RTAI-ROS

Roland Hopferwieser

22. Oktober 2015

Dieses Dokument enthält *Anhang C* der Masterarbeit *Flachheitsbasierte Folgeregelung eines vierrädrigen quasi-omnidirektionalen Roboters mit ROS-Anbindung* [Hop15] und beschreibt RTAI-ROS aus Anwendersicht.

1 Installation

1. Kopieren des Verzeichnisses `rtairos` nach `$(MATLABROOT)/rtw/c/`

2. Ausführen von `setup.m` in Matlab:

   ```
   >> matlabroot
   ans =
   /opt/matlab
   >> cd /opt/matlab/rtw/c/rtairos
   >> setup
   ```

   Das Skript kompiliert alle *S-Functions* und fügt das Unterverzeichnis `devices/` permanent dem Suchpfad[1] von Matlab hinzu. Danach sollten die Blöcke von RTAI-ROS über den *Simulink Library Browser* auswählbar sein.

3. Das Makefile-Template `rtairos.tmf` ist den Gegebenheiten des Zielsystems anzupassen. Das gilt insbesondere für die Variable

   ```
   MATLAB_ROOT = |>MATLAB_ROOT<|
   ```

   die bei der Code-Generierung durch das eigene Matlab-Verzeichnis ersetzt wird, am Zielsystem aber üblicherweise auf `/opt/matlab` zeigen soll. Gegebenenfalls sind auch die Variablen `LINUX_HOME`, `RTAIDIR` und `COMEDI_HOME` anzupassen, falls sich das Verzeichnis von `rtai-config` nicht im Linux-Suchpfad befindet, sowie `ROS_HOME`.

   Erfolgt die Code-Generierung unter Windows, sind im Abschnitt `Rules` außerdem die Zeilen

   ```
   |>START_EXPAND_RULES<|%.o : |>EXPAND_DIR_NAME<|/%.c
       gcc -c $(CFLAGS) $<

   |>END_EXPAND_RULES<|
   ```

   zu löschen, ansonsten enthalten die generierten Makefiles die Windows-Pfade zu den benutzereigenen S-Functions, die wegen der unterschiedlichen Pfadtrenner beim Kompilieren zu einem Fehler führen.

Zur Deinstallation reicht es, das Verzeichnis wieder zu löschen, da Matlab automatisch nicht-valide Verzeichnisse aus dem Suchpfad entfernt. Alternativ können Verzeichnisse auch mithilfe des Befehls `rmpath` aus dem Suchpfad entfernt werden, z. B.

```
>> rmpath /opt/matlab/rtw/c/rtairos
```

[1] anzeigbar mit dem Befehl `path`

2 Verwendung

Es soll hier davon ausgegangen werden, dass es sich bei dem System, auf dem der Quellcode generiert wird und jenem, auf dem das Programm laufen soll, um zwei separate Systeme handelt. Für die reine Code-Generierung wird keine Installation von RTAI oder ROS benötigt.

Ausgehend vom einem Simulink-Modell `MODEL.mdl` sind grundsätzlich die folgenden Schritte durchzuführen, um dieses als eigenständiges Programm auf dem Zielsystem auszuführen:

1. **RTAI-ROS als Target auswählen**. Dazu wird in der Konfiguration Simulation › Configuration Parameters › Real-Time Workshop wie in Abb. 1 dargestellt als *System target file* `rtairos.tlc` für den *Target language compiler* (TLC) ausgewählt; als *Template makefile* sollte dabei `rtairos.tmf` automatisch gewählt werden. Da der Code auf einem anderen System kompiliert werden soll, wird zusätzlich *Generate code only* selektiert.

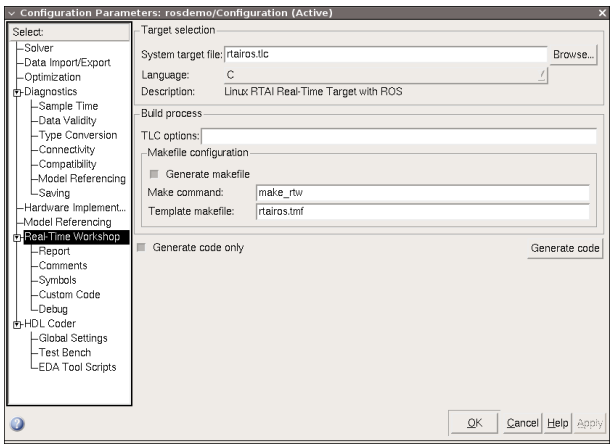

Abbildung 1: Konfiguration des Real-Time Workshops

2. **Generieren des Quellcodes**, z. B. aus dem Simulink-Model über die Tastenkombination Strg + B. Der generierte Quellcode landet dabei im Unterverzeichnis `MODEL_rtairos/` des aktuellen Arbeitsverzeichnisses. Auf Unix-Systemen wird außerdem automatisch ein symbolischer Link unter `Makefile` generiert, der auf MODEL.mk zeigt.

