Object-oriented, open environment with Linux for hydraulic boom controller

Jari Savolainen, Jyrki Kullaa, Miki Wiik, Jari Söderholm

Intelligent Machine Control project, Helsinki Polytechnic Stadia

The objective of this study was to develop an intelligent control system for hydraulic booms and other heavy machinery. An object oriented open source software development environment was utilized and the system was implemented using an embedded device with Linux as the operating system. The software was written in C and C++. Open standards and libraries, mainly the GNU C and C++ libraries, the POSIX standards and the CANopen communication protocol, were used. The CANopen protocol was modeled with C++ classes and presented by Unified Modeling Language (UML). A Domain-Specific Modeling language, specifically targeted at hydraulic booms, enables the initialization of the control software to different kinds of hydraulic booms and enables the modification of the control software using the concepts of the domain. By utilizing the scheduling features of the 2.6.-series Linux kernel, the control system achieved soft real-time behavior. The results were validated using both a simulated environment and a full-scale hydraulic boom.

1 Embedded soft real-time Linux

The requirements for mobile hydraulic system are very demanding and involve the hardware, operating system and software. This study utilized a readily available hardware platform and concentrated on researching the suitability of Linux as the operating system on which to develop and run the control software.

The chief reason behind the growing popularity of Linux is its openness. Source code as well as binaries is freely available for distribution and modification, which is backed up by a global community of users, developers and distributors. In embedded environments the use of Linux is growing a rapidly due to the following facts: It offers compatibility with many existing Application Programming Interfaces (API) and supports a vast number of communication protocols, the kernel is ported to a multitude of platforms and device drivers for most hardware are readily available. There are many programming tools for Linux, some of which are targeted for embedded software development. Many of these programs are available at no cost, making them an inexpensive alternative to conventional embedded software tools. Furthermore, it is easy to find Linux developers.

Much of the development of Linux has traditionally been targeted at server and workstation environments where overall throughput is prioritized. However, recent years have seen an increase in activity of the developing Linux for embedded environments, where the focus is shifted to system timing requirements and effective use of resources. The embedded system used for this study, as well as many other embedded systems, impose strict timing requirements on the operating system.

Although the current Linux kernel cannot guarantee hard real-time response, there are a number of modifications to improve upon this. However, in soft real-time environments were occasional deviations are allowed, the 2.6.-series Linux kernel fares quite well. The standard scheduler of Linux separates real-time processes and gives them priority over normal ones, resulting in soft real-time response on most systems.

There are number of ways in which a Linux distribution can be modified in order to gain improved real-time behavior. By modifying kernel features that affect scheduling and timing, and by disabling all unnecessary services and features gains in timing can be made. Furthermore by minimizing the use of slow hardware devices and system calls, the

responsiveness of a system can be increased substantially.

The standard 2.6. series kernel running on an Intel architecture is generally capable of a response time in the millisecond range. If a lower response time is desired, there are kernel patches and modified variants available that can reduce the response time down to the range of 20 microseconds. Currently, the two most popular open source real-time solutions are the Linux kernel real-time pre-empt patch and Real-time Application Interface (RTAI).

The Linux kernel real-time pre-emption patch adds kernel pre-emption points in order to lower latencies. The patch also uses dedicated pre-emptive kernel threads for the interrupt handlers. The real-time patch is constantly developed and many parts of it have been incorporated into main Linux kernel. RTAI is patch for the standard Linux kernel that enables hard real-time in Linux. It can execute applications in both user and kernel space and is capable of a response time in the 20 microsecond range, where user space adds a delay of a few microseconds compared to kernel space. RTAI is implemented as a nanokernel, where the Linux kernel is run as a low priority process that is executed when no hard real-time tasks are scheduled for execution. In addition to the aforementioned, there are also some commercial implementations of hard real-time enhancements to Linux.

