Diplomarbeit

Run-Time Task Reconfiguration on Multi Digital Signal Processor Systems

Oliver Gerler

Institut für Technische Informatik
der Technischen Universität Graz

Begutachter: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Betreuer: Univ.-Ass. Dipl.-Ing. Martin Schmid

Graz, Bath, Hanau, Alpbach, Shinrone, Juni-August 2001

Abstract

In this Master's thesis a run-time reconfiguration mechanism for multi-DSP systems with point-to-point communication has been developed to improve testability and maintainability. This mechanism is an extension of the rapid prototyping system PEPSY which allows the development of data-flow driven real-time applications. First, existing rapid prototyping systems, data-flow models and multi-DSP operating systems are investigated and compared. Principles of fault tolerance, test and reconfiguration are outlined. Second, an overview of PEPSY is given and the basic functions of the run-time reconfiguration mechanism are discussed. Subsequently, implementational details of the mechanism and the data transfer protocol between the host and the multi-DSP system are shown. Finally, in an experimental evaluation the quality and temporal behavior of the implementation are investigated.

Zusammenfassung

In dieser Arbeit wurde ein Rekonfigurations-Mechanismus für Multi-DSP Systeme mit Punkt-zu-Punkt Verbindungen entwickelt, um Testbarkeit und Wartbarkeit darin zu verbessern. Dieser Mechanismus stellt eine Erweiterung des Rapid Prototyping Systems PEPSY dar, das die Entwicklung von Datenfluß-orientierten Echtzeit-Anwendungen ermöglicht. Zuerst werden existierende Rapid Prototyping Systeme, Datenfluß-Modelle und Multi-DSP Betriebssysteme erörtert und verglichen. Anschließend werden Grundlagen in den Bereichen Fehlertoleranz, Test und Rekonfiguration erläutert. Im zweiten Schritt wird PEPSY vorgestellt und grundlegende Anforderungen an den Laufzeit-Rekonfigurationsmechanismus werden erörtert. Darauf folgend werden Details der Implementierung und des Datentransfer-Protokolls zwischen Host und Multi-DSP System gezeigt. Abschließend wird die Qualität und das zeitliche Vehalten des Lademechanismus experimentell untersucht.

Acknowledgements

Many thanks go to my tutor Martin Schmid for enlightening hints and quick answers. I would also very much like to thank Milena for continued support and her repeatedly proven ability to bother me in precisely the right moments. My parents I would like to thank for the opportunity to do my degree at all.

And in the "No thanks" department: the architects responsible for the buildings, for the many beautiful sunsets I could not experience due to these masses of utterly ugly concrete walls. The tiny patches of sky do not count.

Contents

List of Figures iii

List of Tables iv

Chapter 1 Introduction 1

Chapter 2 Theoretical Background 3

2.1 Overview . . . 3
2.2 Rapid Software Prototyping on Multi-DSP Systems . . . 4
2.2.1 Principles and Development Environments . . . 4
2.2.2 Data-flow Models . . . 6
2.3 Existing DSP Operating Systems . . . 8
2.4 Fault Tolerance . . . 11
2.4.1 Principles . . . 11
2.4.2 Software Fault Tolerance . . . 14
2.4.3 Implementational Issues . . . 15
2.5 Reconfigurability . . . 16

Chapter 3 Principles of Operation 19

3.1 Overview . . . 19
3.2 Data-flow model and code requirements . . . 21
3.3 Reconfiguration Mechanism . . . 26
3.3.1 The Kernel . . . 26
3.3.2 The Host Tool . . . 27
3.3.3 Communication Protocol . . . 28