3. Quellcode auf das Zielsystem **kopieren**, z. B.

```
$ scp -r ../MODEL_rtairos user@target:/tmp/
```

4. Quellcode auf dem Zielsystem **kompilieren**:

```
$ cd /tmp/MODEL_rtairos
$ ln -s MODEL.mk Makefile
$ make
```

   Das kompilierte Programm wird im übergeordneten Verzeichnis (hier `/tmp`) abgelegt.

5. Starten des ROS-Masters

```
$ roscore &
```

   oder Link auf einen bereits laufenden Master-Knoten setzen:

```
$ export ROS_MASTER_URI = http://target:11311
```

6. Starten des Programms:

```
$ ../MODEL
```

Ein generiertes Programm erlaubt mehrere Argumente. Die wichtigsten in Zusammenhang mit RTAI-ROS sind:

-h Hilfe anzeigen

-v Detailiertere Ausgabe (*verbose*)

-w Auf externes Startsignal (über den Service-Aufruf `/MODEL/start`) warten

-n Name des Host Interface Tasks (Normalerweise `IFTASK`)

-N Name des ROS-Knotens (Normalerweise `/MODEL`). Die Änderung des Namens beeinflusst auch die Wurzel, unter dem die Simulink-Parameter ggf. auf dem Parameter-Server abgelegt werden, sowie die Service-Aufrufe `start`, `set_parameters` und `refresh_parameters`.

Die restlichen Schalter können der Hilfeseite entnommen werden. Zusätzlich lassen sich auch Namen innerhalb des Knotens in ROS-üblicher Weise mittels

```
$ ./MODEL name:=new_name
```

beim Start umbenennen.

Es ist möglich mehrere generierte Programme parallel auf der selben Maschine laufen zu lassen. Dazu ist ab dem zweiten Programm der Name des Echtzeit-Tasks (normalerweise `IFTASK`) beim Programmstart über den Schalter `-n` gesondert anzugeben:

```
$ ./MODEL -n TASK2
```

Anzumerken ist, dass die Gesamtzahl registrierbarer RTAI-Objekte, inklusive *Shared Memories* und Semaphoren, auf einem System bei der Installation von RTAI festgelegt wird und die Anzahl der benutzbaren Blöcke begrenzt. Die momentan festgelegte Anzahl ist in der Datei `/proc/rtai/names` unter `MAX_SLOTS` angegeben.

2.1 Anwendungsbeispiel `rosdemo.mdl`

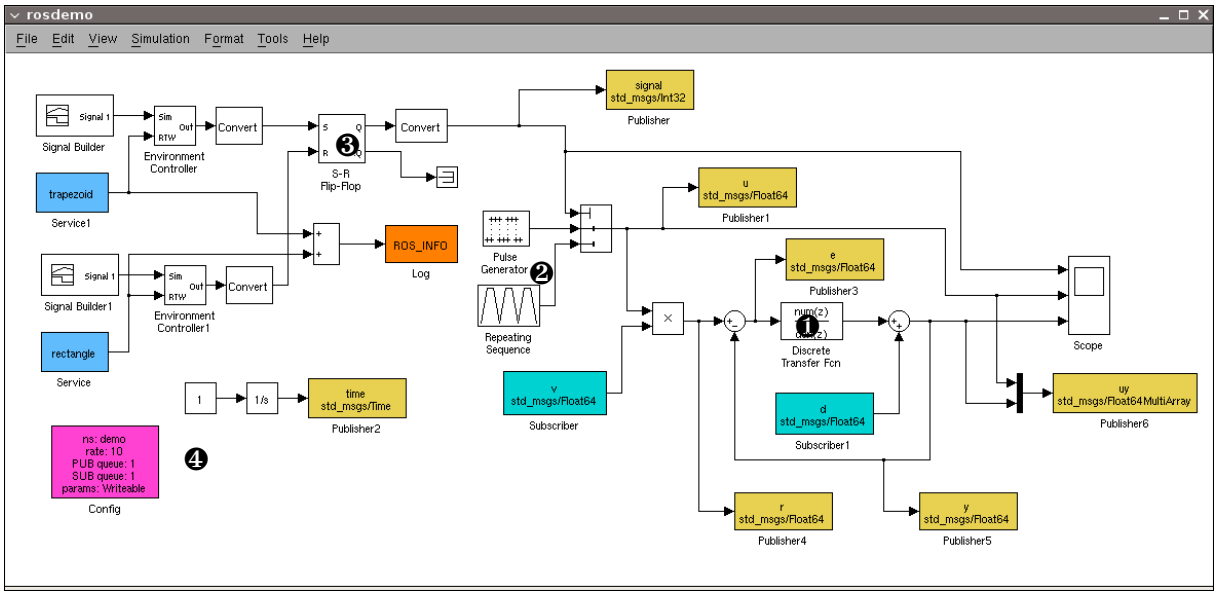

Abbildung 2: `rosdemo.mdl`

Abbildung 2 zeigt das von RTAI-Lab übernommene und um ROS-Blöcke erweiterte Demonstrationsmodell `rosdemo.mdl`, welches man im Verzeichnis `examples/` von RTAI-ROS findet. Das Modell besteht im Wesentlichen aus einem geschlossenen Kreis ❶ und zwei Signalquellen ❷, einem Rechtecksignal und einem trapezförmigen Signal, zwischen denen über die Service-Aufrufe

