The driver concept for the DLR Lightweight Robot III

Robert Burger, Sami Haddadin, Georg Plank, Sven Parusel and Gerhard Hirzinger

Abstract— **In this paper we present the synchronization and driver architecture of the DLR LWR-III, which supplies an easy to use interface for applications. For our purpose we abstracted the robot hardware entirely from the control algorithms using the common device driver concept of modern operating systems. The software architecture is split into two modular parts. On the one side, there are device drivers that communicate with the hardware components. On the other side, there are realtime applications realized as Simulink Models, which provide advanced control algorithms. This ensures a clean separation between the two modules and provides a communication over a common and approved interface. Furthermore we investigated how we can ensure synchronization to the hardware over the device driver interfaces and how we can ensure that it meets hard realtime requirements. The main result of this paper is to realize a synchronization between LWR-III hardware and Simulink control applications while targeting small latencies with respect to hard realtime requirements. The design is implemented and verified on WindRiver™VxWorks™.**

I. Introduction

Implementing various control features of complex mechatronic robotic systems is a challenging task by itself and necessitates well designed tools as Matlab™/Simulink™. The theory and its realization concentrate on a system view and it is of large benefit to have a design that decouples the robot system from the control implementation. In current systems the robot hardware is controlled by a standard pc-hardware on which the control application is running. In the past when computers got more and more complicated, operating systems were developed which abstract the hardware from the software. The goal of our work was to take advantage of such mechanisms, that means to implement an abstract view to the underlying LWR-III hardware. Figure 2 corresponds to the classic view of control systems. Our goal is to produce highly modular, robust, and reusable code. In order to realize such features, we need an operating system that supplies distinct functionalities.

Nowadays operating systems mostly separate between kernel and user space. Kernel space is the place where the system kernel executes and provides its services. It has direct access to all hardware components of the system. The user space is remotely located from the hardware, which can only be accessed from the user space via system calls to the kernel. Its main benefit is to gain memory protection for applications running in the user space, thus increasing their robustness significantly. In this paper we use the aforementioned separation concept and implement it for the LWR-III robot control in order to logically, as well as from an architectural point of view separate our Simulink control applications entirely from the respective hardware. This makes it possible to encapsulate them from each other.

R. Burger, S. Haddadin, G. Plank, S. Parusel, G. Hirzinger, Institute of Robotics and Mechatronics German Aerospace Center (DLR), D-82234 Wessling, *robert.burger@dlr.de*

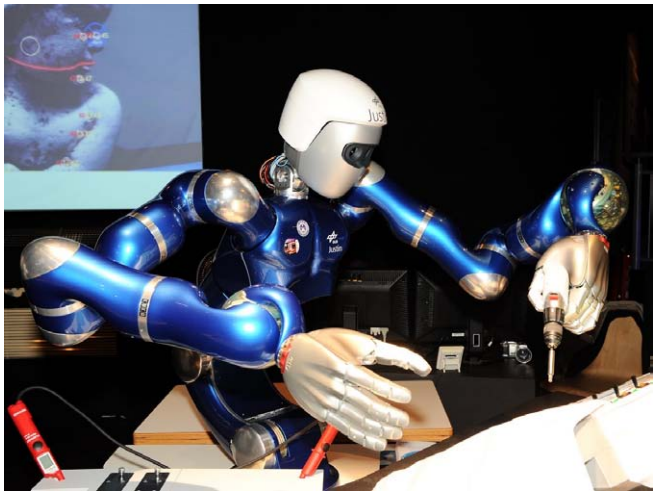

Fig. 1. The LWR-III based Space Justin

In realtime control there exist two common problems.

- How to abstract the real hardware from the control application?
- How to ensure synchronization among those applications, so that they are running in time with the hardware? In other words they have to be hard-realtime conform.

In robotic systems these requirements should be carefully considered, since they heavily influence the system performance.

