointer to the array of *objects associated with this
                                 data block */
   int        ob_list_cnt;    /* count of objects associated with this data block */
   int        ob_list_max;    /* maximum count of objects actually associated with
                                 this data block */
   FL_FORM   **formlist_ptr;  /* pointer to the array of *forms associated with this
                                 data block */
   int        form_list_cnt;  /* count of forms associated with this data block */
   int        form_list_max;  /* maximum count of forms actually associated with this
                                 data block */
} Plcblock_list_Type;

Plcblock_list_Type plc_blocks[PLCBLOCKS_CNT +1 ];
```

Datenaustausch zwischen PLC und xf_visu
– einen plc_block (Datenbaustein) definieren –

Wie aus der function main() ersichtlich, werden danach die plc_blocks definiert:

Prototype:

```
void xfplc_block_init(    int block_nr, /* number of this block */
                  int cpu_nr, /* CPU number to process */
                  int db_nr,  /* db_nr to process */
                  int dbb_nr, /* starting DBB inside the block */
                  int dbb_len, /* size of this block in Bytes */
                  int update_cycle, /* update time [ms] */
                  int always_update /* requests updating always */
);
```

Dabei gilt:

block_nr : Die PLC - Blocks deren durchnummeriert, die Nummerierung startet mit 1.

Zur Erhöhung der Übersichtichkeit ist es sinnvoll, den Blocknummern mit #define symbolischen Namen zuzuordnen. z.B: #define DB_EINGABE 1.

Es erhöht die Arbeitsgeschwindigkeit, die Blocknummern ohne Lücken zu definieren, da dadurch Leerschleifen vermieden werden.

cpu_nr : Da die rk*_server den Zugriff auf mehrere CPUs erlauben:

hier die gleiche Nummer, die der entsprechenden CPU im rc-file des rk*_servers zugeordnet ist.

db_nr : Nummer des zu bearbeitenden Datenbausteins.

dbb_nr : Nummer des Bytes in dem Datenbaustein, ab der dieser Block beginnt.

dbb_len : Anzahl der zu bearbeitenden Bytes in dem DB [db_nr]

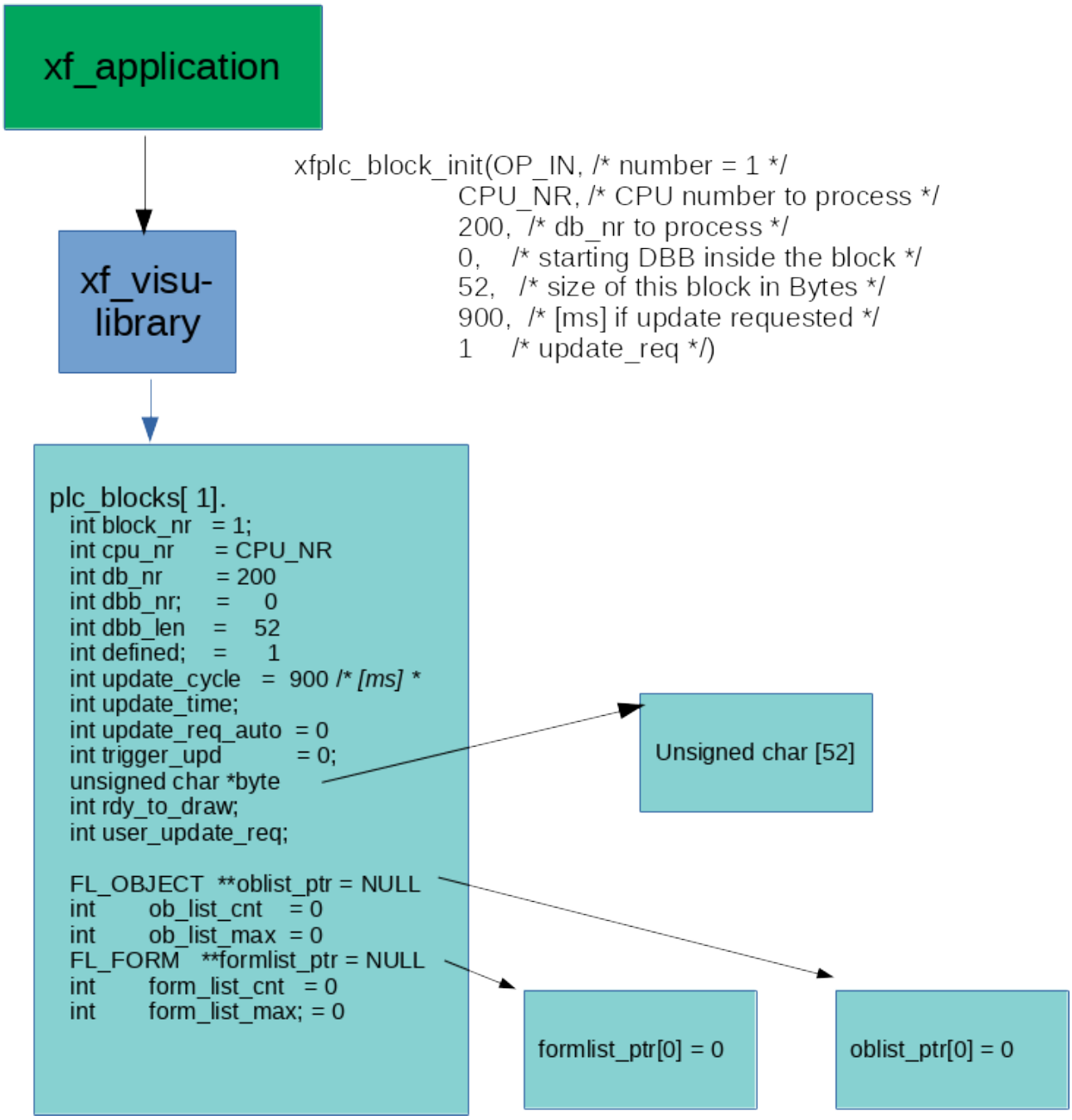
ab DBB [dbb_nr].

update_cycle : Zeitabstand in Millisekunden zwischen 2 Aktualisierungen

… falls der Block aktualisiert wird.

always_update : = 1 : Der Block wird immer in dem durch „update_cycle“ vorgegebenen Takt aktualisiert, unabhängig davon ob aktuell darin enthaltende Variablen sichtbar sind.

= 0 : Der Block wird nur aktualisiert, wenn darin enthaltene Variablen aktuell sichtbar sind.



xfplc_block_init() erzeugt den zur Ablage der Daten notwendigen Speicherplatz.
Außerdem werden 2 dynamische Listen initialisiert:
strulist_ptr : Darin werden später von xf_visu automatisch Pointer auf die „xf_inout_str“ der Objekte eingehängt, die diesen „plc_block“ nutzen.

formlist_ptr : Darin werden später von xf_visu automatisch Pointer auf die Forms eingehängt, die Objekte beinhalten, die diesen „plc_block“ nutzen.

Interaktion mit dem Bildschirm

wird mit FL_OBJECTS durchgeführt.

Welche xforms – Object- Klasse soll benutzt werden ?

Ausgabe- Objekte:

Für **Text-Ausgabe**-Felder (Strings, Zahlen) sollte generell „**text**“ benutzt werden.
Für grafische Ausgaben, z.B. Bitmaps, die umgeschaltet werden, am besten
„**bitmap**“ oder „**pixmap**“.
Eigentlich ist alles möglich, da mit einfachen selbsterstellten Ausgabe-functions alle Typen und Kombinationen daraus benutzt werden können so lange xforms eine Manipulation dieser Objekte ermöglicht.

Das geht hin bis zum Bildbaustein und grafischem Unterprogramm.

Eingabe und Ein-/Ausgabe– Objekte:

Für Ein-/Ausgabe– Objekte kann eigentlich jeder Object-Typ benutzt werden, der eine Eingabe ermöglicht und einen „callback“ auslösen kann. Üblicherweise ist das, bei Tastatureingabe, für Text- und numerische Ein-/Ausgaben: „input“.

Sonderfall Touchdisplay:
Für textuale Eingaben an einem Touchdisplay ohne eine vom System bereit gestellte virtuelle Tastatur stellt „xf_visu“ eine eigene virtuelle Tastatur bereit.
Alle Ein-/ausgabefelder müssen dann als „button“ definiert werden.
Die Betätigung diese „buttons“ startet dann die eingebaute Tastatur.
Näheres dazu weiter unten.

Eingabe – Objekte:

Als reine Eingabe-Objekte stehen aktuell nur Bit-orientierte Eingaben zur Verfügung. Sie werden normalerweise mit der Xforms-Objektklasse „button“ versorgt.

Verbindung des Bildschirms (FL_OBJECT) mit der PLC

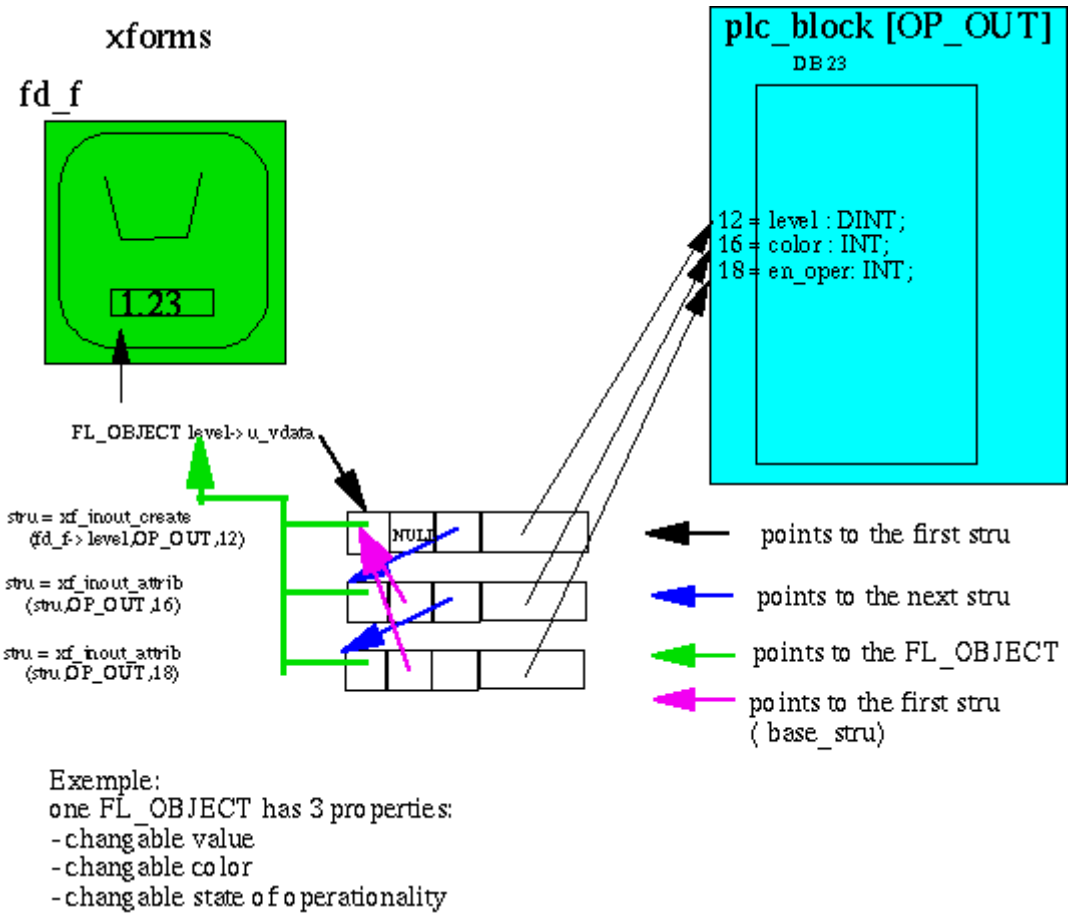
Ausgabe-, Ein-/Ausgabe- und Eingabe-Objekte
sind hinsichtlich der Nutzung der Grundstruktur sehr ähnlich.
Diese wird an Hand eines Ausgabe-Objektes erklärt.

Voraussetzung für alle Typen:

- Es muss bereits das mit **fdesign** erzeugte FL_OBJECT existieren.
  Das Objekt muss einen Namen haben damit „xf_visu“ es adressieren kann.
  Dies gilt für alle Objekte, die „xf_visu“ (xf_Objects) nutzen.

- Es muss bereits der plc_block vorhanden sein, denn er soll ebenfalls referenziert werden.

Die Funktionen **xf_inout_create()** und **xf_inout_attrib()** stellen die Verbindung zwischen dem FL_OBJECT und dem plc_block her:



Exemple:
one FL_OBJECT has 3 properties:
- changable value
- changable color
- changable state of operationality

Definition eines Objekts in xf_visu

(Die oben in der erweiterten xforms-Grundstruktur enthaltene function **link_xf_to_object ()**(siehe oben, in main()) ist eigentlich obsolet, die darin enthaltenen projektorientierten Befehle könnten auch direkt in main() programmiert werden. Es erhöht aber die Übersichtlichkeit wenn die Objekt-Definitionen zusammengefasst sind.)

Die Definition eines xf_visu – Objekts setzt sich aus mindestens 2 Zeilen Code zusammen:

Verbindung ?
  xf_inout_type ***xf_inout_create (FL_OBJECT *obj, int block_nr, int block_offs)**
Was tun ?
   xf_add_<specific_function_for_this_object> (xf_inout_type ***stru,...<handling>)**

(und gegebenenfalls
   xf_<optional_operations> (FL_OBJECT *obj , …)
   xf_<optional_operations> (FL_OBJECT *obj , …)
)

Zuerst zur **Verbindung**:

Die function **xf_inout_create ()** ist für alle „xf-Objekte“ gleich.
Sie stellt nur die Verbindung zwischen dem xforms – FL_OBJECT, einem plc_block, und dem Beginn der zum FL_OBJECT gehörenden Daten her.
Dabei wird eine structure **xf_inout_type stru** angelegt und in dem referenzierten FL_OBJECT **in →u_vdata** ein darauf verweisender Pointer gesetzt.
Umgekehrt zeigt ein Pointer in stru auf das FL_OBJECT.
In der structure stru wird ausserdem für dem plc_block die Nummer und die Byte-Start-Adresse gespeichert.
Beispiel: (von oben:)