Chapter 4 Implementation 30

4.1 Target System . . . 30
4.2 Protocol Commands . . . 32

4.2.1 0x00000000 - request status 34
4.2.2 0x00000001 - initialize download 35
4.2.3 0x00000002 - download packet 37
4.2.4 0x00000003 - start task 38
4.2.5 0x00000004 - stop/kill task 39
4.2.6 0x00000005 - send buffer to host 40
4.2.7 0x00000006 - set mode 41
4.2.8 0x55555555 - error packet 41
4.3 The Kernel 42
4.4 The Host tool 45
4.5 Expansion to PEPSY 47
4.6 Writing New Tasks 49
4.6.1 Development 49
4.6.2 Restrictions 50

Chapter 5 Experimental Evaluation and Results 53
5.1 Performance 53
5.2 Scalability 56
5.3 Fault Tolerance 56

Chapter 6 Discussion 58
6.1 Conclusion 58
6.2 Further work 60

Appendix A Acronyms 64

Appendix B Terms 65

Appendix C Kernel Code Listing 68

List of Figures

2.1 Development phases of rapid prototyping systems 5
2.2 Phases of system lifetime 12

3.1 Design flow of PEPSY 20
3.2 Data-flow in the original task model 22
3.3 Execution order in the original task model 22
3.4 Example task Prototype 23
3.5 Example executive source 25
3.6 Tree structure of DSP connections 27
3.7 Mechanism of command execution 28

4.1 Block diagram of Transtech TDMB412 30
4.2 Connection between Transtech board and ADAC card 31
4.3 Structure of command word 33
4.4 Addressing of DSPs 34
4.5 Flow chart of the DSP kernel 43
4.6 Argument template of host tool 46
4.7 Data-flow in extended task model 48
4.8 Execution order in extended task model 48

5.1 Code sequence for test on presence of commands from host . . 54

List of Tables

4.1 Commands used in DSP-host communication 33
4.2 request status: data from host to DSP 35
4.3 request status: data from DSP to host 35
4.4 initialize download: data from host to DSP 36
4.5 initialize download: data from DSP to host 36
4.6 download packet: data from host to DSP 37
4.7 download packet: data from DSP to host 37
4.8 start task: data from host to DSP 38
4.9 start task: data from DSP to host 38
4.10 stop or kill task: data from host to DSP 39
4.11 stop or kill task: data from DSP to host 39
4.12 send buffer to host: data from host to DSP 40
4.13 send buffer to host: data from DSP to host 40
4.14 set mode: data from host to DSP 41
4.15 set mode: data from DSP to host 41
4.16 error packet: data from DSP to host 42

Chapter 1

Introduction

As society increasingly relies on technology in everyday life, it becomes more and more important to provide reliable systems with high availability, ease of maintenance and the possibility for reconfiguration and (re-) distribution of the application software. This thesis will therefore provide means, based on a multi Digital Signal Processor (DSP) system, to provide all of the above with additional features like in-system test of tasks, incorporation of fault-tolerance principles and administrative tools to control the (embedded) multi-DSP system at run-time by means of a connected host computer. The main advantage of having the ability to reconfigure tasks at run-time is the fact that high-available systems can remain up and running while an update of parts of their software is conducted.

The aim of this thesis will therefore be to implement such a mechanism. Special consideration is given to low-overhead and system stability. This implementation will show the principles of work of such a system and will be compatible to already working software. A rapid prototyping system called PEPSY exists at the Institute for Technical Informatics at Graz University of Technology, which consists of a host PC and multi-DSP boards. This system[1] has been used for various tasks like simulation of the human peripheral auditory system developed by rapid software prototyping using Java classes [Rup01] or simulated annealing [Sch98]. It will be reused for evaluating and implementing methods and protocols to allow for reconfiguration of tasks at run-time under special consideration of low overhead and fault tolerance. This is to accomplish a broad spectrum of executable tasks and algorithms on systems with small footprints.

[1]see chap. 4.1 (p. 30).

Applications based on PEPSY are data-flow driven. The DSPs in the multi-DSP system work as nodes receiving and sending data to other DSPs in the system in a peer-to-peer network. That means, that connections between DSPs only influence the two DSPs connected via that specific physical link.