In many existing designs the robot control applications are located in the same layer as drivers and/or the kernel. The main benefit of such a system is very small latencies, however, at the cost of lacking protection mechanisms. In case of failure, any program may crash the entire system. Thus, a way has to be found, to run the robot control in a more protected environment, fully detached from the hardware components. The control applications do not need any particular information about the LWR-III hardware anymore and the developer can focus on the pure algorithmic implementation. The only information required is how the attached system can be accessed and altered. Therefore we need a common and approved interface to the hardware that allows synchronized access for the control algorithms.

To cover the second problem current systems use common synchronization concepts like semaphores, events, or message queues. This is straightforward if we consider that all application programs run in kernel space and have access to

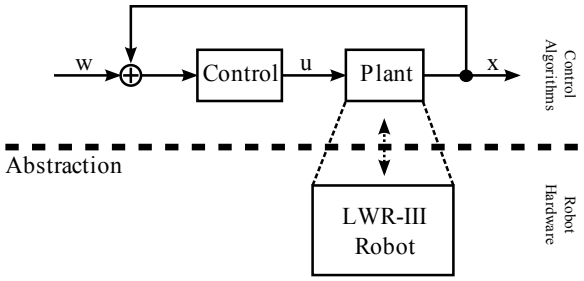

Fig. 2. Target structure for control implementation

each other. In this paper we mitigate the control application to user space and need other synchronization methods, as we cannot access the hardware driver directly.

The paper is organized as follows. Section II introduces the requirements for the interface to the underlying hardware. In Section III the state of the art is discussed to point out the used techniques. This includes basics about operating systems and the used POSIX device driver concept. In Section IV we apply that knowledge to our synchronization approach suitable for the LWR-III. Finally, in Section V and VI we give some programming examples and two of our implemented sample applications, showcasing the effectiveness of our implementation.

II. Robot interface requirements

For our concept we specified some few requirements for the interface to the robot.

- Abstract view of the real robot hardware
- Ensure synchronized access to robot data
- Deliver an interface for control purposes
- Ensure hard realtime requirements

To fit these requirements a common interface operating systems has to be found. One of the possibilities is to use the interface supplied by device drivers. These deliver an abstract view of the underlying hardware. The question now is, how to fit the other requirements when using device drivers to access the robot hardware.

III. State of the art

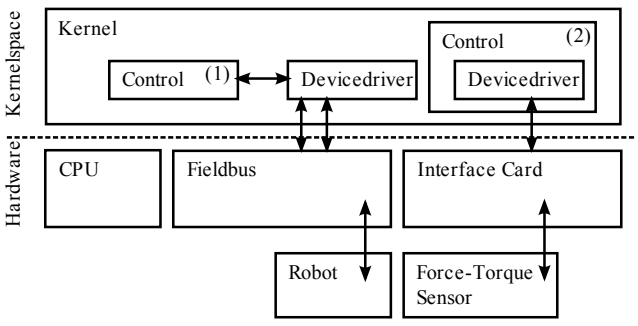

Fig. 3. Exemplary architecture of realtime robot control

Usually, in realtime operating systems all application programs run in kernel space and have therefore access to the entire system hardware and software. As described in [4][5][6] it is very difficult to clearly distinguish RTOS applications from drivers, even though such distinctions exist. The overall synchronization depends on the accessibility of every program part from any other process running on the target system. Such a cross-linked situation is of course potentially hazardous in case of software faults and is not well suited for maximum robustness.

As already mentioned, current control applications are usually embedded as depicted in Figure 3. Development of control applications is straightforward for this design and implements the device driver for the fieldbus and interface cards as part of the application (1), or provides direct access between each other (2).

The control algorithms run as a kernel task inside the operating system kernel and have direct access to the underlying hardware. This is definitely a non-ideal situation for complex mechatronic applications, potentially leading to an entire crash of the system with all its consequences. This is due to the fact that in kernel space protection of memory and system resources is usually not provided, causing a severe weakness and conflicting with many modularity requirements. This means that control applications need to be implemented such that they do not interfere with other tasks that are accessing the same hardware and drivers. This aspect cannot be derived from a control point of view, since it is typically desired to implement module control components with well defined interfaces that correspond to the particular system theoretical aspects.