```
   stru = xf_inout_create ( fd_f->x_soll,  OP_OUT, 4 );
```

stellt die Verbindung zwischen dem Xforms-Object „fd_f->x_soll“ und dem PLC-Datenblock „OP_OUT“ und dort dem Byte 4 (ff) her.

(* XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX >>>>>>>>>>>>>>>>>>>>>>>> *)
Die function **xf_inout_attrib()** ermöglicht es, dem FL_OBJECT weitere Eigenschaften hinzuzufügen,zum Beispiel ein aus der PLC gesteuerte Farbänderung, die Steuerung der Bedienbarkeit usw.
Es können beliebig viele **xf_inout_attrib ()** angehängt werden, es wird jeweils eine zusätzliche structure **xf_inout_type stru** erzeugt und als verkettete Liste an die mit **xf_inout_create ()** erzeugte structure angehängt.
(Diese Speicherplatzverschwendung ist angesichts des Speicherausbaus selbst bei einem aktuellen Raspberry Pi absolut akzeptabel.)

(<<<<<<<<<<<<<<<<<<< XXXXXXXXXXXXXXXXXXXXXXXXX)

Die zweite function bestimmt die Art der Verarbeitung:
- Ausgabe, Ein-Ausgabe, Eingabe.
- Byte,Char,INT,REAL ...

Dazu mehr weiter unten.

Weitere Variable zur Steuerung des FL_OBJECTS

Mit ihr wird ein weiterer Speicherblock an die durch **xf_inout_create ()** erzeugte structure angehängt, der dann mit weiteren

xf_add_<specific_function_for_this_object> (xf_inout_type ***stru,...<handling>)**

Aufrufen weitere Aufträge übergeben werden können.

Es können beliebig viele **xf_inout_attrib ()** angehängt werden.

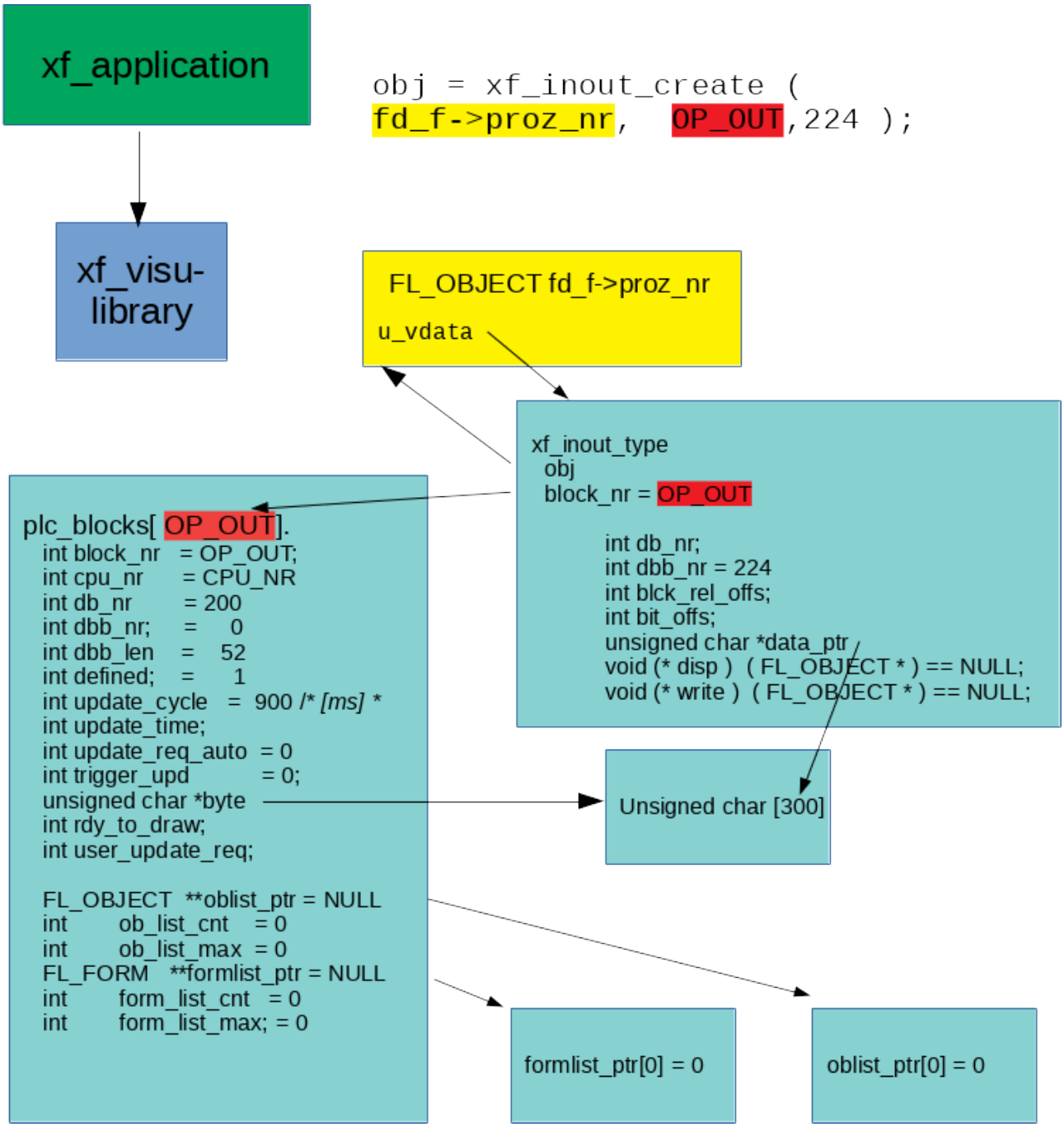
Der dabei benötigte Speicher für den Vergleich auf Änderung befindet sich in der mit ***xf_inout_attrib ()** angehängten structure, die Arbeitsdaten aber in der Regel in der von **xf_inout_create ()** erzeugten Structure.
Näheres entscheidet die anschließend mit **xf_add_..()** angehängte Structure.

***xf_inout_attrib ()** legt den gleichen Structure-Typ an der auch von **xf_inout_create ()** angelegt wird.

Die xf_add..() functions beziehen sich also auf die structure, die durch die vorausgegangene **xf_inout_create ()** oder ***xf_inout_attrib ()** - Function erzeugt wurde.
Um diese zu finden, müssen sie sich intern, von FL_OBJECT kommend, durch die verlinkte Listen der **xf_inout_type**-Structures durchhangeln.
Die xf_add..() functions werden aber nur beim Anlauf durchlaufen, beim Betrieb erfolgt dann der Verweis direkt auf die richtige structure.

Verweis auf ‚plc_block' und Erzeugung des object-spezifischen Speicherbereichs : FL_OBJECT *xf_inout_create ()

„xf_visu" benötigt für jedes FL_OBJECT Speicherplatz, der diesem Objekt zugeordnet ist.
In FL_OBJECT wird dazu der Pointer „u_vdata" genutzt.



xf_inout_type ***xf_inout_create (FL_OBJECT *obj, int block_nr, int block_offs);**

erzeugt eine structure vom typ „xf_inout_type". Diese wird mit dem xforms-FL_OBJECT in beide Richtungen verlinkt.

Das FL_OBJECT (xforms) hat jetzt also Information, in welchem „plc_block“ seine Daten sind, bzw. wo sie anfangen.

Hinweis: Unabhängig von der Startadresse im plc_block[].dbb_nr: der bei **xf_inout_create ()** übergebene Parameter **int block_offs** bezieht sich immer auf das DBB 0 des Datenbausteins. Dies erleichtert die Erkennung des Zusammenhangs zwischen der DB in der SPS und der Parametrierung der Verbindung.

Bearbeitungsmethode : xf_add_… ()

Noch wird nichts bearbeitet, denn bisher ist nur bekannt, wo die Daten herkommen würden.
Es ist aber nicht klar, wie sie interpretiert werden sollen.
Das zweite, die Bearbeitungsmethode, wird von den xf_add_.. functions festgelegt.

Die Namen der Functions sind standardisiert:
int xf_add_<dir>_<typ> .. (**xf_inout_type *stru** , . . .)
wobei : <dir> = **show** für Ausgabe-Objekte
inout für Ein-Ausgabe-Objekte
in für Eingabe-Objekte
und <typ> = **fun** für individuelle Bearbeitungsmethode
auto für PLC-Datentypen

Ausgabe:

Die universelle Ausgabe-function **xf_add_show_fun ( )**