- `/demo/rectangle`
- `/demo/trapezoid`

unter Verwendung eines RS-Flip-Flops ❸ gewechselt werden kann. Bei jedem Service-Aufruf wird dabei eine Log-Nachricht mit der Priorität INFO generiert. Die Signale des Systems und die Simulationszeit werden über die in der Abbildung gelben Publisher-Blöcke in die Topics

- `/demo/signal` (`std_msgs/Int32`)
- `/demo/r` (`std_msgs/Float64`)
- `/demo/u` (`std_msgs/Float64`)
- `/demo/y` (`std_msgs/Float64`)
- `/demo/e` (`std_msgs/Float64`)
- `/demo/time` (`std_msgs/Time`)
- `/demo/uy` (`std_msgs/Float64MultiArray`)

geschrieben. Über die zyanen Subscriber-Blöcke lässt sich mithilfe der Topics

- `/demo/v` (`std_msgs/Float64`)
- `/demo/d` (`std_msgs/Float64`)

das Eingangssignal verstärken bzw. das Ausgangssignal um einen Offset verschieben. Der Namensraum `/demo` kann über den Konfigurationsblock ❹ geändert werden.

Nachdem die zu Beginn des Kapitels beschriebenen Schritte durchgeführt wurden, kann mithilfe der ROS-Werkzeuge auf das laufende Programm zugegriffen werden. Beispielsweise:

- Auslesen registrierter Topics und Services:

```
$ rosnode info /rosdemo
```

- Lesen des Topics `/demo/uy` (Beenden mittels Strg+C):

```
$ rostopic echo /demo/uy
```

- Einmaliges Schreiben des Topics `/demo/v`:

```
$ rostopic pub -1 /demo/v std_msgs/Float64 2
```

- Alle Parameter des Modells auslesen:

```
$ rosparam get /rosdemo
Config: {P1: 10.0, P3: 1.0, P4: 1.0, P5: 3.0}
Constant: {Value: 1.0}
Environment_Controller:
  Switch_Control: {Value: 0.0}
Environment_Controller1:
  Switch_Control: {Value: 0.0}
Integrator: {InitialCondition: 0.0}
Log: {P1: 2.0, P3: 0.0}
Publisher: {P1: 4.0, P3: -1.0}
Publisher1: {P1: 1.0, P3: -1.0}
Publisher2: {P1: 5.0, P3: -1.0}
Publisher3: {P1: 1.0, P3: -1.0}
Publisher4: {P1: 1.0, P3: -1.0}
Publisher5: {P1: 1.0, P3: -1.0}
Publisher6: {P1: 2.0, P3: -1.0}
Pulse_Generator: {Amplitude: 1.0, Period: 10000.0, PulseWidth: 5000.0}
Repeating_Sequence:
  Look_Up_Table1:
    InputValues: [0.0, 2.0, 4.0, 6.0]
    Table: [0.0, 2.0, 2.0, 0.0]
Service: {P2: -1.0}
Service1: {P2: -1.0}
Subscriber: {P1: 1.0, P3: 1.0, P4: -1.0, P5: 0.0}
Subscriber1: {P1: 1.0, P3: 0.0, P4: -1.0, P5: 0.0}
```

- Die Parameter des Rechtecksignals ändern:

```
$ rosparam set /rosdemo/Pulse_Generator '{Amplitude: 0.5, Period: 5000,
   PulseWidth: 2500}'
$ rosservice call /rosdemo/set_parameters
```

- Zum trapezförmigen Signal wechseln:

```
$ rosservice call /demo/trapezoid
```

Mit entsprechenden Plugins von *rqt* lassen sich alle Beispiele auch über eine grafische Oberfläche bewerkstelligen. Insbesondere lassen sich damit aber auch Größen zur Laufzeit zeichnen; Abb. 3 zeigt ein entsprechendes Beispiel.