Other typical tasks can for example be monitoring or parametrization tools. Most existing work discussing such aspects[1][2][3] covers more theoretical approaches to the existing problems as e.g. control engineering aspects of how to ensure deterministic delays in the control loop. However, they do not cover how to gain synchronized hardware access for robots in principle.

In the next section we introduce some of the required operating system concepts necessary to fully outline the presented concept.

A. Operating systems

In order to realize the aimed separation, an operating system is required that distinguishes between kernel- and user-space. The main reason to mitigate robot control to user-space is the aforementioned need for a significant increase in stability of the entire system. Examples for those systems, which are commonly used in realtime applications, are VxWorks™developed by WindRiver™[7] and QNX™from QNX Software Systems™[10]. VxWorks serves as the demonstration system in the present work.

A well suited operating system also needs to provide some common system features. On the one hand, it should be possible to provide standard operating system independent interfaces like POSIX. This ensures code reusability on other operating systems due to a common interface when using synchronization and communication features. On the other hand, it has to supply a device driver concept for our requirements. This concept is described in the next section.

B. POSIX Device Driver Concept

The purpose of device drivers is to obtain a clear separation of kernel- from user-space. Therefore, a common device driver interface is defined in most operating systems. One of its major benefits is that the applications implemented on top of the device interface may work system independent which is especially useful for robotics due to the variety of tools available for control design and implementations.

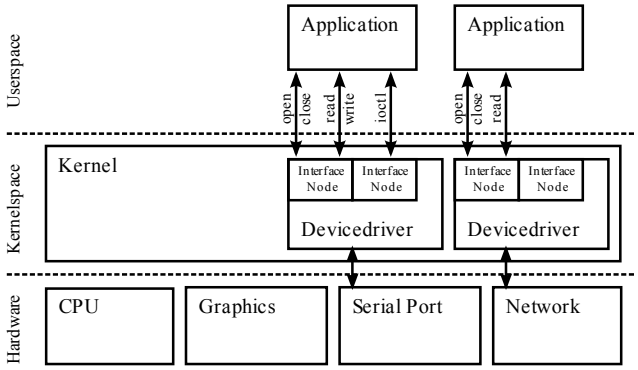

Fig. 4. Device Driver Concept

Figure 4 depicts the system view for the device driver concept in general. The hardware components as e.g. serial interfaces, graphics cards or network interfaces are shown in the hardware layer. The next petition implements the kernel including the device driver to gain the hardware access. On top of the kernel space the user space is located, where all applications run entirely separated from the kernel and therefore protected against each other. Between the kernel and user space exists a common interface. In Unix like systems as VxWorks the interface is accessed via so called device files. These device files are created somewhere in the file system (usually in /dev).

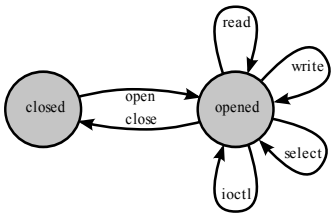

Fig. 5. State Machine Device Driver

They provide only a well defined set of functionalities, show in Figure 5, where the different states of the device files are given. The states are the common sense in most operating systems. The relevant transitions between the two states *closed* and *opened* are described below. They are common to all device drivers, so their access is unique and therefore provides a common interface.

1) open/close: Opens or closes the device-file for manipulation. This has to be done before reading, writing, or parameterizing the device-file. *Open* usually returns a file descriptor to the corresponding file or device. *Close* takes this file descriptor and ends all device-file access.

2) read/write: Reads or writes from, or to a stream-based device. Usually, these calls are parametrized by the previously received device-file descriptor and a memory buffer for *read* or *write*.

3) ioctl: ioctl sends a special command to the device-file as e.g. reset or mode selection. It requires the file handle, the request code, and optionally a memory-buffer as argument.

4) select: This call allows every program to monitor multiple device files. It can be useful if one needs to know whether there is data available from more than one device. The calling program is waken if one or more file descriptors are readable, writable, or if any of them throwed an exception. The program can also specify a timeout.