In this study we used a standard Linux kernel that was compiled with features aimed at reducing response time. A monolithic approach was taken in that only the CAN interface card driver was compiled as a module. A specific distribution was created, that contains only the essential programs and services and that can be executed in read-only mode from flash memory. The control system itself is executed in a continuous loop with a loop time in the 5 millisecond range. The control system is used to drive a hydraulic boom with very good results, confirming the suitability of Linux as an operating system for this type of embedded environment. For further informations see [1][2].

2 Design consideration and programming techniques

Embedded systems of a soft real-time type, such as the one used in this study, are designed with longevity and stability of operation as primary goals, making absolute performance a secondary consideration. The need for stable operation is further emphasized in systems where the user has little or no possibility to interact with the underlying software other than by performing a complete system restart. Since such situations are unacceptable at best, extra safeguarding mechanisms need to be introduced in order to avoid them.

Typical application programming is targeted at powerful multi-tasking systems, where the application is run as a result of user-driven events. Embedded systems differ from this in that they typically run a few specific applications that comprise the functionality of the device. As a result, the timeliness of embedded software becomes easier to predict and the software can allocate most of the resources of the platform.

The software of the control system designed for this study performs a predefined sequence of operations in a timed loop, and as such it is a typical embedded device. For debugging purposes, the sequence is split into logical parts that are implemented as separate processes. In order to facilitate communication between the processes, a shared memory segment containing all relevant data was utilized. In addition to simplifying the structure of the processes this has the advantage of providing a real-time interface to the data that can be used for debugging or by other applications. The control system utilizes semaphores as a synchronization primitive with which to protect the shared data, but also as a mechanism in which to run the separate processes in a predefined sequence. Furthermore, this enables slow operations, such as traffic with the CAN to be implemented as separate processes, resulting in effective use of the systems resources through multitasking. Figure 1 displays an overall picture of the separate

processes and the shared memory segment.

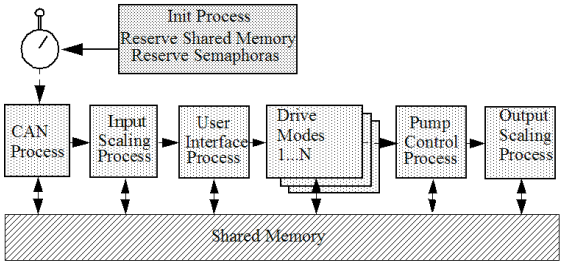

Figure 1: Process model of boom

The popularity of the C++ programming language in embedded environments is increasing. The object oriented programming paradigm of C++ is powerful, yet intuitive and offers an improvement to traditional procedural C. Yet, as C89 forms the subset of C in the C++ standard, incorporating an existing C code base into C++ is very flexible. However, although C and C++ share the same appearance, their basic structure differs greatly and a transition from procedural C to object oriented programming C++ is also a transition in focus from program flow to architecture and components.

The object oriented paradigm offers many benefits: development can be done using advanced case tools, the paradigm gives possibilities to use many kinds of components and design patterns and reverse engineering tools can be used to create ready code from graphical represented class models. There are drawbacks as well: the transition from structural to object-oriented programming can be a demanding task, all object-oriented concepts are not feasible in real-time environments and the size of the executable code can be larger than in a procedural program. However, much of the growth can be located to the constructing and destructing phase of objects. By keeping these outside real-time loops of execution, object-oriented technology can be used even in digital signal processing environments, where real-time requirements are extremely hard.

The dynamic memory management facilities of C++ introduce the possibility of memory fragmentation that have the potential of causing long term instability and added overhead. This study makes use of dynamic memory allocation, but in a stable manner where all dynamic memory allocation is done during construction of the object and all objects are constructed at startup. De-allocation of memory is done at object deconstruction, which in turn is done as system shutdown. As a result, the memory footprint of the program is kept stable during operation and by locking all used pages in memory; the possibility of swapping is eliminated. In future development will require dynamic memory allocation at runtime, an object-caching mechanism similar to the slab layer provided by the Linux kernel, could be utilized.