```
int xf_add_show_fun ( xf_inout_type *stru , void *fmt, void (*disp)())
```

stru verweist auf die xf_inout_type - Structure, die in dem zu bearbeitenden Xforms-Objekt eingehängt ist.
*fmt verweist auf einen optionalen Formatstring, er wird intern nicht kopiert. Bei Nicht-Benutzung: NULL
*disp verweist auf die eigentliche Ausgabefunktion.
Sie muss vom Typ **void *disp (FL_OBJECT *obj)**
sein.

Mit **xf_add_show_fun ()**
kann alles bearbeitet werden denn der Benutzer legt fest, wie die übergebenden Werte zu bearbeiten sind.
Falls das anzuzeigende Object z.B. ein Object mit zusätzlichen Komponenten ist, **fl_add_child()** , kann zum Beispiel eine ganze structure abgeholt und interpretiert werde,

Hier als Beispiel die Ausgabe eines Textes in das Xforms-Object **fd_f->hint** in Abhängigkeit eines Integer-Wertes aus der SPS,
aus dem plc_block[OP_OUT], ab Byte 92.

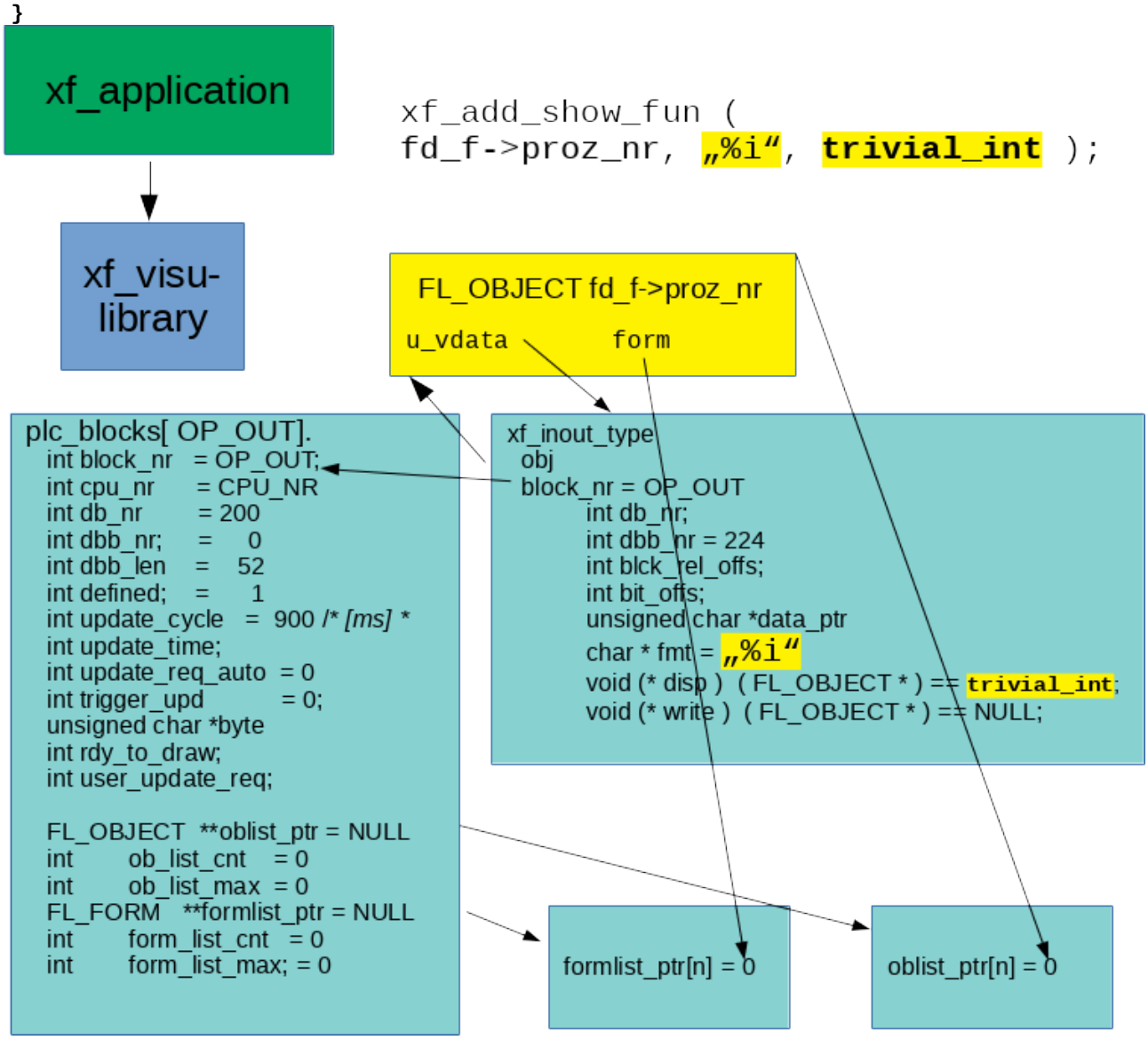
```
stru = xf_inout_create ( fd_f->hint, OP_OUT, 92 )
xf_add_show_fun ( stru,  NULL, zeige_hint_line);
```

```
void zeige_hint_line (FL_OBJECT *ob )

{
    xf_inout_type *stru;
    int i;
    char hstr[100];

    stru = obj→u_vdata;  // points to the struct, generated by xf_inout_create()
    i = get_s7INT(stru→data_ptr); // points to  plc_block[OP_OUT].byte[92]
    switch (i) {
        case 0     : strcpy(hstr,“OFF“); break;
        case 1     : strcpy(hstr,“ON“); break;
        default:   : strcpy(hstr,“Don‘t Know“); break;
```

```
    }
    fl_set_object_label ( ob,hstr);
}
```



zusammen:

```
    stru = xf_inout_create ( fd_f->hint_line, OP_OUT,92 );
      xf_add_show_fun ( stru,  "%i", trivial_int);
```

(Bemerkung: **xf_inout_create ()** gibt die in das FL_OBJECT **fd_f-> hint_line** eingehängte Structure xf_inout_type * stru zurück, damit man in der folgenden Zeile Tipparbeit spart.)

```
xf_add_show_fun ( obj,  "%s", zeige_hint_line);
```

Format:

```
int xf_add_show_fun ( xf_inout_type * stru, void *fmt, void (*disp)()) ;
```

Die Parameter sind:
- die zuvor von **xf_inout_create ()** befüllte und auf das zu bearbeitende FL_OBJECT obj verweisende Structure,
- ein beliebig zu nutzender Pointer vom typ VOID

- ein Pointer auf eine function
  void disp (FL_OBJECT *ob)
  Diese function versorgt das FL_OBJECT mit den Daten aus der SPS.

Trivialisiert könnte eine solche function so aussehen:

```
void trivial_int ( FL_OBJECT *obj)

{
   char hstr[20];
   int i;
   xf_inout_type *stru;

