C-Funktionen und C^{++}-Klassen in Neo/Nst importieren

Kersting Bunte

Prog-Unit-Wrapper

- einfachste Weg um die Funktionalität von NST zu erweitern
- Funktinalität über die Prog-Unit verfügbar
- Unterscheidung zwischen C-Funktionen und C^{++}-Klassen

C-Klassen importieren

1. nach #*include*-Anweisungen folgende Zeile einfügen:
   *char *FUNCTIONS_nnn[] ={„scalar_prod(float*,float*)“, NULL};*

2. Implementation der Funktion selbst, z.B. *PUWimportCF.c*.
   *float scalar_prod(int *piDim, void **ppvArg) { ... }*

3. *gcc -shared -o PUWimportCF.so PUWimportCF.c*

C^{++}-Klassen importieren

```
char *foo_c = {
                   „c_:“,    // Präfix wegen Namenskollisionen
              „foo(int)“,    // Klassen-Konstruktor
                  „~foo“,    // Klassen-Destruktor
              „float fx“,    // eine float-Variable
                „int ix“,    // eine int-Variable
      „Readonly int iy“,
         „float *pfFloat“,   // float-Array
    „void change(int)“,      // exportierte Methode
                   NULL    };

char **CLASSES_nnn[] = {foo_c, NULL};
```

NST

- bietet eine Sammlung von mehrfachverwendbaren Softwarekomponenten, Units genannt.

- die meisten Units besitzen Inputs und Outputs, mit denen sie mit anderen Units verbunden werden können, so entstehen Circuits.

- Circuits können eingekapselt werden, und somit als neue NST-Unit verfügbar gemacht werden.

- ausführbar unter Neo

Der NST/Neo-Wrapper 1/2

- C: *mku+* generiert Interfacecode um eine Kollektion von kooperirenden C-Routinen (C-Klasse) als Unit für NST/Neo zugänglich zu machen.
- C^{++}: *mku++* tut das gleiche für eine C^{++}-Klasse.

Der NST/Neo-Wrapper 2/2

- Für kleine Klassen: NST-Unit mit ihren *exec*, *ctrl*, *load* und *save*-Funktionen implementieren.

- *adapt*-Methode erlaubt Implementation von Klassen mit Main-Adaptionen.

- Große Klassen: sollte man nach Klasseninstanzen trennen und diese als benannte Containerunits erstellen, in denen die Funktionen dann als Subunits mit eigenen *exec*-Funktionen verfügbar sind. Die *ctrl*, *load* und *save*-Funktionen gehören jeweils zu einen Klassencontainer (global).

Erstellen von NST/Units

- Konstruktorroutinen definieren Eigenschaften der Unit
- Rückgabewert ist immer vom Typ *.unitptr* $\rightarrow$ er zeigt auf Datenstruktur unittype die alle wichtigen Informationen enthält. (Inputs, Outputs, Zeiger auf Werte und private Variablen).
- Schließt 6 Funktionszeiger zu C-Subroutinen execution, adaption, loading, saving, control und removal ein.

NST Datentypen

NST_SCALAR	ein einziger float-Wert
NST_FLOAT	ein Array von float-Werten („float-Vektor“)
NST_INT	ein Array von int-Werten („int-Vektor“)
NST_BYTE	ein Array von unsigned char-Werten („byte-Vektor“)
NST_STRING	eine variable length von null-terminated character string („string Typ“)
NST_ANY	für zukünftige Erweiterungen, um willkürliche Datentypen zu handeln

Inputs und Outputs von NST-Units zugänglich machen

NST stellt Macros zur Verfügung, um auf die Werte an den Pins einer Unit zuzugreifen.
NstInp(u,f,p): Wert des p-ten Pin des f-ten Inputfeldes der Unit u.

NST Pintyp	Macroname	steht für
float vector	pfNstOut(u,f,p)	ptr zum ersten Element des float-Vektor
int vector	piNstOut(u,f,p)	ptr zum ersten Element des (short) int-Vektor
byte vector	pcNstOut(u,f,p)	ptr zum ersten Element (unsigned) char-Vektor
string	ptNstOut(u,f,p)	ptr zum Start des var len string
any	pvNstOut(u,f,p)	void ptr zum ersten Datenelement, jeglichen Typs
scalar float	fNstOut(u,f,p)	Wert des scalar-float-Pin
pin	pNstOut(u,f,p)	ptr zum gesamten NST-Pin struct

Das *unittype*-Struct

- für C-Implementation nötig um Objektstruktur nachzubilden.
- Struktur ist in nst_types.h definiert.

```
typedef struct unittype{
        field **in; /* Zeiger zum Feldzeigerarray für das Inputinterface */
        field **out; /* Zeiger zum Feldzeigerarray für das Outputinterface */
        ...
        int (*exec)(struct unittype*); /* main Methode */
        ... weitere Methoden für adaption, loading, saving, removal ...
        void *var;
        ... } unittype, *unitptr;
```

Eine Beispiel C-Klasse: vects.c

- Wir betrachten Beispielklasse vects um *float*-Vektoren mit fester Dimension zu manipulieren.