C. Mapping POSIX Device Driver to robot

We can map the POSIX Device Driver concept to our robot as depicted in figure 6. On the left side we have our control task which monitors the robot to detect violences of the hard realtime and sends simple commands to it with *ioctl*. This includes e.g. access to the robot state machine or exception handling. On the right side we have our control loop, which retreaves meassured data from the robot via *read* calls. After it does all its calculations it sends the new values to the robot with a *write*. After that it needs to get new meassured values from the robot.

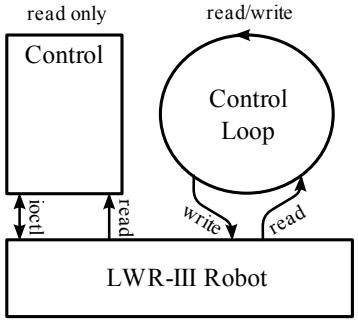

Fig. 6. POSIX Device Driver mapping

To ensure the hard realtime requirements, the device driver has to observe, if the control loop is sending new values every clock cycle. If a violation is detected, the driver switches the robot to a safe state. This can be configured and monitored from the control. The synchronization to the robot is to be discussed in the next section.

D. Summary

In realtime systems *read/write* can provide access to cyclic data on the specific realtime hardware. This hardware can be controlled with calls to *ioctl*. Its interface is designed for gaining common access to different types of device drivers. Special device features are handled via *ioctl*. For each of this interface functions the operating system issues a system call. This involves a context switch from user-space to kernel-space and vice versa. This leads to some additional delay, which role is discussed later in Section IV-D.

Next we present the hardware synchronization mechanism for the LWR-III.

IV. Hardware Synchronization

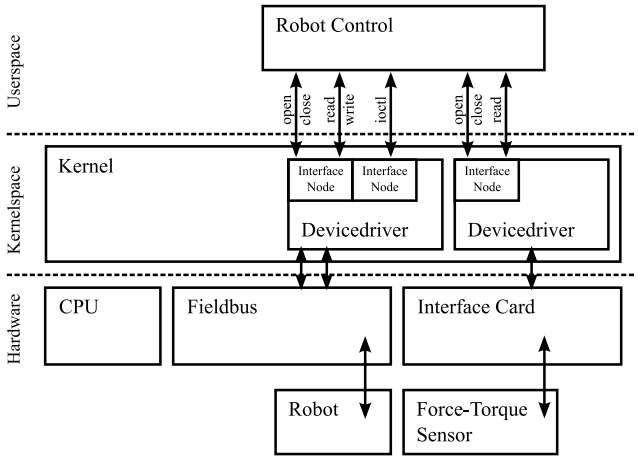

Fig. 7. New Synchronization Control Architecture for a robot

The mapping of the device driver concept to our realtime robot control is a simple mapping. With calls to *open* and *close* the specific control gains or looses access to the underlying device. To receive current values from the attached robot or sensor device, the control only calls *read* on the previously opened device. Then, the current values are copied into a buffer supplied by the robot control. After the control finishes its calculations, the command values need to be written back to the device. This is done by calling *write*, which writes them to the underlying attached device. The resulting synchronized control architecture is given in Figure 7.

The common problem to be solved now is the synchronization of control with hardware. This is of major importance, since otherwise our control applications would need to incorporate latencies, leading to more complex theoretical analysis and possibly unstable system behavior. The synchronization can be implemented in three different ways by using blocking system calls:

- *blocking read*
- *blocking ioctl*
- *select*

One benefit of such a hardware synchronization is the support for multiple control or monitor applications. If more than one application is in blocked state, all of them have to be unblocked in case a new tick (that means new cyclic data/actual values from the LWR-III) is received from the device driver. This makes it possible to run some monitoring tools in parallel to the main robot control application.

In the following we shortly describe the working principle of the three synchronization schemes.

A. blocking read

If the device is opened in blocking mode, every call to *read* will block the calling application until the next driver tick is available. Figure 8 illustrates the synchronization scheme with *blocking read*.