   stru = obj→u_vdata;  // points to the struct, generated by xf_inout_create()
   i = get_s7INT(stru→data_ptr); // points to  plc_block[OP_OUT].byte[92]
   sprintf(hstr, stru->fmt, i);  // points to the string, received bei *fmt
   fl_set_object_label (obj, hstr);
}
```

xf_add_show_fun () führt aber die Funktion nicht aus!
Es werden lediglich die Pointer **void *fmt** und **void (*disp)()** in der Structure **obj→u_vdata** eingetragen damit das ausführende Programm die Information hat, wie das Objekt zu bearbeiten ist.

Ein in der Hauptschleife eingehängtes
xf_upd_obj (FL_OBJECT *obj)
könnte bereits die Aktualisierung durchführen. Das wäre aber nicht sinnvoll, denn dann würde dieses Objekt immer bearbeitet, unabhängig von Sichtbarkeit und Aktualität der Daten.

Deshalb hat **xf_add_show_fun ()** noch weitere Aufgaben:
xf_add_show_fun () trägt noch zusätzlich den Pointer stru, in dem ein Verweis auf das zu bearbeitende FL_OBJECT *obj ist, in die von **xfplc_block_init()** erzeugte Liste **oblist_ptr[] ein**.

Außerdem wird ein Pointer auf die Form, zu der das Objekt gehört, in die von **xfplc_block_init()** erzeugte Liste **formlist_ptr[]** eingetragen.

Der **plc_block[n]** (Datenbaustein-Bereich …) hat also die Information, in welchen Forms sich Objekte befinden, die aus ihm versorgt werden.

Die in dem Hintergrundprogramm background() aufgerufene function **xfplc_main()** sorgt dafür dass die Aktualisierung der Objekte durchgeführt wird.

Immer wenn kein Kommunikationsvorgang läuft, werden alle plc_block[n] überprüft, ob eine der in ihrer **plc_block[n].formlist_ptr[]** eingetragenen Forms sichtbar ist. Wenn ja, wird dieser **plc_block[n]** zum Lesen aus der SPS aktiviert.
Wenn die Daten gelesen wurden, wird für diesen **plc_block[n]** die Liste **oblist_ptr[]** durchgearbeitet und alle Objekte die in sichtbaren Forms sind werden mit Hilfe der von **xf_add_show_fun ()** eingetragenen function ***disp)()** aktualisiert.

Automatische Zuordnung bei PLC-Basistypen:

xf_add_show_auto ( )

xf_add_show_fun () ist universell einsetzbar, würde aber deshalb auch für jeden PLC-Datentyp ein spezielle, in **void (*disp)()** einzutragende function benötigen.
Bei der Menge der Datentypen die z.B. die S7-1500 bietet, wäre dies unpraktisch.
Zur Bearbeitung der S7-Basistypen mit Ausnahme der Bool-Variablen gibt es also noch eine zusätzliche, spezialisierte function:

int xf_add_show_auto (xf_inout_type * stru, int type_nr, char *fmt)

Die Parameter sind:
- die zuvor von **xf_inout_create ()** befüllte und intern auf das zu bearbeitende FL_OBJECT obj verweisende Structure,
- der PLC-Datentyp, dazu sind Konstante vorbereitet.
  Siehe Tabelle 1 **„PLC-Datentypen“.**
  Dies entspricht dem in der PLC benutzten Datentyp.
- ein Pointer auf den Formatstring für die Ausgabe als Text, er wird intern nicht kopiert.

Unterschied zwischen xf_add_show_fun () und xf_add_show_auto () :
xf_add_show_auto () versorgt, dem PLC-Datentyp **type_nr** entsprechend, den Pointer *disp“ automatisch mit einer von „xf_visu“ bereitgestellten function, die typabhängig arbeitet.

Tabelle 1:
PLC-Datentypen

Die Tabelle enthält eine Auflistung der PLC-Datentypen und der zugeordneten Konstanten-Namen sowie ein Beispiel für den Formatstring.

PLC-Datentyp	Konstante	Formatstring Vorschlag
-- none --	S7TYPE_NIL=0,	
BOOL	S7TYPE_BIT,	
BYTE	S7TYPE_BYTE,	%0x
CHAR	S7TYPE_CHAR,	%c
WORD	S7TYPE_WORD,	%0x
INT	S7TYPE_INT,	%i
DWORD	S7TYPE_DWORD,	%0x
DINT	S7TYPE_DINT,	%i
REAL	S7TYPE_REAL,	%.2f
DATE	S7TYPE_DATE,	%m.%d.%y
TIME_OF_DAY	S7TYPE_TOD,	%H:%M:%S
TIME	S7TYPE_TIME,	%H:%M:%S
S5TIME	S7TYPE_S5TIME,	%i
DT	S7TYPE_DT,	%m.%d.%y-%H:%M:%S
STRING	S7TYPE_STRING,	%s
COUNTER	S7TYPE_COUNTER,	%i
DTL	S7TYPE_DTL,	
LDT	S7TYPE_LDT,	
LTIME	S7TYPE_LTIME,	
SINT	S7TYPE_SINT,	%i
USINT	S7TYPE_USINT,	%i
UINT	S7TYPE_UINT,	%i
UDINT	S7TYPE_UDINT,	%i
LWORD	S7TYPE_LWORD,	%0x
ULINT	S7TYPE_ULINT,	%li
LINT	S7TYPE_LINT,	%li
LREAL	S7TYPE_LREAL	%.3lf

```
Beispiele:
xf_add_show_auto (obj,S7TYPE_INT, "%i" );
xf_add_show_auto (obj,S7TYPE_INT, "%0x" );
xf_add_show_auto (obj,S7TYPE_DATE,"%d.%m.%y" );
```

Erweiterte Functionalität:

Die Funktionalität von **xf_add_show_auto ( )** kann durch folgende functions erweitert werden :

int xf_set_scale_auto()

Verändert die Skalierung der Darstellung. Macht z.B. aus einer S7TYPE_INT, die [mm] repräsentiert, eine Darstellung in [cm].

```
Beispiel:
xf_add_show_auto (obj,S7TYPE_INT, "%.1f" );
xf_set_scale_auto(obj,0.1);

==> in PLC:  135  auf Bildschirm 13.5
```
!!! Vorsicht:
xf_set_scale_auto() macht intern aus der INT-Zahl eine Float-Zahl, das Ausgabeformat muss dazu passen. Deshalb hier: "%.1f"

xf_mod_color ()

verändert die vordefinierten Farben. Die Farben col1,col2,lcol entsprechen dabei den gleichnamigen Farben in Xforms (siehe fl_set_object_color () und fl_set_object_lcol ())

Für verschiedene Zustände sind Farben vordefiniert.
(Projektweite Einstellung: ‚xf_visu/xf_appl_config.h‘).

mit

```
int xf_mod_color ( FL_OBJECT *obj,int color_set,
    FL_COLOR col1, FL_COLOR col2, FL_COLOR lcol);
```

kann diese Voreinstellung xf_object - bezogen überschrieben werden.

Die Zustände (Parameter ‚color_set‘) sind:

XF_COLS_NOCON,	keine Averbindung zur SPS
XF_COLS_ERRINP	fehlerhafte Eingabe, nicht in SPS geschrieben
XF_COLS_DIS;	Bedienung gesperrt
XF_COLS_DEF,	Vorgabe: das sind die mit xforms definierten Farben
XF_COLS_RANGE,	Ausser Bereich
XF_COLS_FALSE,	Boolsche Ein-/Ausgaben : FALSE-Zustand
XF_COLS_TRUE	Boolsche Ein-/Ausgaben : TRUE-Zustand

```
 Farben:
 col1    Wenn nicht im Fokus
```

col2 Wenn im Fokus
lcol Schriftfarbe

Die für einen Zustand aktuell eingestellten Farben können mit

```
int xf_get_color ( FL_OBJECT *obj, int color_set,
    FL_COLOR *col1, FL_COLOR *col2, FL_COLOR *lcol);
```

gelesen werden.

Ausgabe von Bool-Werten

xf_add_show_auto_bool ( )

Abgrenzung:
Dies betrifft eigenständige Bool‘sche Werte, denen exklusiv ein FL_OBJECT zugeordnet ist.

int xf_add_show_auto_bool (FL_OBJECT *obj, int bit_nr, void (*disp) ())

im vorhergehenden **xf_inout_create ()** wurde zwar die Byte-Adresse definiert, die Bit-Adresse fehlt noch. Das macht int bit_nr.

Normalerweise geht es um einen Farbumschlag. (Disp = NULL)
Vordefiniert sind die Farben

```
  fg_true  = FL_GREEN;
  bg_true  = FL_GREEN;
  fg_false = FL_RED;
  bg_false = FL_RED;
```

Erweiterte Functionalität:

Diese können aber mit einer danach aufgerufenen function

```
int xf_set_color ( xf_inout_type *stru,int color_set,
                   FL_COLOR col1, FL_COLOR col2, FL_COLOR lcol)
```

verändert die vordefinierten Farben.
Die Farben col1,col2,lcol entsprechen dabei den gleichnamigen Farben in Xforms (siehe fl_set_object_color () und fl_set_object_lcol ())

Für verschiedene Zustände sind Farben vordefiniert.
(Projektweite Einstellung: ‚xf_visu/xf_appl_config.h‘).

mit
xf_set_color ()

kann diese Voreinstellung xf_object - bezogen überschrieben werden.

Die Zustände (Parameter ‚color_set‘) sind:

```
   XF_COLS_NOCON,     keine Averbindung zur SPS
   XF_COLS_ERRINP     fehlerhafte Eingabe, nicht in SPS geschrieben
   XF_COLS_DIS;       Bedienung gesperrt
```

```
   XF_COLS_DEF,      Vorgabe: das sind die mit xforms definierten
                     Farben
   XF_COLS_RANGE,    Ausser Bereich
   XF_COLS_FALSE,    Boolsche Ein-/Ausgaben : FALSE-Zustand
   XF_COLS_TRUE      Boolsche Ein-/Ausgaben : TRUE-Zustand

 Farben:
 col1    Wenn nicht im Fokus
 col2    Wenn im Fokus
 lcol    Schriftfarbe
```

Die für einen Zustand aktuell eingestellten Farben können mit

```
int xf_get_color ( xf_inout_type *stru, int color_set,
    FL_COLOR *col1, FL_COLOR *col2, FL_COLOR *lcol)
```

gelesen werden.

Wenn ein Farbumschlag als Anzeige nicht reicht, muss, wie bei **xf_add_show_fun ()**, eine function für die Anzeige geschrieben werden.

Zusätzliche, abgeleitete Operationen:

Jeder Ausgabe- und Ein-Ausgabe-Funktion kann zusätzliche Operationen angehängt werden, die intern nach der in *disp() eingehängten Function ausgeführt wird.