In the ever evolving area of fast computing and parallel processing a secure and overhead-minimizing way to incorporate task reconfiguration into embedded DSP systems provides smaller turn around times and a quicker time to a working product. This holds because there is no more need to reboot a system once a new version of software has been written. A small kernel on the DSP will be needed therefore to accomplish the task of downloading the code from the host and supervising the execution of it.

This kernel not only has to start the relevant code by jumping to its entry point but also has to take care of real-time constraints and must for instance therefore be able to kill tasks exceeding their initial execution time. It would also be of use to implement some sort of memory protection into the DSP kernel, but this will be left out for further work to be done.

In the course of this thesis methods will be developed to describe the processes and theoretical aspects of the implementation. The existing implementation of rapid software prototyping by tools for automatic code generation shall be expanded to incorporate the necessary changes for an automatic generation of the mini kernel itself. The kernel will then be tailored for the target system and the problem to be solved.

The main aims therefore are for the system to enable task reconfiguration, and with it provide means to enhance the system with for example fault tolerance principles and the ability to recover errors, the possibility to implement N-version programming on a single computing device, updating of DSP firmware, software testing and alike.

The remainder of this thesis is organized as follows: In the next chapter an overview over rapid prototyping systems and existing DSP operating systems (OSs) will be given, and there will be a description of data flow models for DSP applications. Second, an overview of PEPSY is given and the basic functions of the run-time reconfiguration mechanism are discussed. Subsequently, implementational details of the mechanism and the data transfer protocol between the host and the multi-DSP system are shown. Finally, in an experimental evaluation the quality and temporal behavior of the implementation are investigated.

Chapter 2

Theoretical Background

2.1 Overview

The aim of this thesis is to implement a tool set to enable developers of single- and multi-DSP systems to reuse RAM and DSP resources while the system is running. Although re-using resources is not to be confused with reconfiguration of hardware,[1] many concepts from that field can be applied to develop a DSP system which provides for high flexibility, little overhead and a small footprint.

The challenge is to incorporate the design goals into an already existing system, in this case to introduce the capability to load code and data into a multi-DSP system. Two main concepts have to be considered and eventually implemented in this thesis. Rapid prototyping as a concept is given due to previous work and the existing tools and concepts will be incorporated herein. Software fault tolerance principles have to provide for secure operation in different environments and is therefore a basic requirement for reliable systems as hardware failures can never be excluded. In the next two sections, these two concepts will be discussed more closely, together with an evaluation of existing DSP real-time OSs.

[1] The concept of reconfiguring hardware is nothing new. In fact, one of the first computers, Colossus by Alan Turing, could only be programmed by reconfiguring its hardware. See `http://slashdot.org/books/01/01/20/1337235.shtml`.

2.2 Rapid Software Prototyping on Multi-DSP Systems

2.2.1 Principles and Development Environments

Rapid software prototyping and development describes a process that helps developers to implement their intended applications in a faster and smoother way. It is supposed to minimize turn-around times and time-to-market. As product lifetime increasingly becomes shorter and shorter it is important to simplify the development process. Therefore abstractions layers are introduced to modularize development phases. This can be accomplished either by using OSs with powerful APIs or with automatic code generation tools, both introducing layers of abstraction between the target system and the developer. Powerful APIs are useful for functional implementations to proof intended concepts. As they are capsuled and provide defined interfaces, it can happen that similar code is introduced twice or more often into the project resulting in increased memory requirements. Also synergy effects by using pre-loaded registers mostly cannot be utilized as the interface of the API functions prevents use of internal register usage. Because of the attributes depicted above, automatic code generation tools are used in this thesis.

Rapid software prototyping, or rapid prototyping in general, introduces layers of abstraction between the problem at hand and the target platform. Development with rapid software prototyping tools can be divided into several phases as shown in fig. 2.1 (p. 5). First the application to be implemented has to be modelled to fit into the description of the system. Also the hardware abstraction has to be in a format compatible to a mapping tool, which will then generate the mappings and an optimized schedule. Code synthesis tools generate executable code, which can be downloaded onto the DSPs and tested. If there are bugs they have to be removed in the abstract model and the complete process applied again.