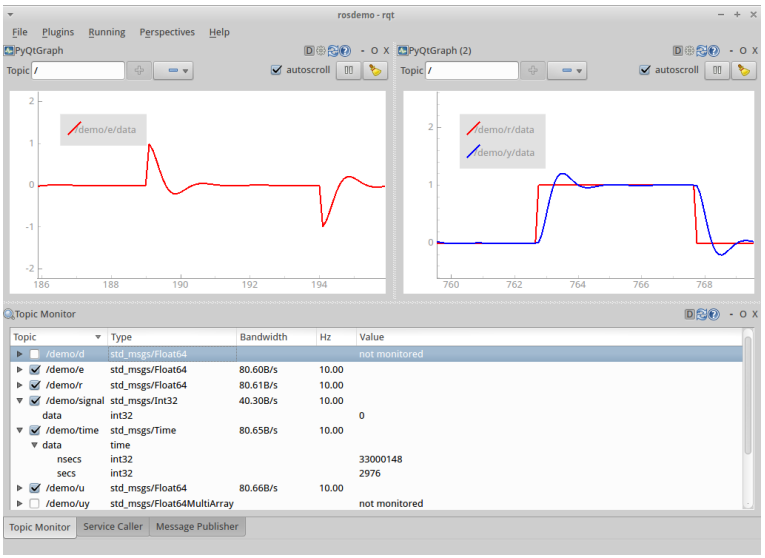

Abbildung 3: Anwendungsbeispiel von *rqt*

3 Simulink-Blöcke

In diesem Abschnitt werden die für RTAI-ROS erstellten und in Abb. 4 dargestellten Blöcke beschrieben. Die Parameter beziehen sich dabei auf die Eingabemaske. Jeder Block enthält außerdem ein Beispiel, per Kommandozeile auf die jeweiligen Daten des Blocks zuzugreifen, um den Einstieg in ROS zu erleichtern.

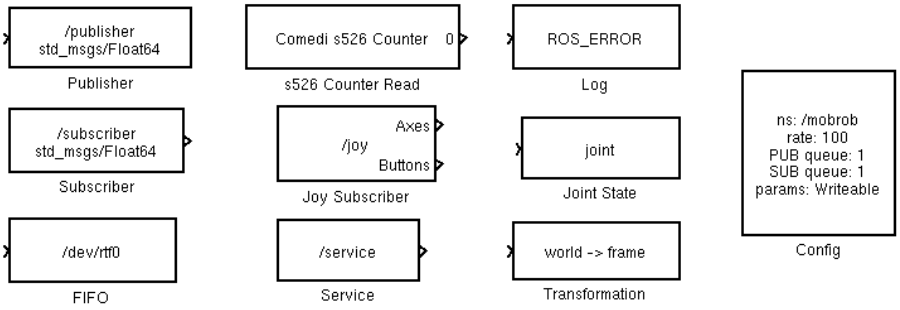

Abbildung 4: RTAI-ROS-Blöcke

3.1 Publisher

Mit dem Publisher-Block können ROS-Topics geschrieben werden. Als Eingang werden die zu sendenden Daten erwartet, die entsprechend der Definition der gewählten Nachricht serialisiert werden. Die Anzahl der Elemente des Eingangs richtet sich normalerweise nach dem gewählten Nachrichtenformat, nur im Fall von `std_msg/Float64MultiArray` wird sie dynamisch aus dem angeschlossenen Block bestimmt. Gestempelte Nachrichten erhalten automatisch einen Zeitstempel aus `ros::Time::now()`.

S-Function `sfun_ros_publisher.c`

Parameter

- *Topic* (String): Topic, in das geschrieben wird; bleibt das Feld leer, wird der Blockname verwendet.
- *Type* (Auswahl): Nachrichtenformat des Topics.
- *Sample Time* (Float): Zeitintervall zwischen zwei Abtastschritten.

Nachrichtentypen

- `std_msg/Bool`
- `std_msg/Int32`
- `std_msg/Float64`
- `std_msg/Float64MultiArray`
- `std_msg/Time`
- `geometry_msgs/Point`
- `geometry_msgs/PointStamped`
- `geometry_msgs/Twist`
- `geometry_msgs/TwistStamped`
- `geometry_msgs/Pose2D`

Über die Kommandozeile lassen sich Topics mithilfe von `rostopic` lesen:

```
$ rostopic echo -n 1 /publisher
```

3.2 Subscriber

Mit dem Subscriber-Block können ROS-Topics gelesen werden. Die empfangenen Daten werden entsprechend der Definition der gewählten Nachricht deserialisiert. Die Anzahl der Elemente des Ausgangs richtet sich normalerweise nach dem gewählten Nachrichtenformat, nur im Fall von `std_msg/Float64MultiArray` wird sie dynamisch aus dem angeschlossenen Block bestimmt.

S-Function `sfun_ros_subscriber.c`

Ausgänge

1. Deserialisierte Daten
2. Sequenznummer und Zeitstempel (nur bei gestempelten Nachrichten)

Parameter

- *Topic* (String): Topic, von dem gelesen wird; bleibt das Feld leer, wird der Blockname verwendet.
- *Type* (Auswahl): Nachrichtenformat des Topics.
- *Initial Value* (Float oder Array): Ein initialer Wert der ausgegeben wird, solange keine Nachrichten empfangen wurden. Wird ein einzelner Wert angegeben, gilt er für alle Elemente des Ausgangs.
- *Sample Time* (Float): Zeitintervall zwischen zwei Abtastschritten.

