Flexible Handling of System Software for Mobile Robots by Using a Module Loader Concept

Richard Bade, Manfred Deutscher-Tiemann and André Herms
University of Magdeburg
Institute for Distributed Systems
Uniplatz 2
39106 Magdeburg, Germany
Email: {ribade,deutscher,aherms}@ivs.cs.uni-magdeburg.de

Abstract— **This paper describes the module loader concept. We successfully used this for the development of system software for our mobile robots. The module loader allows for organizing the software in modules. These can be loaded and removed at runtime. We describe how this is done. Additionally we give some examples on how to use the concept. Exemplified on our robot platforms we describe how to design and implement a modular system architecture.**

I. Introduction

Developing software for robots consists of writing code for sensors, actuators and the controlling system. Commonly used architectures are microcontrollers for sensors, actuators and low level controlling parts communicating with a PC running high level tasks.

Software development for these microcontrollers is normally a time consuming job. Commonly, the actual coding is done in *C* by using the trial-and-error method. But in most cases it is the only possible way. There is also the problem that there is no efficient method to load the compiled software to the microcontroller. Even a small change in the source code requires complete compiling and uploading of the software. Therefore, the development process is very extensive. You need a system, which should be modular, flexible, easy to use and efficient to handle microcontroller software. These requirements result in the development of our module loader.

We are currently using it in a productive environment to develop and deploy system software for our various mobile robot platforms. It allows us to test new software components faster, which results in a more efficient development process. Additionally we got a more flexible runtime environment.

Our paper is organized as follows. In Chapter II we explain the hardware the mobile robot platforms consists of and the software we are running on it. Chapter III describes the module loader concept in detail – the structure of a module, the design, the components, and the loading process. Chapter IV gives ideas and examples for applicational use and in the next chapter we present our developed modules. In the last two parts we compare our module loader to other concepts and afterwards we conclude.

II. System Architecture

In this paper we concentrate on two mobile robots we use for research and education: The six-legged walking robot LAURON [1] and the wheeled robot KURT2 [2]. Although they have different fields of operation, they base on a similar hardware platform. Basically it consists of a PC104 and one or more Siemens C167 microcontrollers [3], interconnected via a CAN bus [4]. The PC runs high level tasks. The C167 microcontrollers are used to communicate with the hardware devices – sensors and actuators (see figure 1). We are using different sets of sensors for both robots, e.g. infrared, ultrasonic, force and inclination sensors. The actuators in the KURT2 are two motors for driving, for the LAURON there are 21 motors in sum - three for every leg, and three for the camera head.

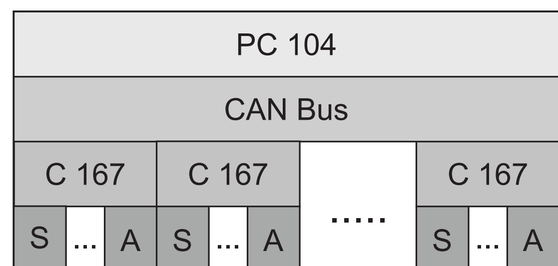

Fig. 1. architecture of the mobile robot systems

The LAURON contains in all seven microcontrollers. To enable communication with the PC and between the microcontrollers themselves they are identified by a unique number. These IDs are automatically assigned once the system is started by a program running on the PC. As the KURT2 has just one microcontroller the assignment process is reduced to simply assigning a fixed number to it.

At the moment the microcontrollers do not run an operating system. To allow further extensions, we support the option to run *PXROS* [5]. The PC is running Linux. As a result we have a two level system architecture: the Siemens C167 microcontrollers for low level tasks and the PC104 system for high level tasks and the application software.

III. Module Loader

In the following we describe the module loader in detail. Our implementation is specific for the C167. But the concept can be applied to other processors, e.g. DSPs, without any

difficulties.

First we define the notion of a module in our system. Afterwards we discuss design goals and their results. Then we explain the functionality of the components of the module loader.