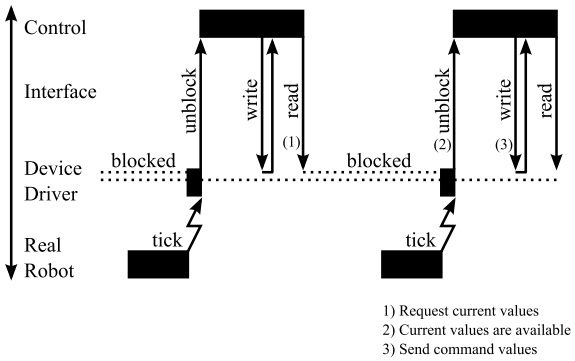

Fig. 8. Synchronization *blocking read*

Usually the device driver receives a tick if new values from the robot are available. In this case all attached control applications are unblocked immediately, allowing them to start calculations. The major benefit of this approach is that the control application reads the most recent available data for the next step of calculating e.g. new actuation variables (torque).

B. blocking ioctl

The second possibility to synchronize our control with the LWR-III hardware is a *blocking ioctl* call. This also blocks the calling application until the next driver tick is available. Its scheme is depicted in Figure 9.

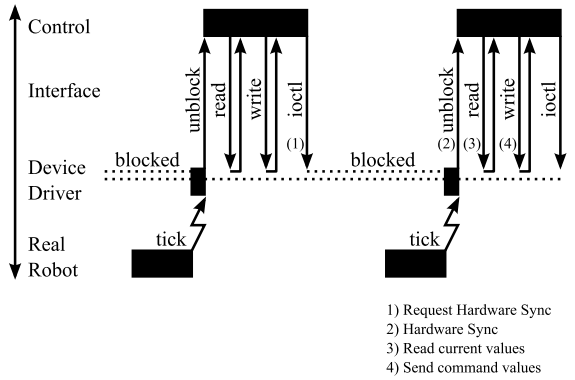

Fig. 9. Synchronization *ioctl*

The resulting behavior is nearly the same as for *blocking read* with one significant difference. If new robot values are available from LWR-III, the driver immediately unblocks all waiting applications which are blocked inside the *ioctl*. Then they have to perform a *read* call to retrieve the new values. This leads to a small overhead compared to the *blocking read* consisting of one more system call. Alternatively, the *ioctl* could be parametrized with a timeout value. If no device driver tick occurs until this timeout, the ioctl unblocks and returns an error code. Then, the application could react to this timeout in a proper way.

C. select

The most promising possibility we implemented for synchronization is to perform a *select* in order to synchronize to more than one hardware device. This call blocks the control until one of the given file descriptors is readable, writable, or received an error. In most cases one would use *select* to wait for a readable file descriptor, which means that new cyclic data is available. The timing behavior looks very similar to blocking *ioctl*, e.g. Figure 10.

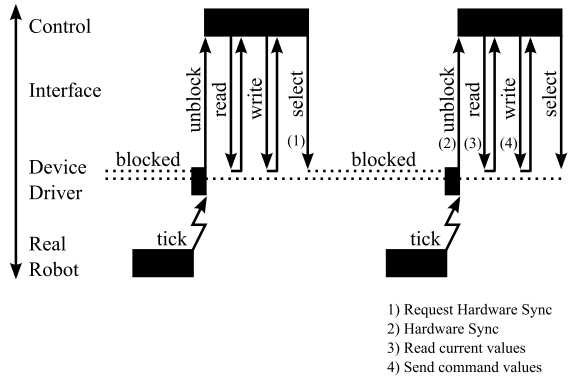

Fig. 10. Synchronization *select*

This also leads to a small overhead compared to blocking *read* by one more system call.

The main criterion for choosing the preferred *select* synchronization scheme is whether the time delay caused by a system call produces significant latencies.

D. Time measurement

In order to determine the impact of system calls to the realtime system behavior time measurements are performed. These tests were carried out on VxWorks 6.7 running on a Pentium 4 with 3.0 GHz.

	time [μs]
average time	1.108
average deviation	0.002
maximum deviation	0.035

TABLE I
SYSTEM CALLS TIME MEASUREMENT

Table I gives the quantitative results. The average time for a system call is ≈ 1 μs, and the average and maximum jitter are very low and fulfill hard realtime requirements.