One such system, Grape-II, a graphical rapid prototyping environment, provides the developer with a GUI to apparently ease the development process due to abstraction. Its main advantage over specialized systems is the employment of "general-purpose reusable hardware to minimize development cost and a structured prototyping methodology to reduce programming effort" [LEAP95]. In that it releases the developer from tedious implementational issues and they can concentrate on problem specific topics. Grape-II

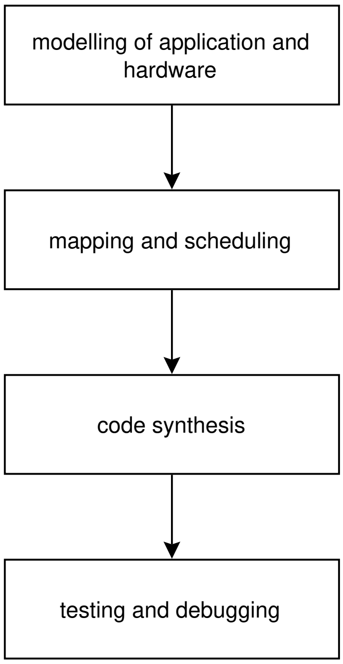

Figure 2.1: Development phases of rapid prototyping systems

is able to modify parameters stored in the DSP RAM at run-time, but there is no provision for incorporating new tasks into the system once it is started.

Another system, SynDEx, also abstracts the hardware from the developer to increase development speed and reduce turn-around times. “The implementation consists of distributing and scheduling the data-flow graph on the multicomponent hyper graph while satisfying real-time constraints. The distribution and schedule are calculated first and are static. SynDEx represents the distributed processing tasks into the specified processors and then, generates an intermediate macro-code, which is a direct translation of the obtained distribution and scheduling. Finally, a macro processor M4 generates the appropriated executive for the target architecture" [FAdar]. The ability to autonomously map tasks and data onto parallel and mixed architectures is implemented as well as in the other systems described above [FAdar, FdAar].

Also the development environment developed and provided by Texas Instruments, Code Composer, can be seen as a rapid prototyping system as its integrated multi-DSP capabilities extremely simplify development for such systems. It is able to intensively monitor the DSPs at run-time via a JTAG connection to the DSPs. With this it is possible to inspect and change memory locations and CPU registers at run-time. Integrated debugging facilities and trace modes provide further simplification and acceleration of the development process.

The Code Composer provides high integration and abstraction of hardware and presents the user with an useable frontend to its integrated development envirnoment.

A fourth system, PEPSY, developed at the Institute for Technical Informatics, Graz University of Technology, provides another set of tools to mainly release the developer from distributing the tasks and data by themselves as it utilizes the resources in an optimal way, gained by simulated annealing. Given an extended data flow graph of the DSP application and a description of the target multi-processor system, PEPSY automatically maps and schedules the DSP application onto the multi-processor system and generates complete code for each processor. An expansion exists to automatically execute partitioning of the given problem with the help of costs assigned to implementations on different target hardware as well as in software [Müc01].

2.2.2 Data-flow Models

Nowadays two computing principles are generally at work: von Neumann architecture and the data-flow concept. As von Neumann computers are mainly for serialized execution of commands, data-flow provides for parallelizing and will therefore be preferred.[2]

Data flow is usually represented in graphs. Here edges depict the flow of data between tasks denoted by the nodes of the graph. Several data-flow models can be distinguished, based on the amount of data (relative to other inputs and outputs) to process in each step and its synchronization methods. Depending on how the consumption and production together with the firing rules are specified, graphs or subgraphs can be divided into different classes:[3]

- single-rate (or homogeneous) data-flow (SRDF)
- multi-rate (or synchronous, regular) data-flow (SDF, MRDF)

[2] For a historical overview on data-flow and von Neumann computing, see [ŠRU97].