A. Structure of a Module

We use the following definition:

> *module:* A program unit that is discrete and identifiable with respect to compiling, combining with other modules, and loading.

Here we use the module definition for a piece of binary code resulting from a compiled set of *C++* functions. We will explain how this fulfils our module definition. It is possible to compile the functions independently, which results in a compiler object file. The modules can be loaded and combined independently. How this is achieved will be explained in the following sections.

The module must be in a special structure. It can be placed at an arbitrary position in the memory. Therefore, it must be *relocatable*. This means the addresses in the compiled code must not be fixed. The module must contain information for setting this addresses when the final position of the module is known. Fortunately, we already have such a format: the compiler object files mentioned above. The addresses are not fixed after compilation. The linker modifies them when creating the final binary. So the object files already fulfill all our needs.

B. Design Goals

The most important aspect in the design is the asymmetric relation between PC and microcontroller. Compared to the C167 the PC system has enough CPU power and memory for handling most tasks. Therefore, the PC should be used to do the most expensive operations.

As a further requirement, modules should be loaded and removed at runtime. While the robot is still executing some system critical tasks, the system task must not be stopped. Furthermore, the loading process must not block the system for times longer than a few microseconds. Otherwise some controller functions could fail its timing assertions.

A high level of system independence is an additional requirement. This enables us to change the operating system (e.g. to PXROS) on the microcontroller or to work on the pure hardware without modifying the module loader.

C. Design Results

The design goals result in the following properties of the module loader:

1) External Memory Management: As explained, the microcontroller has no system software running on it. Therefore, there is no memory management. But modules are loaded dynamically, which requires dynamic management of memory used by them. As we do not want to burden the microcontroller with this, we use an *external memory management* on the PC system. So the memories of all client microcontrollers are managed by the PC. It maintains a list of all free memory and knows about the memory used by the modules.

When loading a module, memory management is asked for a suitable piece of memory. After loading, the memory area is marked as used by the module. This all happens without intervention of the microcontroller. For unloading the memory belonging to the module is marked as unused.

2) External Dynamic Linking: As described above a module is a compiler object file. Normally this is linked with others to form the application. As we want to add it at runtime the linking is done while the program is already running. This is well known from the concept of dynamic linked libraries. But in contrast to our concept the code is loaded by the application itself. The code is already on the same system as the application. Here the module is inserted and removed from the outside – from the PC.

Common to both methods is that the addresses in the code have to always be corrected to reflect the final memory location of the module. The *relocation process* is responsible for this. With dynamic linked libraries the code is relocated by the application. Here the modules are not processed by the microcontroller as this would cost too much time and could block the system. So this is done externally on the PC. After allocating memory for the module the offset for all addresses is adjusted. After that the relocated code can be transferred to the microcontroller.

With the method described so far, code can only be loaded. But it cannot interact with other modules. For this, it should be able to call functions of other modules. This is possible due to the structure of the object files. If a function in an object file accesses a function or variable outside, the compiler generates an external reference. The address of the call and the name of the function is stored in the file. As the address is known from earlier loading, the code can be updated. This is exactly what the module loader does. It holds a symbol table for every microcontrollers, which contains all names and addresses used in the loaded modules. If an external reference is found, it is looked up and the code is updated. With this method a new module can use all variables and functions provided by modules loaded before.

3) CAN-Hook: The module loader needs some piece of software running on the microcontroller that does the actual module loading. As we use no operating system, we cannot rely on our current CAN driver interface. To make it work with arbitrary drivers, we had to make the interface as portable as possible. This requirement results in an very simple interface. It consists of a single function only: `loader_do(char *data)`. If a CAN packet is received, this function is called with a pointer to the data. The packet is processed and an answer is generated. This answer is stored in the buffer given as argument. The CAN driver only has to deliver this packet and wait for the next one arriving. This way it is really easy to use any kind of CAN driver.