- *Reset value on enable* (Boolean): Wird dieser Wert gesetzt und der Subscriber-Block befindet sich innerhalb eines aktivierbaren Subsystems (*Enabled Subsystems*), wird beim Aktivieren des Subsystems der Subscriber auf den initialen Wert zurückgesetzt.

Nachrichtentypen siehe Publisher

Über die Kommandozeile lassen sich Topics mithilfe von `rostopic` schreiben:

```
$ rostopic pub -1 /subscriber std_msgs/Float64 3.14159
```

3.3 Service

Mit dem Service-Block lassen sich Service-Aufrufe empfangen, deren Anfrage und Antwort vom Typ `std_msg/Empty` ist. Ein empfangener Service-Aufruf generiert ein Triggersignal, indem der Ausgang für einen Zeitschritt auf Eins gesetzt wird, welches z. B. an einen *Stateflow*-Block oder ein *Triggered Subsystem* weitergereicht werden kann.

S-Function `sfun_ros_service.c`

Parameter

- *Topic* (String): Topic, in den geschrieben wird; bleibt das Feld leer, wird der Blockname verwendet.
- *Sample Time* (Float): Zeitintervall zwischen zwei Abtastschritten.

Über die Kommandozeile lassen sich Services mithilfe von `rosservice` aufrufen, z. B.

```
$ rosservice call /service
```

3.4 Transformations-Publisher

Mit dem Transformations-Publisher lassen sich Transformationsdaten veröffentlichen, wie sie für die Darstellung in *rviz*[2] zur Visualisierung von 3D-Modellen verwendet werden. Es kann zwischen einem ebenen Eingang mit drei Elementen (x, y, φ_z) und einem räumlichen Eingang mit sechs Elementen (x, y, z, φ_x, φ_y, φ_z) gewählt werden.

S-Function `sfun_ros_tf.c`

Topic `/tf` (`tf2_msgs/TFMessage`[3])

Parameter

- Eingangsformat
- Name des transformierten Objekts (`child_frame_id`)
- Referenzrahmen, auf den sich die Transformation bezieht (`header.frame_id`)

[2] http://wiki.ros.org/rviz
[3] http://wiki.ros.org/tf

Über die Kommandozeile lassen sich Transformationsdaten etwa mittels `tf_echo` ausgeben, z. B.

```
$ rosrun tf tf_echo /world /mobrob
At time 1408394837.142
- Translation: [0.781, 0.434, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.400, 0.916]
            in RPY [0.000, -0.000, 0.824]
```

3.5 Joint-State-Publisher

Mit dem Joint-State-Publisher lassen sich Gelenksdaten veröffentlichen, die vom *Robot-State-Publisher*[4] unter Zuhilfenahme eines URDF-Modells[5] in Transformationsdaten umgewandelt werden können. Der Eingang kann zwischen ein und drei Elemente umfassen, die der Reihe nach in die Felder `position`, `velocity` und `effort` geschrieben werden. Der Header wird automatisch gesetzt und erhält als Zeitstempel `ros::Time::now()`. Sämtliche Joint-State-Blöcke eines Simulink-Modells werden zu einer gemeinsamen Nachricht zusammengefasst.

S-Function `sfun_ros_joint_state.c`

Topic `/joint_states` (`sensor_msgs/JointState`)

Nachricht

```
Header header
string[] name
float64[] position
float64[] velocity
float64[] effort
```

Parameter

- *Joint Name* (String): Gelenkname, der in das entsprechende Element des Feldes `name` geschrieben wird; bleibt das Feld leer, wird der Blockname verwendet.
- *Sample Time* (Float): Zeitintervall zwischen zwei Abtastschritten.

Damit Gelenksdaten in Transformationen umgewandelt werden, benötigt der Robot-State-Publisher unter dem Parameter `robot_description` ein URDF-Modell des Roboters:

```
$ rosparam set robot_description -t model.urdf
$ rosrun robot_state_publisher robot_state_publisher &
```

3.6 Joystick-Subscribers

Dieser Block liest Topics vom Typ `sensor_msgs/Joy`, wie sie von Joysticks generiert werden.

S-Function `sfun_ros_joy.c`

[4] http://wiki.ros.org/robot_state_publisher
[5] http://wiki.ros.org/urdf

Parameter

- *Topic* (String): Topic, von dem gelesen wird; bleibt das Feld leer, wird der Blockname verwendet.
- *Number of Axes* (Integer): Anzahl der Achsen am Ausgang. Überzählige Elemente erhalten den Wert 0.
- *Number of Buttons* (Integer): Anzahl der Knöpfe am Ausgang. Überzählige Elemente erhalten den Wert 0.