The software used within this study consists of multiple processes that are run in a predefined sequence, resulting in consistent operation that is straightforward to debug. By excluding the more complex facilities of C++ such as exception handling, run type information and dynamic casting, the code itself can be kept simple and straightforward while keeping the memory overhead low. Even though C++ allows the construction of very complex objects and inheritance relationships, special care needs to be taken to ensure the consistency in execution time of branches and child objects. If need be, delays or timers could be utilized in order to guarantee timely operation.

Although embedded systems traditionally rely heavily on C, there is no overwhelming reason that would prevent the use of the C++, as long as the special considerations of the environment are kept in mind. Indeed, the development of the software for this study has come to completely rely on C++ with excellent results. For further information: [3].

3 Object representation of the hydraulic boom

The shared memory segment that the control system utilizes for Inter-Process Communication (IPC) contains all the relevant data of the hydraulic boom using object representation. The data container object contains object representation of the physical parts of the boom, which provides an intuitive way to store the data

collected from specific devices. Figure 2 displays the layout of the data container.

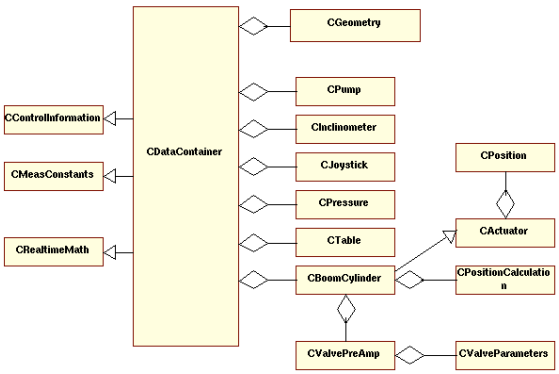

Figure 2: Data container of components

An application for a mobile machine has a lot of device-specific configuration parameters. By combining the object representation of the hydraulic boom with domain specific GOPRR, the complexity of device configuration can be simplified and automated. Figure 3 shows a DSM model of the hydraulic boom, which is used by a code generator to automatically produce the all initialization functions needed by the control system.

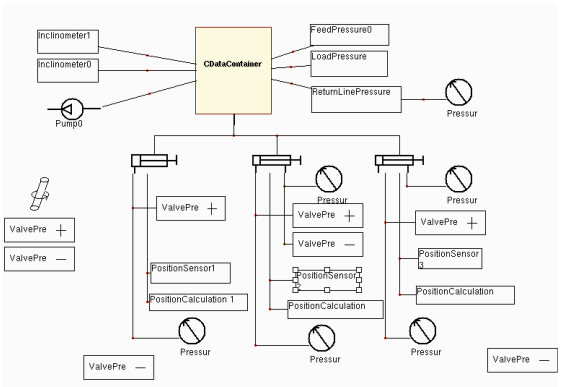

Figure 3: DSM model of boom

The control system itself contains intelligent functionality for electronic load compensation, negative load control of cylinders, active vibration damping and a x-y-coordination drive based on Jacobian matrix. Thus, the control system provides intelligent features, yet is easy to configure.

4 Using UML in embedded development

The aim of UML is to be a helpful tool for analyzing, architecture modeling and the creation of software systems. One idea of UML is to help a project group in planning the whole system and developers of details to understand the principles of the whole program. UML can be introduced to the development process gradually. If a system is designed using object oriented languages such as C++ or Java, it is possible to convert class headers automatically to UML class diagrams. Typically, generated headers are very similar to manually produced code. However, complete automatic creation of functions based on UML is not possible. To maximize the benefit of UML development tools they need to communicate well with the rest of the development environment. Optimally, UML should be used as an alternative view to the code and for effective debugging.