[3] A description of the most important data-flow models as well as the development environments can be found in app. B (p. 65) and [LEAP95, BELP96, FAdar]

- cyclo-static data-flow (CSDF)
- dynamic data-flow (DDF)

Systems using SRDF produce exactly one output unit for each data unit at their inputs. The data rates at inputs and outputs are therefore always the same and constant. MRDF systems have a fixed relation between the amount of data they read from each input to output one data unit. This relation is fixed, but the data rates differ from gate to gate. CSDF is (as used in the Grape-II [LEAP95] environment) an extension to MRDF in that in can change the rate of input and output data according to the data itself. Nevertheless, the changes are cyclic and predictable. For SRDF, MRDF and CSDF it is possible to apply static (offline) scheduling for constructing a valid schedule. DDF requires online scheduling and is therefore the most universal but also the most complex data flow concept. Many more data-flow models are discussed in [Bha99] as extensions to SDF to fill the gap between SDF and DDF while maintaining its compile-time predictability. data-flow, MD-SDF; well-behaved stream flow-graphs, WBSFG; BDF; cyclo-dynamic data-flow, CDDF; heterochronous data-combines the SDF paradigm with finite state machine data-flow, BDDF. As will be seen in chap. 3.2 (p. 21) and chap. 4.5 (p. 47), the existing data-flow model uses buffered data transfers and therefore constitutes a (multi-rate) synchronous data-flow model. There is no provision for cyclo-static data-flow, but as the expansion for reloadability is in no way restricted to the contents of the task itself, other models and behavior patterns can be implemented.

However, the scheduling for cyclo-static tasks is more demanding than for the single-rate or even multi-rate data-flow model. "For a proper run-time execution, the schedule must guarantee that all necessary data is available when a task is executed and that the amount of data in the buffers remains nonnegative and bounded" [BELP96]. It can be shown that as the schedule will be executed repetitively, every vertex has to step through a complete execution sequence for each loop and that the amount of tokens produced by an edge during one iteration of the schedule must equal the amount of tokens consumed from it [BELP96]. This sounds trivial but is important to remember while implementing to avoid buffer over- or underflows. Deadlocks would follow and the system would be halted. One approach to avoid deadlocks is to implement circuitry to detect deadlocks and try to re-synchronize all affected vertices. This approach shall be used in this thesis.

To convert a data-flow representation of a model into a running system, the development environment has to assign the vertices to processing devices, define routes for data paths through the network and determine the execution order of the tasks. This can be done dynamically, at run-time, but imposes severe overhead onto the system. Static (offline) scheduling will be preferred, although dynamic data-flow models cannot be used anymore. Therefore, several extensions to the SDF have been proposed to circumvent this restriction of the SDF. But to enable static scheduling at all, only tasks with invariant behavior can be supported in the classical SDF model. CSDF is a more powerful data-flow model that can incorporate cyclically changing behavior. To support reloadable tasks one would have to use DDF, but with certain restrictions SDF or CSDF can also be used and static scheduling is still possible. The restrictions imposed will be further discussed in chap. 4.6.2 (p. 50).

It would of course be possible to implement an online scheduler to support DDF, but the high overhead is unacceptable in resource restricted systems. However, most problems behave predictably in one way or the other and so make it possible to apply extended SDF models, which results in the possibility for static scheduling. Therefore a new data-flow model is proposed: **Quasi-Dynamic Data-Flow**, QDDF, shall incorporate quasi-dynamic behavior while still being statically scheduled offline. "Quasi-dynamic" shall denote the dynamic behavior in that it is not predictable in what state the system is at a given moment, as any well-behaving[4] task or none at all could have been downloaded into the system before. It is therefore not possible to give a schedule beforehand which would be valid at every point in time. The only feasible schedule incorporates all fixed tasks in the system plus additional time left for the kernel and a possible task under test to fit in.