Damit der Zustand eines generischen Joysticks über ROS gesendet wird, muss ein Joy-Knoten gestartet werden:

```
$ rosparam set joy_node/dev /dev/input/js0
$ rosrun joy joy_node &
```

3.7 Config

Mit diesem Block lassen sich Eigenschaften des ROS-Tasks beeinflussen. Sinnvollerweise wird der Block nur einmal innerhalb eines Simulink-Modells verwendet, mehrfache Verwendung führt jedoch zu keinem Fehler.

S-Function `sfun_ros_config.c`

Parameter

- *Rate* (Float): Frequenz mit der der ROS-Task ausgeführt wird. Beeinflusst die Frequenz, mit der Publisher senden und Lognachrichten ausgelesen werden.
- *Namespace* (String): Präfix aller Publisher, Subscriber und Services mit einem relativem *Graph Ressource Name*.
- *Publisher queue size* (Integer): Maximale Anzahl der Nachrichten die für Subscriber zwischengespeichert werden.
- *Subscriber queue size* (Integer): Anzahl der ankommenden Nachrichten die zur Verarbeitung zwischengespeichert werden.
- *Expose Parameters* (Auswahl): Gibt an, ob Block-Parameter an den Parameter-Server gesendet werden sollen und ob diese verändert werden können.

3.8 Logger

Mit diesem Block lassen sich Log-Nachrichten, ausgelöst durch eine steigenden Flanke am Eingang, an ROS schicken.

S-Function `sfun_ros_log.c`

Topic `/rosout` (`rosgraph_msgs/Log`)

Eingänge

1. Ein Trigger-Signal in Form einer steigenden Flanke, um die Nachricht zu senden.

2. Zusätzliche Daten, die an die Nachricht angehängt werden sollen. Die Anzahl der Elemente wird dynamisch bestimmt.

Parameter

- *Level* (Auswahl): Wichtigkeit der Nachricht (DEBUG, INFO, WARN, ERROR oder FATAL)
- *Message* (String): Die Nachricht, die an ROS geschickt werden soll. Die Gesamtlänge der Nachricht, inklusive Daten, ist auf 255 Zeichen beschränkt.
- *Show data port* (Boolean): Gibt an, ob der Dateneingang angezeigt werden soll.

Log-Nachrichten lassen sich zur Laufzeit z. B. mit `rqt_console`[6] lesen.

3.9 RTAI-FIFO

Mit dem FIFO-Block lassen sich Daten in RTAI-FIFOs schreiben, um Daten aus dem Kernelspace an den Userspace zu schicken, ohne die Gefahr, den Echtzeit-Task damit zu blockieren. Die 64 möglichen FIFOs besitzen dabei einen entsprechenden Eintrag `/dev/rtf0` (z. B. für FIFO 0) im Dateisystem, über den die Daten ausgelesen und weiterverarbeitet werden können; Separator zwischen den einzelnen Elementen ist ein Tabulator. Der Block ist nicht von ROS abhängig und lässt sich auch mit dem originalen RTAI-Lab verwenden.

Parameter

- *FIFO* (Integer): Ein Wert zwischen 0 und 63, der die verwendete FIFO angibt.
- *Sample Time* (Float): Zeitintervall zwischen zwei Abtastschritten.
- *Include time column* (Boolean): Gibt an, ob zusätzlich ein Zeitstempel hinzugefügt werden soll.

Die Daten aus den FIFOs lassen sich leicht mit Linux-Systemmitteln weiterverarbeiten, z. B.

- an eine Datei anhängen:
  ```
  $ cat /dev/rtf0 >> /tmp/data.txt
  ```
- Ersetzen der Tabulatoren durch Kommata:
  ```
  $ cat /dev/rtf0 | tr '\t' ','
  ```
- Reduzieren auf die Spalten 2, 3 und 5:
  ```
  $ cut /dev/rtf0 -f 2-3,5
  ```
- an STDOUT ausgeben und gleichzeitig an eine Datei anhängen:
  ```
  $ cat /dev/rtf0 | tee -a /tmp/data.txt
  ```

[6] http://wiki.ros.org/rqt_console

3.10 Comedi s526 Counter Read

Mit diesem Block kann der Zähler eines s526-Counters gelesen werden. Über den Block-Dialog in Abb. 5 lassen sich die ersten zwölf Bit des *Counter Mode Registers* (siehe [Sen09, S. 26]) konfigurieren, ohne den Registerwert dafür explizit angeben zu müssen.

Parameter (ohne *Counter Mode Register*)

- *Device* (Auswahl): I/O-Karte des Counters; üblicherweise `comedi0`.
- *Channel* (Integer): Kanal/Nummer des Counters.
- *Sample Time* (Float): Zeitintervall zwischen zwei Abtastschritten.