D. PC Part

1) User Interface: On Linux and RT-Linux exists the Linux module loader. It performs nearly the same actions as our loader. For convenience we made our loader behave nearly the same as the already used one. The Linux loader uses the commands `insmod` and `rmmod`. The version for handling modules for the C167 systems are called `insmod166` and `rmmod166`.

A call of the function looks like this:

```
insmod166 -t 7 mymodule.o param1=0x1243
```

In this example the module `mymodule.o` is uploaded to the microcontroller with the number seven. An optional module parameter with the name `param1` is set to the value `0x1234`.

2) Parameters: When writing code like control algorithms, most time is spent with tuning parameters. These are often implemented as constants in the module code. Normally it has to be recompiled when changing the value. We provide a smarter way of doing this. A module can have parameters, which can be set when uploading. In the program code they appear as normal variables. In fact they are. For defining a parameter only a global variable with the same name has to be defined. Executing `insmod` with an option *name*=*value* will set the variable to the given value. This allows for a very efficient development.

3) Special Functions: Most modules require some initialization after the upload. This is done by a special function `init_module()` which returns an integer. If found inside the module, it is called after the upload. A return value of zero reflects that the initialization succeeded. Otherwise the loading of the module is canceled.

A similar function exists for unloading a module. Before removing the code, it must be assured that it is not in use anymore. This is done by the function `remove_module()`. It should try to stop the module. A return value of zero reflects that this succeeded. Otherwise the module cannot be removed.

4) The Complete Loading and Unloading Process: The loading of a module consists of the following steps:

- The size of the module is detected. A memory area big enough to hold the module is reserved. This results in a memory offset, specifying the start of the memory area.
- The addresses in the module are relocated according to the determined offset.
- The binary image is transferred to the microcontroller.
- If any parameters are given, the corresponding values are adjusted.
- If a function `init_module` exists, it is called on the microcontroller. The return value is transmitted to the PC.
- If the return value reflects an error, the allocated memory is freed.
- The symbols of the module are merged with the symbol table of the microcontroller.

The unloading is done as follows:

- If the module contains a function `cleanup_module`, it is called.
- If the return value reflects an error, the unloading process is stopped.
- The symbols defined by the module are removed from the symbol table.
- The memory of the module is marked as unused.

E. Microcontroller Part

The microcontroller runs a part of the module loader system that is responsible for receiving data and performing the tasks necessary for loading and activating the module. This part was kept as small and efficient as possible. We tried to isolate the smallest set of functionality needed. This results in the following three operations.

1) Store data in memory.
2) Run a function at a given position.
3) Read data from memory.

In fact the last operation is not necessary for the normal operation of the module loader. But it adds some very useful functionality consuming only a few bytes of code. The whole module loader part on the microcontroller consists of only 586 bytes of machine code.

IV. Possible Applications

The most important reason for developing the module loader is to shorten development cycles. But there are other applications that benefit from it.

A. Adapting Controlling Software at Runtime

Different fields of application need different properties of the controlling software. For example consider a walking robot which has to go to a wall and drill a hole there [6]. This includes two different tasks – the dynamic walking and the static holding of the tool. Both tasks require different kinds of controller software to achieve optimal functionality. One possibility would be to store two controller programs on the microcontroller and switch between them. But for more tasks this becomes cumbersome. It leads too many distinctions of cases. The structure becomes more and more unmanageable.

The module loader provides a smarter way of doing this. It allows to replace the whole controller software at runtime. This results in the option to have a single controller for every task. The different controller types remain independent. They can be switched by unloading and loading of the corresponding modules. In our example, there would be one controller system for walking and one for drilling. First the robot will load the modules for walking and use them for reaching the wall. After that, these modules are unloaded and the modules for the drilling controller would be loaded. They provide the static behavior, which the walking controller cannot.

The advantage for the developer is that he can design the controller for every task independently, minimizing the complexity. This method can also be applied to other parts of the system.