In Figure 11 the measured times for system calls are shown for 1000 test cycles. These tests were made a big ymount The single peaks represent the maximum jitter of ≈ 0.035 μs. The average jitter (exposed area) is around 2 nanoseconds.

These tests show that the impact of system calls is negligible. For both synchronization methods - *blocking read* and *blocking ioctl* we obtain a very low delay. The blocking *ioctl* and the *select* need one extra system call, so 50 % more system calls than for *blocking read* are required. Thus, one calculation step needs extra 2.2 μs in case of *blocking read* and 3.3 μs in case of *blocking ioctl*. The respective system call count and the additional latencies are shown in Table II.

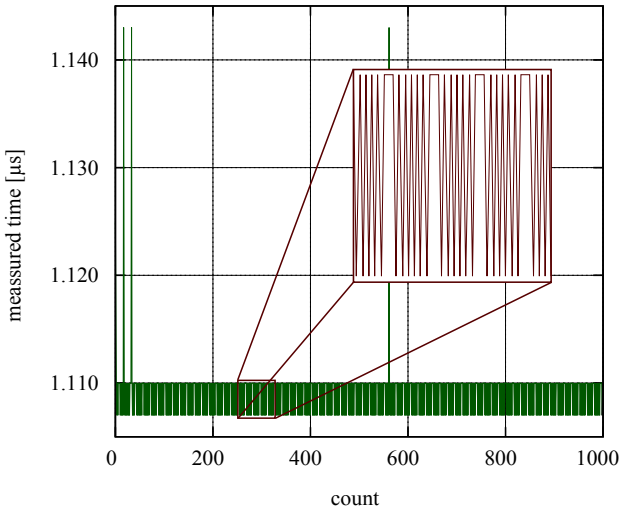

Fig. 11. Jitter measurement

	system calls	time [μs]
blocking read	2	2.2
blocking ioctl	3	3.3
select	3	3.3

TABLE II
AVERAGE LATENCIES

E. Programming Concept

There are multiple ways of writing applications, that are synchronized with the hardware. The first step would be to open the device and acquire a file descriptor. For example a device is opened for read/write access and called "/dev/mydev". There are mainly two common flags needed for our concept. The Flag `O_RDWR` gains read/write access to the device and `O_NONBLOCK` ensures that no blocking operation during *read* is performed. If `O_NONBLOCK` is not supplied, the device is opened in blocking mode.

```
fd = open("/dev/mydev", O_RDWR);
```

Now, the application is synchronized to the hardware by simply performing *blocking read*s. This can be done by supplying a buffer to store the actual cyclic data from the attached robot. This call returns the number of bytes actually read. This call will block until new data from the robot is available. Calculated cyclic data could also be written to the device. The *write* call looks similar to the *read* call, which would send the data to the attached robot and return afterwards immediately. The device driver monitors if a *write* call is made every clock cycle. Otherwise the robot will be switched to a safe state.

```
bytes_read = read(fd, buf, size);
bytes_written = write(fd, buf, size);
```

Asynchronous access to the hardware is also possible if the device is opened with the `O_NONBLOCK` flag. The

synchronization should be done with blocking *ioctl* calls. This call also blocks until new robot data is available as the *blocking read* call does.

```
ioctl(fd, DEV_IOCTL_WAITTICK);
```

With this very simple chain of commands it is now possible to write control applications that are synchronized with the robot hardware.

F. Comparision

The three presented synchronization schemes can be used depending on the system requirements. *Blocking read* is always usefull if just one device is used for synchronization and is supplying sensor data to the control application. *Blocking ioctl* may be used in case that the device is supplying only a clocktick and no sensor data. Synchronization to multiple devices can be made with *select*, where the application will be informed, if one or more of the selected devices are signalling that they are ready. In Table III the comparision between the different schemes is shown.

	hardware sync	sensor data	multiple devices
blocking read	yes	yes	no
blocking ioctl	yes	no	no
select	yes	no	yes

TABLE III
SCHEME COMPARISION

In the next section we show practical programming examples of our implementation.

V. USING THE POSIX INTERFACE

This section addresses some possible programming options. The most interesting method for control engineering is the usability of modeling tools as Matlab™/Simulink™[8]. Furthermore, the implementation of synchronized utility tools was carried out.