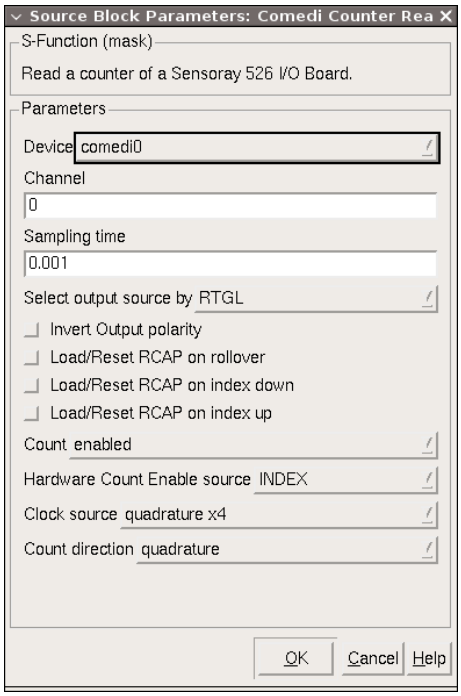

Abbildung 5: Dialog des Blocks *Comedi s526 Counter Read*

4 Eigene *S-Functions* und neue Nachrichtentypen

RTAI-ROS lässt sich leicht in eine eigene S-Function einbinden. Ein verständlicher Wunsch beispielsweise ist das Senden von Lognachrichten an ROS direkt aus einer S-Function, da dies um einiges flexibler ist, als Konstruktionen über Log-Blöcke. Es soll deshalb hier als Beispiel dienen. Der Aufbau ist recht ähnlich dem in Kapitel 5, Listing 5.1 in [Hop15] dargestellten Publisher, lässt sich aber mithilfe der Funktionen `registerRosBlock()` und `cleanRosBlock()` aus `ros_block.c` noch etwas komprimieren. Listing 1 zeigt alle notwendigen Änderungen an der S-Function, um für den Fall `error == true` in Zeile 25 eine Fehlermeldung an ROS zu senden. Graue Textzeilen geben den Kontext an, in dem die Änderungen durchgeführt werden. Der Name `rosout` in Zeile 13 dient als Platzhalter, der im Fall vom Blocktyp `LOGGER` nicht verwendet wird.

Listing 1: Verwendung von RTAI-ROS in einer eigenen S-Function

```
#ifndef MATLAB_MEX_FILE
#include <ros_block.h>
#endif

static void mdlInitializeSizes(SimStruct *S) {
  ssSetNumIWork(S, 1);
  ssSetNumPWork(S, 2);
}

#define MDL_START
static void mdlStart(SimStruct *S) {
#ifndef MATLAB_MEX_FILE
  rosBlockInitResult_t block = registerRosBlock(S, "rosout", LOGGER, 0);
  block.shm->msg.level = LOG_ERROR;
  ssSetIWorkValue(S, 0, block.num);
  ssSetPWorkValue(S, 0, (void *)block.shm);
  ssSetPWorkValue(S, 1, (void *)block.sem);
#endif
}

static void mdlOutputs(SimStruct *S, int_T tid) {
#ifndef MATLAB_MEX_FILE
  rosShmData_t *shm = (rosShmData_t *)ssGetPWorkValue(S, 0);
  SEM *sem = (SEM *)ssGetPWorkValue(S, 1);
  if (error) {
    if (rt_sem_wait_if(sem) != 0) {
      memcpy(shm->msg.text, "Error occurred", MAX_LOG_MSG_SIZE);
      shm->msg.state = NEW_VALUE;
      rt_sem_signal(sem);
    }
  }
#endif
}

static void mdlTerminate(SimStruct *S) {
#ifndef MATLAB_MEX_FILE
  cleanRosBlock(ssGetIWorkValue(S, 0));
#endif
}
```

Um RTAI-ROS um neue Nachrichtentypen zu erweitern, sind zusätzlich noch Änderungen an dessen Quelltext notwendig. Die dafür notwendigen Schritte sollen kurz anhand von `geometry_msgs/Vector3` erörtert werden:

1. In `include/ros_defines.h` ist eine eigene Nachrichtennummer als Präprozessorvariable zu definieren, z. B.

   ```
   #define PUBLISHER_VECTOR3  42 // bzw.
   #define SUBSCRIBER_VECTOR3 42
   ```

2. In `rtmain.cpp` ist die entsprechende Header-Datei einzubinden:

   ```
   #undef RT
   #include <geometry_msgs/Vector3.h>
   #define RT
   ```

Dabei ist darauf zu achten, dass die Präprozessorvariable `RT` zu diesem Zeitpunkt nicht definiert ist, anderenfalls kann es wegen der Verwendung der *Boost*-Bibliotheken zu Kompilierfehlern kommen.

3. Weiter unten im Quelltext wird in der Klasse `RosPublisher` bzw. `RosSubscriber` die entsprechende Übersetzung zwischen den seriellen Daten im Shared Memory und dem gewünschten Nachrichtentyp implementiert. Beim Publisher erfolgt dies in der Funktion `publish()` durch eine zusätzliche Verzweigung, z. B.