B. Overlay Technique

An important aspect when writing software for the microcontrollers is the very limited amount of memory available. This does not only limit the size of data structures used but also the code size. One possible way to overcome this is the overlay technique. This was a common way of writing software for 8-bit processors. Commonly only small amounts of the code are actively used. There exist many functions that are only responsible for the initialization of the hardware or data structures. Normally they run only once at startup. So the code can be removed after execution to reuse the memory e.g. for data structures.

We used this approach for some initialization routines. They are running at startup for calibrating the legs of the walking robot (see section V – *joint calibration module*). The code is rather complex and in all it requires 12 kByte of RAM. As we only have 48 kByte of free RAM, this is significant. The code is a separate module that is loaded, executed and removed. This method could be applied too many parts of the controlling software.

C. System Monitoring

In section III-E we mentioned an extra operation for reading data from the microcontroller remotely. This operation was added because it allows for additional functionality. Together with the symbol table, we can read values of variables without influencing the normal program execution. This way the system can be observed and easily debugged.

D. Simple Remote Procedure Calls

The interface of the microcontroller gives us a method of calling functions at a custom location. This is used for calling `init_module` and `cleanup_module`. But it can also be used for other functions. The only restriction is that they cannot have parameters and must return an integer. These restrictions can be abolished by various means. Complex return values may be stored in global variables that can be read by the system monitoring (section IV-C). The value of variables can be changed in a similar way. This allows us to store parameters in variables.

This interface provides a functionality similar to other remote procedure calls. But it is not as transparent as others. In return, no additional code is required on the microcontroller.

V. Evaluation

As we have seen in the previous chapter the module loader can be used for several scenarios. Now we explain various modules that we have developed when applying the module loader concept. These are our testbed to evaluate the feasibility and applicability of our concept to real-world examples.

As mentioned before we have two similar mobile robot platforms. Therefore, it is useful to design modules that can be used on both ones. So it is necessary to figure out what they have in common. First, both robot systems have a CAN bus. For using it we need a module – the *can driver module* – providing the CAN interface. Furthermore, the realization of the output is implemented similar, too. The *output stream module* provides the functionality to send messages from the C167 to the PC. Our main use is for debugging purposes. As we need fractional numbers for some tasks but the C167 has no floating point unit, we had to achieve it in an other way. First we tried to use some floating point modules. But they use 25 kByte RAM in total. We had too few memory left to get other tasks running. Furthermore, it was too expensive due to the emulating algorithms. So we introduced the *fix point module*, which encapsulates corresponding arithmetic functions. Overall, it is more efficient than the floating point emulation and uses only 6 kByte of RAM.

At this point the implementation is very hardware specific. As you can see in figure 2 there are modules for both robot platforms.

We will describe the software modules for the mobile robot platform LAURON in detail. First there exists a module – the *Lauron board module* – which provides control of the basic hardware the C167 is connected to. This includes the control of the A/D converter and the shaft encoders but also the control of the joint and camera head motors. Furthermore, we developed modules for controlling the LAURON. First of all, there is the *joint calibration module*. It is responsible for calibrating the leg joints. After calibration, the *joint calibration module* can be removed as described in section IV-B. The calibration results are stored in the *Lauron board module*. Now it is possible to control the legs with the help of the *leg control module*. This module uses the *joint control module* and the *kinematics module* to move the joints of one leg according to its foot position. The gait controller on the PC sends to the C167 units how they should move the legs. The *leg control interface module* ensures that the instructions are available for the *leg control module*.

The *kinematics module* has to do computations with high precision. Therefore, it uses the *fix point module*. Furthermore, there exists a *camera head control module*, which is used to control the bias and the viewing direction of the camera head, as well as a *force sensor module* and an *ultrasonic sensor module*, which get data from the corresponding sensors.