A. Command line Tools

A positive side effect of this device driver concept is the possibility to use multiple applications simultaneously. As mentioned before, we may run monitoring utilities in parallel to the main control application. A very basic option to control the realtime hardware we implemented are rudimentory command line tools for parametrization and diagnosis. These should allow to set or get parameters, modes, etc. from the attached devices, or to supply monitoring features. The utilities should run with low priority so they do not interfere with the main robot control. A well known example for such monitoring tools are file loggers or tcp loggers. The logged data can then be processed and analyzed off time.

B. Matlab™/Simulink™RealTime Workshop™

Matlab Simulink in combination with RealTime Workshop (RTW) allows the generation of universal code from Simulink models. Access to hardware is granted by an individual S-Function-Block. S-Functions offer the possibility to embed C/C++ code into Simulink. A S-Function is implemented with a common interface, which consists of a start and a terminate routine. The calculations steps are done by the update routine.

1) Model start: The S-Function usually opens the hardware device in his start-routine.

```
void mdlStart(SimStruct *S)
{
  fd = open("/dev/mydev", O_RDWR);
}
```

In the start routine the device file is opened to acquire a file descriptor to the device for later usage. This file descriptor can be stored in one of the S-Function vectors.

2) Model terminate: The Simulink model terminates all S-Functions with a call to their termination routine.

```
void mdlTerminate(SimStruct *S)
{
  close(fd);
}
```

At this point the acquired file descriptor should be closed.

3) Model update: In each Simulink calculation step the model calls the update-routine of the S-Function. This update-routine should at first write the last calculated values to the driver and then read the current values from it. The *read* blocks until new values are available as described in section IV-A.

```
void mdlUpdate(SimStruct *S)
{
  ... get calculated values ...
  write(fd, wbuf, sizeof(wbuf));

  /* read from device (blocking) */
  read(fd, rbuf, sizeof(rbuf));
  ... set actual values to model ...
}
```

The given examples show that it is possible to use nearly the same code basis as for the command line tools in order to embed synchronization structures into the interface for control engineering tools as Matlab/Simulink. In the next section we present some of our implented application on top of this concept, resulting in a full separation of synchronization schemes from the robot control.

VI. APPLICATION TO ROBOT CONTROL

At DLR we have implemented multiple applications, using the presented synchronization architecture. They consist of a standard pc hardware platform and a DLR LWR-III. The LWR-III is connected via the serial fieldbus SERCOS to the computer. This bus is running at a cycle time of 1 kHz and triggers all the applications with its clock rate.

A lot of full scale applications were already implemented, incorporating full robot control, external sensoring, and reasoning. We developed a co-worker scenario[12] and a robotic billiard player[11] (both Figure 13). Furthermore we applied this to concept to our Space Justin (Figure 1), to our newly developed DLR Walker (Figure 13) and to a Brain Machine Interface[13] (BMI, Figure 13).

As depicted in Figure 12 we implemented the target scheme depicted in Figure 2. In this model we can switch between the real robot and its full dynamic simulation at runtime. This significantly accelerates development and testing