```
xf_also_stru_type *xf_do_also ( xf_inout_type *stru,
    void (*do(),
    int val_1, int val_1, void *p1,void *p2)
```

Es können beliebig viele xf_do_also () Aufrufe an ein Objekt angehängt werden.

Die Ausführung wird mit den gleichen Kriterien getriggert, mit denen auch die Ausgabe-Funktion *disp() getriggert wird.

Der Unterschied zu
int xf_add_show_fun (xf_inout_type *stru, void *fmt, void (*disp)()) ;

besteht darin, **dass** **xf_add_show_fun ()** die automatische Anzeigefunktion ersetzt, **xf_do_also ()** aber hinter der Ausgabefunktion zusätzliche beliebige Funktionen ausführt.

Die übergebenen Funktionen können frei definiert werden, solange die folgende Schnittstelle eingehalten wird:
void (function) (xf_also_stru_type* also_ptr, int val_1, int val_1, void *p1,void *p2)
Über den übergebenen Pointer auf
xf_also_stru_type* also_ptr
hat die aufgerufene function auch Zugriff auf das FL_OBJECT und die Basis-Structur *stru.

Die aufgerufene function xf_do_also () bringt einen Pointer auf die intere structure vom Typ xf_also_stru_type zurück.

Darin kann der Anwender weiteren Speicher einhängen.
Falls der allozierte Speicher nur aus einem Block besteht, braucht free_u_spec() nicht definiert zu werden, der Speicher wird mit „fl_free()“ automatisch zurückgegeben.

Bei komplizierteren, verlinkten structuren muss sich der Anwender allerdings selbst um die Freigabe der Speicherblöcke kümmern und die Anwender -definierte function free_u_spec() einhängen.

Zur Verdeutlichung:
Unterschied xf_inout_attrib .. und xf_do_also ..

xf_inout_create() und xf_inout_attrib() bearbeiten das **gleiche FL_OBJECT** in Abhängigkeit von **verschiedenen** PLC-Variablen.

Es könnte z.B. einem FL_OBJECT, das einen Füllstand anzeigt, mit xf_inout_create() der Wert und mit xf_inout_attrib() der Farbumschlag, z.B. in Abhängigkeit von einem Grenzwert übergeben werden.

xf_do_also () bezieht sich auf die zuvor zugewiesene **PLC-Variable** und führt bei deren Änderung Funktionen aus, die sich nicht auf das zuvor von **xf_inout_create ()** referenzierte FL_OBJECT beziehen.

Beispiel: Userlevel
xf_inout_create() stellt den Zusammenhang zwischen der zugehörigen PLC-Variablen und seiner Anzeige durch ein FL_OBJECT auf dem Bildschirm her.
xf_do_also () überwacht ebenfalls diese PLC-Variable und ändert in deren Abhängigkeit die Bedienbarkeit von vielen unterschiedlichen FL_OBJECTs.

Ein-/Ausgabe-Objekte

Ein-/Ausgabe-Objekte sind im Prinzip Ausgabe-Objekte mit zusätzlicher Eingabe-Funktionalität. Sie benötigen daher eine zusätzliche Methode, wie die Daten zur SPS zu schreiben sind.

xf_add_show_fun () und **xf_add_show_auto ()** versorgen beide, wenn auch unterschiedlich, den Pointer **void (*disp)()** in der am FL_OBJECT angehängten structure.
Dieser referenziert die Funktion, die zur Ausgabe auf den Bildschirm benutzt wird.

Anwender-definierte Ein-/Ausgabe :

Beispiel:

universelle Ein-/Ausgabe function xf_add_inout_fun ( )

Zum Vergleich: Ausgabe

```
int xf_add_show_fun ( xf_inout_type *stru, void *fmt, void (*disp)()) ;
```

Ein-/Ausgabe

```
int xf_add_inout_fun ( xf_inout_type *stru, char *fmt, void (*disp)(),
                       void (*write)() );
```

xf_add_inout_fun () benötigt einen zusätzlichen Parameter, mit dem die function zur Dateneingabe übergeben wird, **oid (*write)()** .
Da die function **xf_add_inout_fun ( )**
universell einsetzbar ist, muss auch die Eingabe-Reaktion individuell festgelegt werden. Das bedeutet:
Das zugeordnete FL_OBJECT benötigt eine callback -function.
Diese kann direkt im forms-designer oder auch im Anwenderprogramm definiert werden.

Sonderfall: Touch-Display und nicht-PLC-Datentyp konforme Zahleneingabe:

Konkreter Fall:
In der Simatic beinhaltet eine Integer-Zahl Sekunden, die aber als [minuten]:[sekunden] dargestellt und geändert werden sollen.
Dies ist mit **xf_add_inout_auto ()** nicht mehr darstellbar, es muss **xf_add_inout_fun ()** benutzt werden.

Als Ein-Ausgabe-FL_OBJECT wird, da Touch-Display, ein Button vom Typ NORMAL_BUTTON benutzt.
Als Ausgabebereich kann das Label des Buttons benutzt werden.
Als callback des Buttons kann die im xf-System enthaltende function **cb_touch_inp_start ()** verwendet werden.
Damit wird die virtuelle Zahlentastatur startet.
Diese speichert einen Pointer auf das FL_OBJECT, von der sie gestartet wurde. Dadurch ist die Rückgabe des Editierergebnisses zum xf_Object und die Weiterverarbeitung möglich.
Näheres in der Beschreibung zu **cb_touch_inp_start ().**

Ein-/Ausgabe bei PLC-Basistypen

Die function zur Eingabe von **PLC-Basistypen** ist

xf_add_inout_auto ( )

int xf_add_inout_auto (xf_inout_type *stru, int type_nr, char *fmt);

und benötigt diesen Parameter nicht. Der zu bearbeitende PLC-Basistyp wird als Konstante in **int type_nr** übergeben, genau wie bei **xf_add_show_auto () .**
Die Konstanten sind in der Tabelle 1 beschrieben.
Wie **xf_add_show_auto ()** bearbeitet auch **xf_add_inout_auto ()** kein BOOL-Variablen.

xf_add_inout_auto () versorgt intern automatisch die Pointer *disp“ und „write“ mit von „xf_visu“ bereitgestellten functions, die typabhängig arbeiten.

Das xf_Object wird auch automatisch mit den Grenzwerten versorgt, die für den in **int type_nr** definierten Typ gelten.
z.B: S7TYPE_USINT : 0.. 255, da gleich unsigned char.
S7TYPE_SINT : -128.. 127, da gleich signed char.

Sonderfall: Touch-Display

Als Ein-Ausgabe-FL_OBJECT wird ein Button vom Typ NORMAL_BUTTON benutzt.
Ein callback ist nicht notwendig, den versorgt **xf_add_inout_auto ( )** automatisch mit der internen function **cb_touch_inp_start ().**

Funktionserweiterungen:

Die Funktionalität von **xf_add_inout_auto ( )** kann durch folgende functions erweitert werden :

int xf_set_scale_auto()

Verändert die Skalierung der Darstellung. Macht z.B. aus einer S7TYPE_INT, die [mm] repräsentiert, eine Darstellung in [cm].

Beispiel:
xf_add_show_auto (obj,S7TYPE_INT, "%.1f");
xf_set_scale_auto(obj,0.1);

==> in PLC: 135 auf Bildschirm 13.5
!!! Vorsicht:
xf_set_scale_auto() macht intern aus der INT-Zahl eine Float-Zahl, das Ausgabeformat muss dazu passen. Deshalb hier: "%.1f"

```
int xf_set_color ( xf_inout_type *stru,int color_set,
                   FL_COLOR col1, FL_COLOR col2, FL_COLOR lcol)
```

verändert die vordefinierten Farben. Die Farben col1,col2,lcol entsprechen dabei den gleichnamigen Farben in Xforms (siehe fl_set_object_color () und fl_set_object_lcol ())

Für verschiedene Zustände sind Farben vordefiniert.
(Projektweite Einstellung: ‚xf_visu/xf_appl_config.h‘).

mit
int xf_set_color ( )
kann diese Voreinstellung xf_object - spezifisch überschrieben werden.

Die Zustände (Parameter ‚color_set‘) sind:
- XF_COLS_NOCON, keine Averbindung zur SPS
- XF_COLS_ERRINP fehlerhafte Eingabe, nicht in SPS geschrieben
- XF_COLS_DIS; Bedienung gesperrt
- XF_COLS_DEF, Vorgabe: das sind die mit xforms definierten Farben
- XF_COLS_RANGE, Ausser Bereich
- XF_COLS_FALSE, Boolsche Ein-/Ausgaben : FALSE-Zustand
- XF_COLS_TRUE Boolsche Ein-/Ausgaben : TRUE-Zustand

Farben:
- col1 Wenn nicht im Fokus
- col2 Wenn im Fokus
- lcol Schriftfarbe

Die für einen Zustand aktuell eingestellten Farben können mit