5 Domain specific modeling

The starting point of the Domain Specific Modeling (DSM) paradigm is very different from UML. DSM models do not contain unnecessary information and they are possible to develop without using object oriented vocabulary, instead the model is built using the terminology of the application, which makes understanding the model easy without knowledge of programming terminology. The application code is created by parsing through all models, therefore the consistency of the models is good. DSM models are more tightly bound to each other than UML models and don't contain undocumented relations. All parts of the code, except libraries, are presented as models. Flaws in the logic of an application are relative easy to find because all model problems are mirrored in the code immediately at the next code generation.

With DSM it is possible to develop easily decentralized programs. As one data is presented only once in a model, it is possible to take fragments of a model an reuse it in other places of the code. Consistency between processes is easy to control with tight relations in the model. The developer of the DSM project defines the models which belong to application. Therefore all objects within the application are objects of only the specified domain. By using a programmable code generators it is easier to produce more optimized

code than that of fixed generators. The CANopen stack of the control system produced by this study contains an example of code generation. A typical CANopen stack contains between 1000 and 2000 rows of source code compared to the 150 rows of the implementation of this study. Figure 4 displays a sample model of a boom using the DSM of the boom domain.

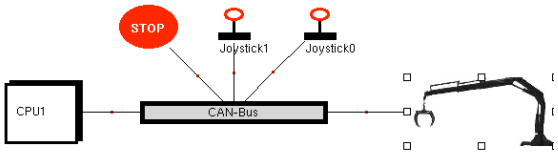

Figure 4: DSM model of a hydraulic boom

When defining the modeling language of the domain, GOPRR-notation (Graph, Object, Property, Relationship and Role) can be used to create a graph-based meta model of the language. All objects used within the project are drawn into graphs that contain the objects roles and the relationships thereof. GOPPR objects are more common than UML objects and can be for example a process, a thread, a class or an instance of class. A property describes features of graphs, objects, roles and relationships. A relationship connects objects by assigning them roles in the activity of the object.

Using GOPRR graphs it is possible to automatically produce code by parsing every graph in a project. The coherency of GOPPR graphs is better than in UML models. UML code originates from every model separately and therefore the consistency of UML models is not guaranteed. GOPRR is more flexible than UML, but at an expense. It is not possible to create a DSM and code generators before the specifications of a product family exists, except when using a standard modeling language, such as UML and SA/SD. Furthermore, not all of the intelligence of a system can be included in the models, but some have to be included in the code generator. Keeping these limitations in mind, DSM can become a valuable tool in developing and configuring product families.

6 Object Model of the CANopen protocol

The various messages of the CANopen application protocol make use of the 11-bit identifier and 8 byte data payload of the CAN message frame in a diverging fashion. Depending on the message, a data field can contain both specific information, such as the message identifier, and indirect meta-like information such as the originating node of a TPDO. This complexity is increased further by the added abstraction of the object dictionary.

At first glance, there is no apparent pattern in how the different messages utilize data, which from a programmers perspective means that the CANopen protocol might prove challenging to master in a simple and straightforward manner by just manipulating the raw data. Furthermore, since some bytes are split into subgroups, the data itself cannot always be interpreted in a simple manner using hexadecimal notation, resulting in reduced legibility[1].

The CANopen protocol does however exhibit aspects that combined with object-oriented programming techniques result in a hierarchical and simple categorization of the various CANopen messages according their type. All messages rely on the data fields of the CAN message frame, but make use of them in a predefined manner. Thus, a parent-child relationship is formed, were the child provides its own implementation for the data fields. This object-relationship is further evident when used in conjunction with the pre-defined broadcast and peer-to-peer connection sets as illustrated by figure 5.

[1] For instance, the first three bits of the SDO command specifier byte represents the type of transfer, while the interpretation and value of the fourth bit varies.