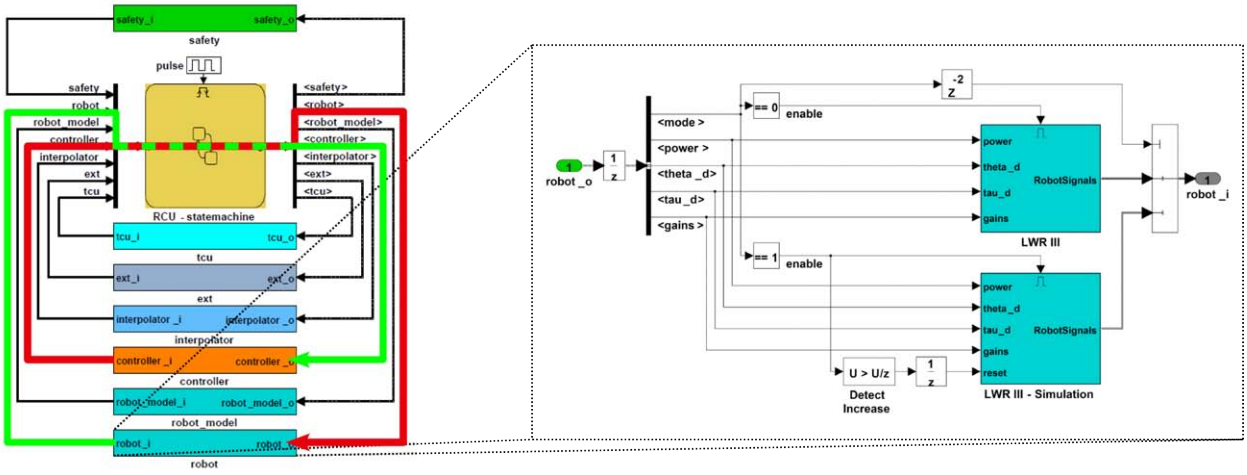

Fig. 12. Simulink implementation of control with robot interface, carrying out the presented synchronization concept

Fig. 13. Implementations (from left to right): DLR Walker, Coworker scenario, Billard player, BMI

times for the sophisticated control components. The target scheme is integrated in the full robot control architecture. The merit of this is to achieve a logical separation of the control components and the attached robot hardware. This Simulink example can be compared to Figure 2. The red line shows the outputs (u) of the implemented controller (*Control*) to the robot (*Plant*). The meassured sensor values (x, green line) and the desired values (w, blue line) are connected to the inputs of the controller.

VII. Conclusion

In this paper we implemented a synchronization to the LWR-III hardware in consideration of a clean separation between kernel and user applications. With this implementation it is possible to develop control algorithms hardware independent and reduce development cycles. In our implementation control applications run now in the protected user space and cannot touch or be affected by other parts of the system.

References

[1] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE, 91, 64 83.*

[2] Benveniste, A. and Berry, G. (2001). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE, 79(9), 1270 1282.*

[3] Kopetz, H. and Bauer, G. (2001). The time-triggered architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software.*

[4] Bill Weinberg, Porting RTOS Device Drivers to Embedded Linux, *http://www.linuxjournal.com/article/7355*, 2004.

[5] Swetanka Kumar Mishra & Kirti Chawla, RTOS - Design and Implementation, *www.cs.virginia.edu/ kc5dm/colloquiums/rtos.ppt*, University of Virginia.

[6] Dr. Peter Troger, Dr. Siegmar Sommer, Embedded und RT-Betriebssysteme, *https://devel-rok.informatik.hu-berlin.de/svn/EMES2009/web/13-os.pdf*, 2010.

[7] WindRiver, VxWorks, *Wind River Way, Alameda, CA 94501 USA*, *http://www.windriver.com*

[8] The MathWorks USA, *The MathWorks, Inc. 3 Apple Hill Drive, Natick, MA 01760-2098*, *http://www.mathworks.com*

[9] Shannon, C.E. (1948). A mathematical theory of communication. *The Bell Systems Technical Journal, 27, 379423,623656.*

[10] QNX Software Systems, QNX, *175 Terence Matthews Crescent, Ottawa, Ontario, Canada, K2M 1W8*, *http://www.qnx.com*

[11] Sven Parusel, Playing billard with an anthropomorphic robot arm, *Kempten & German Aerospace Center (DLR)*, 07-2009

[12] Sami Haddadin, Michael Suppa, Stefan Fuchs, Tim Bodenmueller, Alin Albu-Schäffer and Gerd Hirzinger, Towards the Robotic Co-Worker, *International Symposium on Robotics Research (ISRR2007), Lucerne, Switzerland*, 2009

[13] Jörn Vogel, Sami Haddadin, John D. Simeral, Sergej D. Stavisky, Dirk Bacher, Leigh R. Hochberg, John P. Donoghue, Patrick van der Smagt: Continuous Control of the DLR Light-weight Robot III by a human with tetraplegia using the BrainGate2 Neural Interface System, *International Symposium on Experimental Robotics (ISER2010), Dehli, India*, 2010.