Listing 2: Ergänzungen in der Klasse `RosPublisher`

```
void publish() {
  ...
  } else if (subType == PUBLISHER_VECTOR3) {
    geometry_msgs::Vector3 msg;
    msg.x = shmData.data[0];
    msg.y = shmData.data[1];
    msg.z = shmData.data[2];
    pub.publish(msg);
  }
}
```

Beim Subscriber durch Überladen der Funktion `callback()`, z. B.

Listing 3: Ergänzungen in der Klasse `RosSubscriber`

```
void callback(const geometry_msgs::Vector3 msg) {
  if (!this->sem_wait()) return;
  shm->data[0] = msg.x;
  shm->data[1] = msg.y;
  shm->data[2] = msg.z;
  this->sem_signal();
}
```

4. In der Funktion `rosInterface()` wird im Initialisierungsteil einer Nachrichtennummer der entsprechende Nachrichtentyp zugeordnet, z. B.

Listing 4: Ergänzungen in `rosInterface()`

```
// Subscribers
} else if (rosBlockConfigs[i].type == SUBSCRIBER) {
  ...
  } else if (subscriber->subType == SUBSCRIBER_VECTOR3) {
      subscriber->sub = nh.subscribe<geometry_msgs::Vector3>(subscriber->name,
    rosConfig.subStackSize, &RosSubscriber::callback, subscriber);

// Publishers
} else if (rosBlockConfigs[i].type == PUBLISHER) {
  ...
  } else if (publisher->subType == PUBLISHER_VECTOR3) {
    publisher->pub = nh.advertise<geometry_msgs::Vector3>(publisher->name,
    rosConfig.pubStackSize);
```

5 Dateien

`ros_block.c`: Hilfsfunktionen für ROS-Blöcke

`rtairos_genfiles.tlc`: Zusatzskript für den Target Language Compiler

`rtairos.tlc`: Hauptskript für den Target Language Compiler

`rtairos.tmf`: Makefile-Template

`rtmain.cpp`: Hauptprogramm

`setup.m`: Installationsskript; kompiliert alle S-Functions in `devices/` und fügt dieses Verzeichnis dem Suchpfad von Matlab hinzu.

`devices/`

- `rtairos.mdl`: Simulink-Bibliothek
- S-Functions für Comedi:
  - `sfun_comedi_counter_read.c`: Lesen von Zählern
  - `sfun_comedi_data_read.c`[7]: Lesen analoger Eingänge
  - `sfun_comedi_data_write.c`[7]: Schreiben analoger Ausgänge
  - `sfun_comedi_dio_read.c`[7]: Lesen digitaler Eingänge
  - `sfun_comedi_dio_write.c`[7]: Schreiben digitaler Ausgänge
- `sfun_rtai_fifo.c`: S-Function zum schreiben von RTAI-FIFOs
- S-Functions für ROS:
  - `sfun_ros_config.c`: Config-Block
  - `sfun_ros_joint_state.c`: Joint-State-Publisher
  - `sfun_ros_joy.c`: Joystick-Subscriber
  - `sfun_ros_log.c`: Logger
  - `sfun_ros_publisher.c`: Publisher
  - `sfun_ros_service.c`: Service
  - `sfun_ros_subscriber.c`: Subscriber
  - `sfun_ros_tf.c`: Transformations-Publisher
- S-Functions für RTAI-Lab[7]:
  - `sfun_rtai_automatic_log.c`
  - `sfun_rtai_led.c`
  - `sfun_rtai_log.c`
  - `sfun_rtai_meter.c`
  - `sfun_rtai_scope.c`
  - `sfun_rtai_synchronoscope.c`
- `slblocks.m`: Konfiguration der Simulink-Bibliothek; sorgt dafür, dass die Blöcke im *Simulink Labrary Browser* erscheinen.

[7] von RTAI-Lab übernommen

`examples/`

- `rtaitest.mdl`[7]: Beispielmodell für RTAI-Lab
- `rosdemo.mdl`: Beispielmodell für ROS
- `rosdemo.perspective`: *rqt*-Perspektive für das Beispielmodell

`include/`

- `ros_block.h`: Hilfsfunktionen für ROS-Blöcke
- `ros_defines.h`: Gemeinsam vom Hauptprogramm und den ROS-Blöcken verwendete Konstanten und Datenstrukturen

Literatur

[Hop15] Hopferwieser, Roland: *Flachheitsbasierte Folgeregelung eines vierrädrigen quasi-omnidirektionalen Roboters mit ROS-Anbindung.* Masterarbeit, Institut für Robotik, Johannes Kepler Universität Linz, Linz, 2015.

[Sen09] Sensoray: *PC/104 Multifunction I/O Board Hardware Manual, Model 526*, 2009.