All these modules, from the LAURON as well as from the KURT2 , are independently exchangeable. We first implemented a simple joint controller for the LAURON because we wanted to get the robot walking as soon as possible. By now we use a more complex joint controller to move the legs more precisely. Therefore, just the *joint control module* had to be changed. Also the development of an enhanced gait controller on the PC introduced no problem. We only had to replace the *leg control module* and the *leg control interface module* by new ones.

Our experience with the module loader so far showed that using it encourages a structured design, because the developer is forced to implement one set of functionality as one *module*.

VI. Related Work

Modular design is well known in software development. There are many concepts and programming languages realiz-

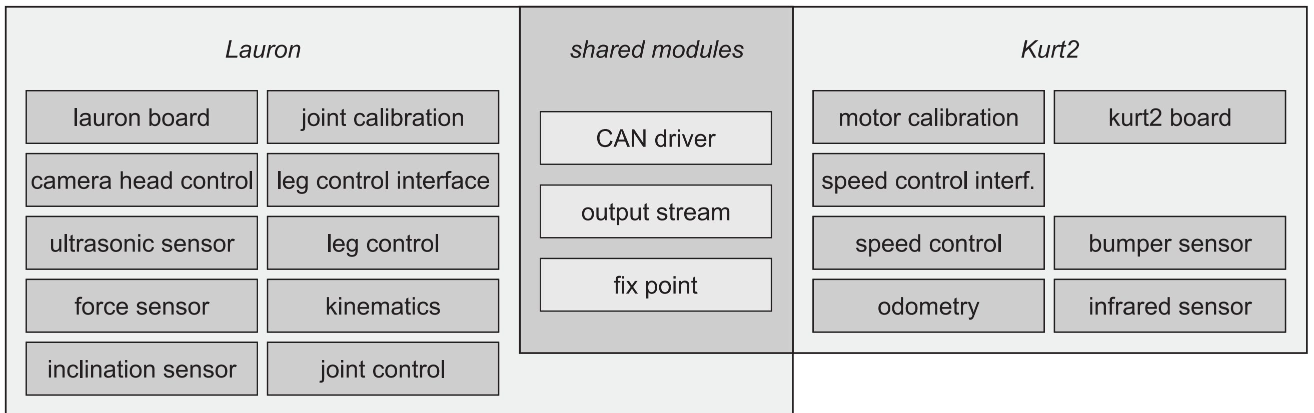

Fig. 2. overview of the developed modules for LAURON and KURT2

ing it. Anyway, this mostly results in modules at source code level. Their goal is a better software design. On the other hand, we concentrate on flexible handling of already written code. Especially in the context of robot programming such systems seems to be rarely used.

A. Modular Controller Architecture

The *Modular Controller Architecture* MCA *is a modular, network transparent and realtime capable C/C++ framework for controlling robots and other kind of hardware* [7], [8]. It has been used to develop the controller software for several mobile robots like the LAURON. The system consists of modules that are connected via a unified interface. This allows a very restricted but clean way of implementing components. Unfortunately, modules exists only at source code level – represented by *C++* classes. Compilation results in a monolithic binary that is loaded as a whole. We considered to use it for our development, but the system was too static and inflexible for our needs.

B. Linux Module Loader

The most similar system is the module loader of the Linux operating system [9]. It is normally used for loading device drivers. We tried to mimic the user interface and functionality. The main difference is that the Linux operating system loads the modules itself, while we do it from an external system.

C. Overlay Technique

The overlay technique was a common way of making large programs fit into small memory. Especially in the era of 8-bit computers, this was commonly used. Normally it is not needed anymore for typical personal computers. Overlays allow different parts of the program to share the same memory space. Only the overlay that is currently in use must be loaded. The others remain on disk and are loaded as needed. Programming overlays can be a difficult task. Only a few compilers (like [10]) support them directly. So assembler programming might be necessary. The loading must also be triggered manually inside of the code.

The overlay concept as described for the module loader also aims to save memory used by program code. But the overlay process is different as it is done externally by the module loader.