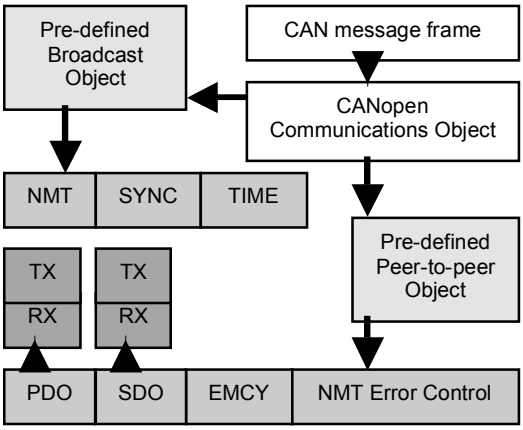

Figure 5: Object relationship of CANopen messages

By utilizing object representations of the CANopen messages, rather than manipulating the raw data, all data can be formatted and processed with respect to the message in which it is contained. Consequently, the developer is presented with a interface that is both intuitive and presents the various CANopen messages in a internal relationship that is in accordance with the standard. Furthermore, using message objects as the primary data primitives increases legibility and correctness by providing automatic formatting of the various data fields.

A completely object-oriented CANopen application is currently being developed by this study. By including object representations of the physical CAN nodes, the application will provide a powerful and intuitive interface to the CAN, with a one-to-one relationship between the physical devices, the CANopen protocol and the software objects.

For further information

CANopen generic pre-defined connection set:

CiA draft Standard Proposal 301. CiA Version 4.1. 21 February 2006.

7 Development of the drive modes

The control system was utilized in the development of a coordinate drive mode for a vehicle crane. The coordinate drive mode enables moving the tool or tip of the crane along horizontal or vertical trajectories instead of controlling the joints independently.

The actuator velocities are solved at each time increment using the geometry information of the crane and the current configuration of the joints. The geometry parameters are fixed, whereas the current position has to be updated at each increment. The joint positions are measured using a displacement sensor at each cylinder.

The kinematics software then solves the actuator velocities that are required to drive the tool along the specified axis at a given speed. For this the algorithm utilizes matrix algebra. The actuator velocities are converted from binary format to voltages that the valve amplifier uses, by using the relationship between the volume flow and voltage to the valve.

By using the control system, it was possible to concentrate on the kinematics problems, rather than the technical details that were irrelevant to the problem at hand. The position sensor data was visible as variables within the software and the computed actuator outputs were directed to corresponding valve amplifier. The coordinate drive mode could be selected with a joystick button, after which joystick motions were defined as tool velocities in each axis direction.

8 Conclusion

The use of Linux in embedded environments is growing. This study did not reveal any limitations of the operating system that would prevent its use in embedded soft real-time applications for mobile machines. Indeed, Linux has proved itself very suitable for the purposes of this study. If, however, hard real-time operation is desired, the standard Linux kernel does not provide sufficient support. Instead, patches to the kernel or alternative operating systems should be considered.

Timeliness and stability of operation need to be prioritized in the design of embedded soft real-time systems. By excluding the more complex facilities of C++, the language has been used to realize an object-oriented control system that adheres to the requirements of timely

behavior and a stable memory footprint. In addition, by combining object representations of CANopen messages with object representations of the physical CAN nodes, an interface to the CAN was created, with a one-to-one relationship between the physical devices and their software object counterparts.

The mobile machine industry has many possibilities to make gains in productivity by the effective use of available software development technologies. UML has achieved a strong position, but it is currently being mainly used for planning and documenting purposes. DSM-GOPPR adds a new level to software development by allowing intelligent project management inside product families. Furthermore, DSM allows automating the development process, which is becoming increasingly important as the complexity of software is continuously increasing.

Conclusively in can be said, that this study has reinforced the view that programming techniques commonly used in desktop application development and the Linux operating system can be successfully employed in soft real-time environments.

References

[1] The Linux kernel pre-empt patch: http://people.redhat.com/mingo/realtime-preempt/

[2] RTAI Real-time application interface: https://www.rtai.org/

[3] Robert Love, *The Linux slab layer*, Linux Kernel Development, 2nd Edition. Novell Press 2005.