```
int xf_get_color ( xf_inout_type *stru, int color_set,
    FL_COLOR *col1, FL_COLOR *col2, FL_COLOR *lcol)
```

gelesen werden.

xf_enable_operation ()

Bearbeitung freigeben oder sperren

Normalerweise sind alle inout- xf_objects zur Bearbeitung freigegeben. Um unterschiedlich priorisierte Bediener-Levels zu realisieren gibt es die function
int xf_enable_operation (FL_OBJECT *obj, int yes) ;

Die Bedienbarkeit soll dann allerdings auch angezeigt werden können. xf_visu signalisiert dies durch Änderung der labelcolor, also der Farbe des Textes.
Die Farbe ist in xf_appl_defaults.h vordefiniert,
die Änderung erfolgt mit dem Wechsel, den xf_enable_operation ( ) erzeugt.

xf_add_inout_auto ( ) setzt die Länge der FL_OBJECTs zur Eingabe automatisch, so dann die für den S7_TYPE_* definierte maximale Zahl

noch eingegeben werden kann.
Dies kann mit

int xf_set_maxchars (xf_inout_type *stru, int maxchars)

verändert werden.
Besonders bei der Eingabe eines Strings ist das interessant.

Ein-/Ausgabe bei Boolschen Werten

xf_add_inout_auto_bool ()

Abgrenzung:
Dies betrifft eigenständige Bool‘sche Werte, denen exklusiv ein FL_OBJECT zugeordnet ist.

Das übliche FL_OBJECT ist ein ‚button“.

int xf_add_inout_auto_bool (xf_inout_type *stru, int bit_nr, int type_nr ,
void (*disp)() , void (*write)())

im vorhergehenden **xf_inout_create ()** wurde zwar die Byte-Adresse definiert, die Bit-Adresse fehlt noch. Das macht int bit_nr.
Als int **type_nr** sind zulässig:

- BIT1 : setzt das Bit in der SPS auf 1
- BIT0 : setzt das Bit in der SPS auf 0
- BITT : togglet das Bit in der SPS
- BITP : setzt das Bit auf 1 bei Tastendruck und rücksetzt beim Loslassen

Bei der Erstellung der Buttons mit **fdesign** muss die Zuordnung berücksichtigt werden:

- BIT1 : NORMAL_BUTTON oder INOUT_BUTTON
- BIT0 : NORMAL_BUTTON oder INOUT_BUTTON
- BITT : NORMAL_BUTTON
- BITP : INOUT_BUTTON

Für Lesen und Schreiben sind Funktionen sind vordefiniert. Wenn diese benutzt werden sollen, wird der jeweilige Funktionspointer void (*disp) () oder void (*write)() mit NULL besetzt. Benutzerdefinierte Funktionen können hier eingehängt werden.

Ausgabe:
Normalerweise ist der Farbumschlag nicht relevant und wird nicht ausgeführt.
Bei einem Toggle-Bit (BITT) ist es sinnvoll, den aktuellen Zustand anzuzeigen. Bei den anderen Typen ist der Farbumschlag ausgeschaltet.
Mit der function „**xf_set_color_active ()**“ kann die Farbanzeige auch für xf_Objects mit anderer **type_nr a**ktiviert oder deaktiviert werden.

int xf_set_color_active (FL_OBJECT *obj, int yes)

mit yes >< 0 wird die Farbanzeige aktiviert, mit 0 deaktiviert.
Als returnwert gibt die function den vorherigen Zustand zurück.
Da nur bei Toggle-Bits eine Farbe für TRUE definiert ist, muss dann aber zusätzlich mit **xf_set_color ()** die anzuzeigende Farbe programmiert werden.

Vordefiniert sind die Farben, die bei der FL_OBJECT-Erstellung in **fdesign** programmiert worden.

Bei Typ BITT (toggle) ist aber eine Anzeige sinnvoll.
Daher gelten für TRUE :
  fg_true = FL_GREEN;
  bg_true = FL_GREEN;

xf_set_color ()

verändert die vordefinierten Farben. Die Farben col1,col2,lcol entsprechen dabei den gleichnamigen Farben in Xforms (siehe fl_set_object_color () und fl_set_object_lcol ())

Für verschiedene Zustände sind Farben vordefiniert.
(Projektweite Einstellung: ‚xf_visu/xf_appl_config.h‘).

mit
```
int xf_set_color ( xf_inout_type *stru,int color_set,
                   FL_COLOR col1, FL_COLOR col2, FL_COLOR lcol)
```

kann diese Voreinstellung xf_object - bezogen überschrieben werden.

Die Zustände (Parameter ‚color_set‘) sind:
   XF_COLS_NOCON,   keine Averbindung zur SPS
   XF_COLS_ERRINP   fehlerhafte Eingabe, nicht in SPS geschrieben
   XF_COLS_DIS;   Bedienung gesperrt
   XF_COLS_DEF,   Vorgabe: das sind die mit xforms definierten Farben
   XF_COLS_RANGE,   Ausser Bereich
   XF_COLS_FALSE,   Boolsche Ein-/Ausgaben : FALSE-Zustand
   XF_COLS_TRUE   Boolsche Ein-/Ausgaben : TRUE-Zustand

 Farben:
 col1   Wenn nicht im Fokus
 col2   Wenn im Fokus
 lcol   Schriftfarbe

Die für einen Zustand aktuell eingestellten Farben können mit

```
int xf_get_color ( xf_inout_type *stru,int color_set,
                   FL_COLOR col1, FL_COLOR col2, FL_COLOR lcol)
```

gelesen werden.

Wenn ein Farbumschlag als Anzeige nicht reicht, muss wie bei **xf_add_show_fun ()** eine function für die Anzeige geschrieben werden.

Eingabe:
write = NULL
Wenn xf_Objects mit **xf_add_inout_auto_bool ()** bearbeitet werden, benötigen sie normalerwweise keine vom Anwender programmierte callback-function. Sie wird automatisch versorgt und benutzt eine in „xf_visu“ definierte Standard-callback function.

write != NULL
Wenn eine Anwender-definierte Eingabe-Funktion übergeben wird, muss auch die callback-routine übergeben werden.

Diese kann direkt in **fdesign**
mit fl_set_object_callback(obj, callback , ..)
oder mit xf_mod_callback (FL_OBJECT *obj, void (*callback)())
übergeben werden.

Funktionserweiterungen:

xf_enable_operation ()

Bearbeitung freigeben oder sperren

Normalerweise sind alle inout- xf_objects zur Bearbeitung freigegeben.
Um unterschiedlich priorisierte Bediener-Levels zu realisieren gibt es die function
int xf_enable_operation (FL_OBJECT *obj, int yes) ;

Die Bedienbarkeit soll dann allerdings auch angezeigt werden können.
xf_visu signalisiert dies durch Änderung der labelcolor, also der Farbe des Textes.
Die Farbe ist in xf_appl_defaults.h vordefiniert,
die Änderung erfolgt mit dem Wechsel, den xf_enable_operation ( ) erzeugt.

Eingabe bei Boolschen Werten

xf_add_in_auto_bool ( )

Abgrenzung:
Dies betrifft eigenständige Bool‘sche Werte, denen exklusiv ein FL_OBJECT zugeordnet ist.

Das übliche FL_OBJECT ist ein ‚button“.

```
int xf_add_in_auto_bool ( FL_OBJECT *obj, int bit_nr, int type_nr ,
                      void (*write)() )
```

im vorhergehenden **xf_inout_create ()** wurde zwar die Byte-Adresse definiert, die Bit-Adresse fehlt noch. Das macht int bit_nr.
Als int **type_nr** sind zulässig:

- BIT1 : setzt das Bit in der SPS auf 1
- BIT0 : setzt das Bit in der SPS auf 0
- BITT : togglet das Bit in der SPS
- BITP : setzt das Bit auf 1 bei Tastendruck und rücksetzt beim Loslassen

Bei der Erstellung der Buttons mit **fdesign** muss die Zurodnung berücksichtigt werden:

- BIT1 : NORMAL_BUTTON oder INOUT_BUTTON
- BIT0 : NORMAL_BUTTON oder INOUT_BUTTON
- BITT : NORMAL_BUTTON
- BITP : INOUT_BUTTON

Ausgabe: keine.
Es werden über den oben beschriebenen **plc_block[n].formlist_ptr[] -** Mechanismus keine Daten bestellt.

Eingabe:
write = NULL
Wenn xf_Objects mit **xf_add_inout_auto_bool ()** bearbeitet werden, benötigen sie keine vom Anwender programmierte callback-function.
Sie wird automatisch versorgt und benutzt eine in „xf_visu“ definierte Standard-callback function.

write != NULL
Wenn eine Anwender-definierte Eingabe-Funktion übergeben wird, muss auch die callback-routine übergeben werden.
Diese kann direkt in **fdesign**
oder mit xf_mod_callback (FL_OBJECT *obj, void (*callback)())
oder mit fl_set_object_callback(obj, callback , ..) übergeben werden.

Funktionserweiterungen:

xf_set_color ()

verändert die vordefinierten Farben. Die Farben col1,col2,lcol entsprechen dabei den gleichnamigen Farben in Xforms (siehe

fl_set_object_color () und fl_set_object_lcol ())

Für verschiedene Zustände sind Farben vordefiniert.
(Projektweite Einstellung: ‚xf_visu/xf_appl_config.h').

mit
```
int xf_set_color ( xf_inout_type *stru,int color_set,
                   FL_COLOR col1, FL_COLOR col2, FL_COLOR lcol)
```

kann diese Voreinstellung xf_object - bezogen überschrieben werden.

Die Zustände (Parameter ‚color_set') sind:

XF_COLS_NOCON,	keine Averbindung zur SPS
XF_COLS_ERRINP	fehlerhafte Eingabe, nicht in SPS geschrieben
XF_COLS_DIS;	Bedienung gesperrt
XF_COLS_DEF,	Vorgabe: das sind die mit xforms definierten Farben
XF_COLS_RANGE,	Ausser Bereich : bei Zahlenwerten
XF_COLS_FALSE,	Boolsche Ein-/Ausgaben : FALSE-Zustand
XF_COLS_TRUE	Boolsche Ein-/Ausgaben : TRUE-Zustand
XF_COLS_HI_LIMIT	Wenn der Wert den optionalen Vergleichswert überschreitet
XF_COLS_Lo_LIMIT	Wenn der Wert den optionalen Vergleichswert unterschreitet
XF_COLS_USER_1	kann durch ein a xf_inout_attrib() Kommando aktiviert werden.
XF_COLS_USER_2	kann durch ein a xf_inout_attrib() Kommando aktiviert werden.

Farben:

col1	Wenn nicht im Fokus
col2	Wenn im Fokus
lcol	Schriftfarbe

Die für einen Zustand aktuell eingestellten Farben können mit

```
int xf_get_color ( xf_inout_type *stru,int color_set,
                   FL_COLOR col1, FL_COLOR col2, FL_COLOR lcol)
```

gelesen werden.

xf_enable_operation ()

Bearbeitung freigeben oder sperren

Normalerweise sind alle inout- xf_objects zur Bearbeitung freigegeben.
Um unterschiedlich priorisierte Bediener-Levels zu realisieren gibt es die function
int xf_enable_operation (FL_OBJECT *obj, int yes) ;
yes != 0 : Bedienen freigegeben, yes = 0 : Bedienung gesperrt.

Die Bedienbarkeit soll dann allerdings auch angezeigt werden können.
xf_visu signalisiert dies durch Änderung der labelcolor, also der Farbe des Textes.

Die Farbe ist in xf_appl_defaults.h vordefiniert,
die Änderung erfolgt mit dem Wechsel den xf_enable_operation ( )
erzeugt.

Um die automatische Farbumschaltung (z.B. bei Toggle-Bits) aktivieren
und deaktivieren zu können, gibt es die function

xf_set_color_active (xf_inout_type *stru, int yes)
muss nur einmal aufgerufen werden, mehrmals ist auch erlaubt.

Datenübergabe an die SPS: Prinzip-Aufbau

Interne Bearbeitung:

xf_add_inout_auto() versorgt das Ein-Ausgabe xf_Object intern mit den callbacks, die zu den Eingabe-Routinen führen.
Das Schreiben zur PLC wird über den unten beschriebenene Mechanismus ausgeführt.

Benutzung bei Anwenderprogrammierung:

Für den Anwender stehen zusätzlich verschiedene functions zur Verfügung:
Wenn die Schreib-Operation in Zusammenhang mit einem „xf_Object“ steht: (Die Adresse in der PLC also bereit bekannt ist :)

```
int xf_write_to_plc ( )

int xf_write_to_plc ( FL_OBJECT *obj, int typ, uint64_t val,double rval);
```

Damit werden S7-Basistypen (s.o. Tabelle 1) zur Simatic geschrieben.
Prinzipiell: Ein Float-Wert wird über **double rval** übergeben, die anderen Basistypen über **uint64_t val**.

S7-Strings benötigen eine zusätzliche function:

```
int xf_writeStr_to_plc ( )

int xf_writeStr_to_plc ( FL_OBJECT *obj, char* s7str);
```

Der dabei übergebene String muss als kompletter S7String aufgebaut sein, also
1. Byte = uint8 size (Maximale Länge des Strings in der PLC)
2. Byte = uint8 length (aktuelle Länge des Strings in der PLC)
3. und folgende bytes : der String-Inhalt.

Für das freie **Schreiben ohne xf_Object -Definition**:

```
void xf_write_to_plc_addr ( )

void xf_write_to_plc_addr ( int typ, int cpu,
                     int db,int dbb, int bit,
                     uint64_t val,double rval)
```

!!! Diese function sollte vorsichtig benutzt werden, eine Fehlersuche Auf Basis von Konstanten kann sehr aufwendig werden.

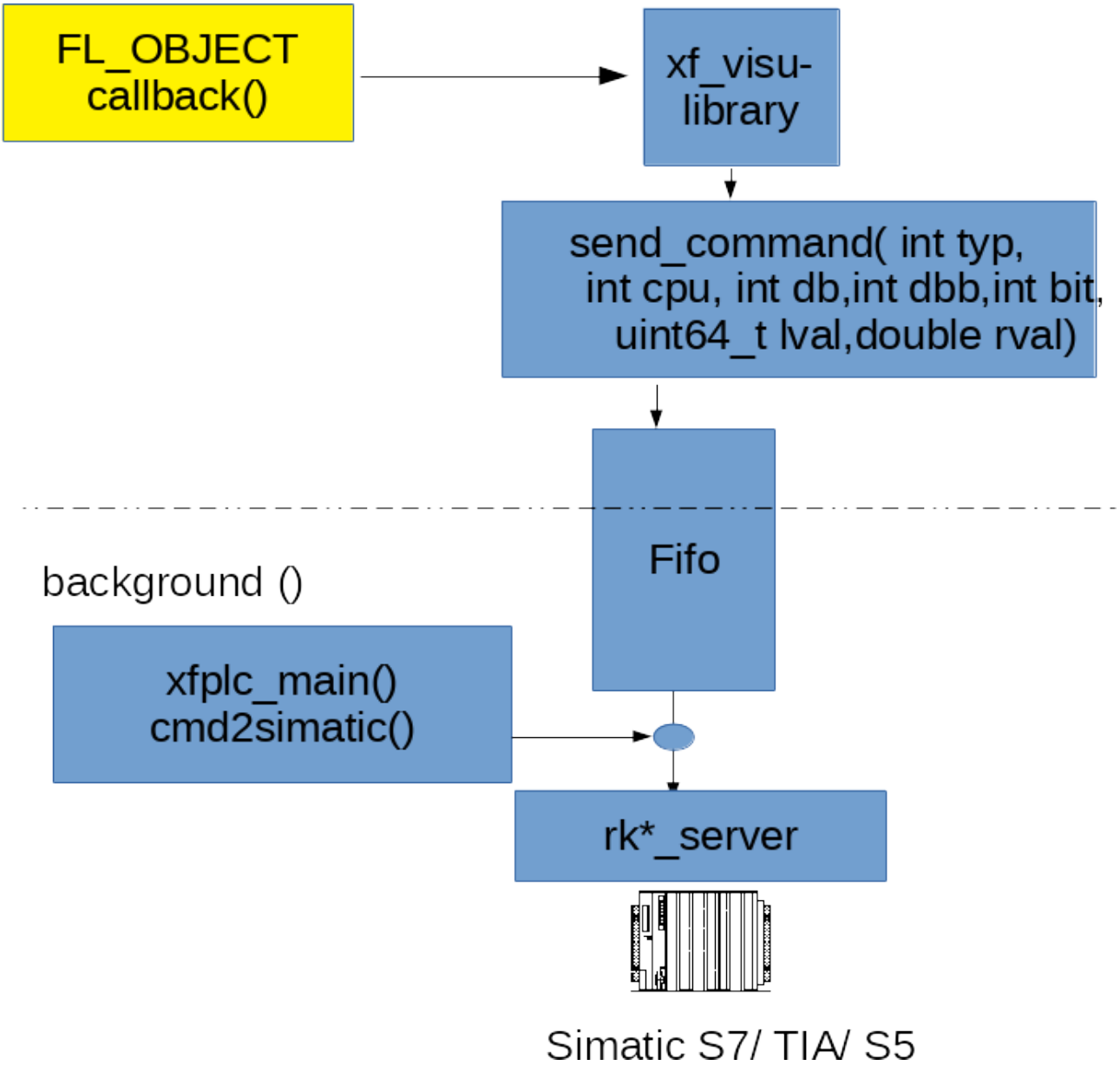
```
void xf_writeStr_to_plc_addr ( )

void xf_writeStr_to_plc_addr ( int cpu , int db,int dbb, char* s7str);
```

ist zum Schreiben von Strings zuständig.

„xf_visu“ schreibt alle Werte als Kommandos in einen Fifo, der in der Hintergrund-Bearbeitung durch **xfplx_main()** zu der
Simatic geschrieben wird.

Die function **send_command()** wird intern von den oben beschriebenen functions benutzt. **send_s7string()** ist intern für Strings zuständig.



Der Fifo hat derzeit Platz für 100 Kommandos.
Alle Operationen, die zur SPS schreiben, werden über diesen Mechanismus ausgeführt.
Ein Kommando kann bis zu 12 Bytes übergeben, dies entspricht der maximalen Länge eines PLC-Basis-Typs.
Strings werden ebenfalls durch den Fifo transportiert.

Der Parameter „typ“ wird, abhängig vom zu schreibenden Wert, mit einer der folgendene Konstanten versorgt, die Parameterübergabe erfolgt dann

über den markierten Eingangsparameter lval oder rval

Warum über Fifo und nicht direkt ?
Das ermöglicht der Applikation, mehrere Kommandos / Variablen direkt zusammen abzusetzen ohne zu prüfen ob der „Schreib-Kanal“ bereits frei ist. Der Programmierer behält außerdem die Kontrolle über die Reihenfolge.

PLC-Datentyp	Konstante	autom.Umrechnung	lval	rval
-- none --	S7TYPE_NIL=0,			
BOOL	S7TYPE_BIT,			
BYTE	S7TYPE_BYTE,	ja	ja	
CHAR	S7TYPE_CHAR,	ja	ja	
WORD	S7TYPE_WORD,	ja	ja	
INT	S7TYPE_INT,	ja	ja	
DWORD	S7TYPE_DWORD,	ja	ja	
DINT	S7TYPE_DINT,	ja	ja	
REAL	S7TYPE_REAL,	ja		ja
DATE	S7TYPE_DATE,	ja	ja	
TIME_OF_DAY	S7TYPE_TOD,	ja	ja	
TIME	S7TYPE_TIME,	ja	ja	
S5TIME	S7TYPE_S5TIME,	ja	ja	
DT	S7TYPE_DT,	ja		
STRING	S7TYPE_STRING,	ja		
COUNTER	S7TYPE_COUNTER,	ja	ja	
DTL	S7TYPE_DTL,	(*)		
LDT	S7TYPE_LDT,	(*)		
LTIME	S7TYPE_LTIME,	(*)		
SINT	S7TYPE_SINT,	ja	ja	
USINT	S7TYPE_USINT,	ja	ja	
UINT	S7TYPE_UINT,	ja	ja	
UDINT	S7TYPE_UDINT,	ja	ja	
LWORD	S7TYPE_LWORD,	ja	ja	
ULINT	S7TYPE_ULINT,	ja	ja	
LINT	S7TYPE_LINT,	ja	ja	
LREAL	S7TYPE_LREAL	ja		ja
Zusatzfunktionen Blockmove, l**val** als Pointer				
Memcpy (2 bytes)	RAW16		ja	
Memcpy (4 bytes)	RAW32		ja	
Memcpy (8 bytes)	RAW64		ja	
Memcpy (12 bytes)	RAW96		ja	
Kommandos für Bits				
Setze	BIT1			
Rücksetze	BIT0			
Toggle	BITT			

(*) **xf_add_inout_auto () kann diese Formate verschicken: sie werden vorverarbeitet und über die RAW-Befehle verschickt.**

Tabelle 2: send_command() Typen und Übergabeparameter

send_command () schreibt in den Fifo.

Auslesen des Kommando-Fifos und Schreiben über rk*_server.

Zur Datenübergabe in die Simatic wird bei Start eine Task des rk*_servers alloziert. Diese dient nur dem Schreiben in die Simatic(en).

Da die rk*_server formatlose Datenblöcke übergeben und die Kommunikation zur Simatic asynchron zu deren Zyklus abläuft, muss einer Fragmentierung vorgebeugt werden da sonst Daten verfälscht übergeben werden könnten.

In der Simatic

werden deshalb ein DB und ein FC eingebaut. Diese beiden Bausteine sind Bestandteil von xf_visu und werden für Classic-CPUS und TIA-CPUs mitgeliefert.

In den DB wird geschrieben:
Operation, Ziel-DB-Nr, Ziel-DBB, Bit, Wert, Echo_der_Operation

Der im OB1 eingehängte FC vergleicht
Operation == Echo_der_Operation
sind beide gleich, sind alle Daten komplett angekommen.
Er führt die Operation aus und setzt im DB die Werte Operation und Echo_der_Operation auf 0, d.h. keine Operation.

Als Operation wird eine Kennung für die Anzahl der in der CPU zu platzierenden Bytes übergeben, für Bit-Operationen gibt es die Kennungen „Setze Bit“, „Rücksetze Bit“ , „toggle Bit“.

cb_touch_inp_start () : die virtuelle Zahlentastatur

Die virtuelle Zahlentastatur wird bei mit xf_add_inout () definierten xf_Objekten und Nutzung eines Touch-Screens automatisch als callback-function eingetragen.

Sie kann aber auch vom Anwender benutzt werden.
void cb_touch_inp_start (FL_OBJECT * obj, long data)
startet die virtuelle Zahlen-Tastatur. Diese wird von „xf_visu“ bereitgestellt. Data wird nicht ausgewertet.
Die function wird von Anwender als callback aufgerufen.
Aus dem Formatstring in der „xf_visu“-structure wird ermittelt, welche Tasten freigeschaltet werden sollen.

Die function **cb_touch_inp_start()** speichert einen Pointer auf das FL_OBJECT, von der sie gestartet wurde. Dadurch ist die Rückgabe des

Editierergebnisses zum xf_Object und die Weiterverarbeitung möglich.

TODO

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Fragmente, später vielleicht noch gebraucht.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

also:
FL_OBJECT * obj;
obj = xf_inout_create (<form> -> <object>,
<nummer_des_plc_blocks> ,<byte_offs>);

Beispiel:
```
obj = xf_inout_create ( fd_f->proz_nr,  OP_OUT,224 );
```

Für das Object „fd_f→proz_nr“ (dass sich also in der Form „fd_f“ befindet) wird eine structure vom Typ „xf_inout_type“ angelegt.
Sie wird in dem Object „fd_f→proz_nr“ in dem Pointer ->„u_vdata“ eingehängt.
Es wird registriert, von woher das Objekt mit Daten versorgt wird: „plc_block“ [OP_OUT], das ist z.B. DB 201, darin ab Byte 224.
Die Byte-Adresse wird nicht ab „plc_block“-Anfang, sondern immer ab dem DBB 0 des Dbs, hier also ab DB201.DBB 0 gerechnet.

Wie wird ein Objekt bearbeitet ?

(Schrittweise Herleitung der xf_visu -Funktionen, die die Objekte bearbeiten.)

Generell ist es möglich, einen „plc_block“ zyklisch zu lesen und in der Hauptschleife , also der per idle_callback () aufgerufenen Hintergrundbearbeitung (hier : background())
eine function aufzurufen, die dann das FL_OBJECT aktualisiert.

Dabei ist zu beachten: der „plc_block“-Datenbereich besteht lediglich aus gelesenen Bytes, eine Interpretation als S7-Datentypen muss dann noch
durchgeführt werden.
(Dazu gibt es unser Paket „convert_lib“)

(1) also: z.B. Integerzahl ausgeben