D. Remote Debugging

The functionality of the module loader is similar to the remote interface of debuggers like the GDB [11]. It can be used to load binary code to a client system, and it also has functions for setting and reading variable values. The difference is that the code can only be loaded at once. The interface is not as efficient as it depends on a plain text protocol via serial interface [12]. We first considered to use this interface for our programming. But it was too slow and not flexible enough. It also has additional unused functionality like setting breakpoints. This costs additional memory we cannot spare.

E. JTAG Interface

A possible alternative to the module loader is the JTAG Interface [13]. The JTAG Interface standard was defined in 1990 by the Joint Test Action Group. It allows testing and controlling of a microprocessor via a five-pin serial interface. Motorola invented a similar method – the Background Debugging Mode (BDM) [14]. But it is available for their own microprocessors only. With these methods it is possible to upload binary code to the processor, but it is more complex as you have to write directly to the processor registers. This leads to the problem that the implementation of such a *software loader* is hardware specific. Another disadvantage is the speed of the interface. It is limited to the speed of the serial bus. This is too slow for our needs.

VII. CONCLUSION

In this paper, we describe the concept and use of our module loader. It allows us to handle the controller software of our robots in a efficient and flexible way. The software of the microcontrollers consist of several discrete modules. These can be loaded at runtime. We describe how the module loader achieves this. Furthermore, we give examples on how the

resulting functionality can be practically used. The modules used on our robots are also described. This concept can be applied to other systems, too. They could also benefit from this.

REFERENCES

[1] B. Gaßmann, K.-U. Scholl, and K. Berns, "Locomotion of Lauron III in Rough Terrain," in *Int. Conference on Advanced Mechatronics*, July 2001, como, Italy.

[2] R. Worst and F. Kirchner, "KURT2 - Eine mobile Plattform für die Robotikforschung," in *Robotik 2002: Leistungsstand, Anwendungen, Visionen, Trends*. VDI-Verlag, June 2002, pp. 389–394, iSBN 3-18-091679-6.

[3] *C167CR User's Manual, 16-Bit Single-Chip Microcontroller*, Infineon Technologies AG, 2000, munic, Germany.

[4] *CAN Specification Version 2.0*, Robert Bosch GmbH, 1991, stuttgart, Germany.

[5] PXROS, "Project Homepage," June 2004, http://www.hightec-rt.com/pxros.html.

[6] T. Ihme, "Posture control and distributed force sensing for technical applications of walking robots," in *Proceedings of the 11th Conference on Advanced Robotics, ICAR 2003*, vol. 2, Jun 30 – Jul 3 2003, pp. 1032–1037, ISBN: 972-96889-8-2.

[7] Modular Controller Architectur Version 2, "Project Homepage," http://mca2.sourceforge.net/.

[8] K.-U. Scholl, J. Albinez, and B. Gassmann, "MCA - an expandable modular controller architecture," in *3rd Real-Time Linux Workshop*, 2001, milano, Italy.

[9] M. Beck *et al.*, *Linux-Kernel-Programmierung*, 2nd ed. Addison-Wesley, 1994.

[10] Homepage - Borland developer network, *Antique Software: Turbo Pascal v3.02*, jun 2004, http://bdn.borland.com/museum/.

[11] *Homepage – GDB: The GNU Project Debugger*, http://www.gnu.org/software/gdb/gdb.html.

[12] B. Gatliff, "Implementing a Remote Debugging Agent for GDB," http://billgatliff.com/articles/gnu/gdb-agent.pdf, Tech. Rep., May 2004.

[13] Joint Test Action Group (JTAG), *IEEE Standard Test Access Port and Boundary-Scan Architecture*, Institute of Electrical and Electronic Engineers (IEEE), New York, Mar 21 1990, IEEE Std. 1149.1-1990.

[14] S. Howard, *A Background Debugging Mode Driver Package for Modular Microcontrollers*, Motorola, Motorola Literature Distribution, doc. no.: AN1230/D.