- *add()* Hinzufügen von Vektoren

- *set(), get()* Austauschen oder Bekommen der Elemente eines Vektors

- *clear()* Entfernen aller Vektoren

- *forall()* Funktion um etwas für alle Vektoren durchzufüren

Eine C-Klasse mit mku+ wrappen

Der Aufruf

mku+ vects vects add num get set clear

generiert die drei folgenden Dateien:

1. **nst_vects.c** Code-Schablone für NST-Konstruktorfunktion vects(...) und dazugehörigen *exec*-Funktionen

2. **neo_vects.c** Code-Schablone zum wrappen der NST-Konstruktorfunktion aus *nst_vects.c* für Neo. (Hauptsächlich zur Spezifikation der Dialog-Fenster der Unit)

3. **vects.mak** Makefile, das nicht mehr benötigt wird, wenn das Makefile.custom angepasst wurde.

Eine C^{++}-Klasse mit mku++ wrappen

nst_vects.C ist die C^{++}-Sourcedatei für vorhergehendes Beispiel, mithilfe von *vects.H* und *vects.C* implementiert. Sie hat folgende Form:

```
... /* #include-Direktiven, die mit Hilfe der xxx.H-Datei erzeugt wurden. */
class vects_aux : vects{ /* begleitende Klasse */
   public:
   vects_aux( ... ) : vects( ... ){ };
   /* Es folgen die Interfacefunktionen für NST, also exec-Funktionen
     für jede Methode und die globalen ctrl, load und save-Funktionen */
   int exec_add(unitptr u) {... return OK;};
-* int exec_add(unitptr u){ add(pfNstInp(u,0,0)); fNstOut(u,0,0)=iNum;
                       return OK; }; *-
        ... get ...
```

```
int ctrl(int mode, unitptr u){(void) mode; (void) u; ... return OK;};
int load(FILE *fp, unitptr u){(void) mode; (void) u; ... return OK;};
int save ...
}
/* Nun beginnt die NST-Konstruktorfunktion, wobei alle exec_METHODE(unitptr)
  und adapt_METHODE(unitptr), globale ctrl, load, save und remove-
  Funktionen bereits von mku++ generiert wurden, wir fahren also fort
  mit dem unitptr: */
extern „C“ {
  unitptr vc1( ... , unitptr uHost){
  -* unitptr vc1(int iDim, unitptr uHost) für dieses Beispiel *-
    unitptr uNew;
    if((uNew=primitive_unit(NST_HOST,uHost))==NULL) ... „Error“
        ... return NULL;}
    ...
```

```
nst_var_of(uNew) = new vects_aux( ... ); /* Instanz erstellen */
...
/* Es folgen die Subunits für jede Klassenmethode */;
{ unitptr uSub_add = primitive_unit( ... , NST_HOST, uNew);
-* unitptr uSub_add = primitive_unit(NST_INP, NST_FLOAT, iDim,
                        NST_OUT, NST_SCALAR, 1,
                        NST_HOST, uNew)
Parameter: ein Input vom Typ Floatvektor, der Dimension iDim,
        ein Output des Typs Scalar der Dimension 1 und
        schließlich der Container uNew. *-
            ...
 nst_name_unit(„add“, uSub_add );
 nst_var_of(uSub_add) = nst_var_of(uNew);
 uSub_add -> exec = exec_add;
            ...
```

```
    }
    /* Nun werden die ctrl, load und save-Methoden noch gesetzt (das
      hat mku++ schon erledigt) und anschließend die begleitende Klasse
      mit dem return-Befehl beendet */
    ...
    label_unit(uNew, „vc1“);
    return uNew;
  }
  /* Als letztes kommt das Interface zu den Elementfunktionen, dass von
    mku++ generiert wird. */
  ...
}
```

Die Form der *neo_nnn.c* Datei

... / externe Deklarationen für neue Units */*
extern *unitptr vects(* **...** *unitptr);*
extern *NstIconDef_t ICONDEF_vects;*
/ Hier wird deklariert, welche Icons der Folder (in welcher Reihenfolge) exportiert: */*
BEGINFOLDER(vects)
ICONENTRY(vects)
ENDFOLDER;
/ Nun kommt der Wrapper für die Unit vects */*
ICONFORM1(vects) / formt Dialogfenster1 Inputwindowsyntax */*
„unit vets - %n“
- weitere Angaben, z.B. „dimension: %2i %n“ *-*

```
ICONLABEL(„vects“) /* Label für die Unit */
ICONMANPAGE(„vects“) /* Name der Manpage */
ICONMAKE(vects){
   unitptr uNew;
   SRCLINE(„$ = #(AT); \n“); /* !Kriptisch! */
   /* Aufruf der Konstruktorfunktion, der Dimensionsparameter ist im
      IVAL(0). Alle Inputfelder des Inputwindow sind numeriert
      und können mit (int)IVAL(i), (float)FVAL(i) oder
      (char*)TVAL(i) abgerufen werden. */
   uNew = vcl(IVAL(0),HOST);    uNew = vc1(HOST);
```

```
/* mku++ hat hier einen langen Kommentar eingefügt, der
  nichtkommentiert einen Hilfstext in neo ausgeben kann. */
z.B. ICONFORM_FOR_METHOD(clear)
   „clear – clear all vectors %n“
  END;
   ...
} ...
```

Dateien zu diesem Vortrag

Die Beispieldateien, Folien, sowie das ausführliche Script liegen auf den AG-Rechnern unter

/homes/kbunte/project/vortrag/