```
   i = get_S7int( & ( plc_blocks[block_nr].byte[2] ) );
   sprintf(hstr,“%i“,i);
   fl_set_object_label ( <object>, hstr);
```

Nachteile:
- wird immer ausgeführt, auch wenn keine neuen Daten vorhanden sind bzw.
  diese sich nicht geändert haben.
- Was passiert, wenn sich zwar nichts geändert hat, aber eine Form das erste mal gestartet wird, also den aktuellen Stand braucht ?
- Viel Schreibarbeit

(2) eine etwas vereinfachte Möglichkeit wäre, dies in eine function zu kapseln, z.B.

```
/* -------------------------- */
/* einzelansteuerung : BEISPIEL
 * wird bei jedem background() - Zyklus bearbeitet aber nur angezeigt, wenn eine
Änderung
 * erfolgt
 */

void zeige_int_einzel( FL_OBJECT *ob, unsigned char *wert)

{
  char hstr[100];
  int i = get_s7INT(wert);

  if ( ob->u_vdata) {

  xf_inout_type *dp = ob->u_vdata;

     if ( dp->init == 0) {
          if ( i == dp->old.i)
              return;
      }
      dp->init = 0;
      dp->old.i = i;
  }
  sprintf(hstr,"%i",i);
  fl_set_object_label (ob, hstr);

}
Aufruf:
zeige_int_einzel (fd_f->int_wert, plc_blocks[OP_OUT].block->byte[2]);
```

Das Problem mit der Erstaktualisierung einer neu geöffneten Form wird

in dem obigen Beispiel mit , dp→init‘ gelöst: Die Variable wird durch **xf_inout_create ( )** auf 1 gesetzt, so das ein Erstaufruf immer zum Aktualisieren des Objects führt.

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Aber: in dem Speicherbereich, der **struct xf_inout_type,** der mit **xf_inout_create()** an dem FL_OBJECT *obj angelegt wurde, ist der Verweis auf den „io_bock“- Speicherbereich bereits hinterlegt.
Jeder „plc_block“ besitzt eine Variable **„rdy_to_draw“,** die nach dem erfolgten Lesen für einen „background()“- Zyklus == 1 ist.

Bleibt noch die individuelle Interpretation der Daten:
Integer, float, bitmuster oder Nummer für Grafik ?
Die function
int xf_add_show_fun (FL_OBJECT *obj, char *fmt, void (*disp)());

erledigt bereits alles außer der Dateninterpretation.
- obj verweist auf das Objekt und somit auch auf die eingehängte Struktur **struct xf_inout_type.**
  Darin ist die Datenherkunft hinterlegt und somit auch die Möglichkeit, **„rdy_to_draw“** auszuwerten.
  Es ist dort auch Form bekannt, in der das Objekt sitzt.
- Optional kann ein String, in der Regel ein Formatstring übergeben werden. **char *fmt**
- **Mit void (*disp)() wird der Pointer auf eine function übergeben. Diese ist vom Typ**

Prinzipiell gibt es 3 Typen von „xf_visu“-basierten Objekten:
- **Ausgabe-Objekte**
- **Ein/Ausgabe-Objekte**
- **Eingabe-Objekte**

Der Typ eines Objekts wird durch die function definiert, die der function **FL_OBJECT *xf_inout_create ()** folgt.
Die sind: (z.B)
- xf_add_show_auto (obj,S7TYPE_COUNTER,"%i");
  ‚**show**‘ besagt: Ausgabe-Objekt, muss aktualisiert werden.

- xf_add_inout_auto (obj,S7TYPE_COUNTER,"%i");
  ‚**inout**‘ besagt: ist Ein-/Ausgabe-Objekt, muss aktualisiert werden.

Die functions, die für Ausgabe- oder Ein-/Ausgabe -Objekte benutzt werden, sorgen noch für das automatische Update:

z.B:
```
    obj = xf_inout_create ( fd_typTest->i_s5counter,  TYPETST,60 );
xf_add_inout_auto (obj,S7TYPE_COUNTER,"%i" );
```

In „plc_block“ [**TYPETST**] wird in der Liste **oblist_ptr** der Verweis auf

das Object „fd_typTest->i_s5counter“ eingehängt.
Außerdem wird in der Liste **formlist_ptr** der Verweis auf die Form „fd_typTest“ eingehängt.

Dies wird von der „xf_visu“ Hauptroutine **xfplx_main()** benötigt:
Wenn kein Kommunikationsvorgang zur SPS läuft, werden sequenziell die „plc_blocks“ daraufhin untersucht, ob eine in deren Liste **formlist_ptr** eingetragene Form aktuell angezeigt wird.
Ist dies der Fall, dann wird der „plc_block“ zu Lesen aktiviert.
Wenn die Daten für den „plc_block“ eingetroffen sind, wird dessen Liste **oblist_ptr** abgearbeitet und alle Objekte in aktuell sichtbaren Forms werden aktualisiert.

Reine Eingabe-Objekte sind, von Buttons abgesehen, aktuell (noch ?) keine vorgesehen.
Sie benötigen jedenfalls keine Aktualisierung, sie werden also in die Listen **oblist_ptr** und **formlist_ptr** nicht eingetragen.

Bem: bei dem obigen Beispiel sieht man auch, weshalb die function **xf_inout_create ()** das übergebene Objekt zurückgibt:
Einfach nur, um bei den nachfolgen aufgerufenen functions Schreibarbeit zu sparen.

Variablen im OP ohne Verbindung zur SPS

Variable, die nur in der Visu benutzt werden und keine Verbindung zur plc haben, werden genau wie die plc-Variablen definiert,
es folgt aber hinter der Definition der Basisstruktur als letzte function

xf_set_intern (xf_inout_type *stru, void *addr)

wobei *addr auf die Adresse im Rechner verweist.

Mit **xf_set_intern ()**

wird die zuvor definierte, sps-orientierte Adresse der Variablen umgebogen und die functions für *disp() und *write() werden geändert.
Es werden die zuvor definierten Datentypen (z.B. s7type_INT)
in eine rechner-orientierte Form umdefiniert.

Interne Variable werden nicht getriggert,da die automatische Triggerung durch das Eintreffen des bestellten Datenbausteins erfolgt.
Sie müssen bei Bedarf aktualisiert werden, z.B. mit einem

xf_upd_obj(FL_OBJECT *obj) - Aufruf, der in der Background()-function eingehängt ist.

PLC-Datentyp	Konstante	autom.Umrechnung	lval	rval
-- none --	S7TYPE_NIL=0,			
BOOL	S7TYPE_BIT,			
BYTE	S7TYPE_BYTE,	ja	ja	
CHAR	S7TYPE_CHAR,	ja	ja	
WORD	S7TYPE_WORD,	ja	ja	
INT	S7TYPE_INT,	ja	ja	
DWORD	S7TYPE_DWORD,	ja	ja	
DINT	S7TYPE_DINT,	ja	ja	
REAL	S7TYPE_REAL,	ja		ja
DATE	S7TYPE_DATE,	ja	ja	
TIME_OF_DAY	S7TYPE_TOD,	ja	ja	
TIME	S7TYPE_TIME,	ja	ja	
S5TIME	S7TYPE_S5TIME,	ja	ja	
DT	S7TYPE_DT,	ja		
STRING	S7TYPE_STRING,	ja		
COUNTER	S7TYPE_COUNTER,	ja	ja	
DTL	S7TYPE_DTL,	(*)		
LDT	S7TYPE_LDT,	(*)		
LTIME	S7TYPE_LTIME,	(*)		
SINT	S7TYPE_SINT,	ja	ja	
USINT	S7TYPE_USINT,	ja	ja	
UINT	S7TYPE_UINT,	ja	ja	
UDINT	S7TYPE_UDINT,	ja	ja	
LWORD	S7TYPE_LWORD,	ja	ja	
ULINT	S7TYPE_ULINT,	ja	ja	
LINT	S7TYPE_LINT,	ja	ja	
LREAL	S7TYPE_LREAL	ja		ja
Zusatzfunktionen Blockmove, l**val** als Pointer				
Memcpy (2 bytes)	RAW16		ja	
Memcpy (4 bytes)	RAW32		ja	
Memcpy (8 bytes)	RAW64		ja	
Memcpy (12 bytes)	RAW96		ja	
Kommandos für Bits				
Setze	BIT1			
Rücksetze	BIT0			
Toggle	BITT			

TODO: Liste mit Rechnerninternen Repräsentation der Datentypen

Interne Variable werden nicht getriggert, sie müssen bei Bedarf aktualisiert werden, z.B. mit
xf_upd_obj(FL_OBJECT *obj)