2.3 Existing DSP Operating Systems

The concepts described above are already implemented in several real-time operating-systems, with the exception of task reconfiguration. The question arises, why should one not investigate in available real-time DSP OSs and use an open-source version, which can be tailored to one's need?

[4] The task at hand has to be compiled before downloading and the developer will have taken care not to exceed the limits set by the system.

As a quick answer, they all have too many features and too much overhead for being used as a small kernel, which in this application mainly has to provide means for task reconfiguration. Furthermore open-source versions of DSP OSs are very rare and hard to find. Commercially available real-time OSs, like Virtuoso[5], OS-9[6], QNX RTOS[7], Neutrino[8], Elate[9], DSP-Bios[10] and others[11], are either targeted at the desktop market (Elate or Neutrino, for example, can be run on a desktop PC featuring support for various processors), rarely, or at embedded systems with an arbitrary selection of restrictions on memory, energy consumption, processing power and space. The latter would fit the requirement, but they do not provide the ability for reloading code at runtime (and are not freely available or even open-source, either). Using open-source variants, despite the fact one would have to find one first,[12] would provide for adaptability of needed features and make it possible for experienced developers to include new ones. Since this approach is several layers of abstractions too low if rapid prototyping principles are to be applied and to use already existing tools and concepts, a new mini kernel, written in DSP assembler, is to be developed, which suits the needs exactly and is very low on overhead (in speed as well as in functionality and resource requirements).

Task handling in these OSs is done by a main executive providing several different multitasking scheduling principles. Common to all of them is the requirement of tasks already existing in the system to be schedulable. Out of this set of tasks the one with the highest priority according to the scheduling algorithm (round-robin, first-deadlines-first, etc.) is chosen and being executed. Also interrupts are handled by these DSP-OSs and depending on their interrupt priority each of them can interrupt another interrupt. The usual

[5]URI: `http://www.eonic.com/mainnav.cfm?webcat=2&level1=1&level2=29`

[6]URI: `http://www.microware.com/Products/Software/OS9.html`

[7]URI: `http://www.qnx.com/products/os/rtos6.html`

[8]URI: `http://www.qnx.com/products/realtimeplatform/index.html`

[9]URI: `http://tao-group.com/2/tao/elate/elatefact.pdf`

[10]URI: `http://dspvillage.ti.com/docs/toolssoftwarehome.jhtml`

[11]At `http://www.realtime-info.be/encyc/buyersguide/rtos/Dir228.html` a comprehensive list of RTOSs can be found.

[12]One approach could be RTEMS as utilized in a mixed architecture high-performance real-time system described in [FMF98]. RTEMS is as a real-time kernel embedded into the FRESCO system, an open-source development environment maintained by ESA which consists of ports of `gcc` and `gdb`, published under the GPL. Unfortunately, however, it seems to be abandoned by now, according to the pages at `http://www.estec.esa.nl/wmwww/EME/compilers/Fresco/erc32/fresco.htm`.

design process with these OSs includes generation of task sets and downloading of the complete executable including the executive onto the DSP before starting. The author is not aware of any DSP OSs featuring task reloadability in a quick and low overhead imposing way. These OSs handle tasks and data depending of their state in time rather than in amount of data. Time intervals and points in time define the execution order of the application tasks. In that they are of no real relevance to the dataflow driven executive discussed and expanded in this thesis.

The typical embedded DSP system is developed on simulators running on workstations while the resulting end product runs stand-alone with no connection to a host whatsoever. There is no or little need for a software or firmware change throughout the lifetime of the product if the design phase has been executed properly enough. Mostly, however, problems arise due to the simulation of the systems. As a simulation can never be perfectly accurate, the most common causes for failures on the final target system are incompatible interface timings [BELP96]. In order to circumvent this disadvantage, developing directly on the target hardware is desired. As this usually is very dangerous if the target system is already running and has to stay as available as possible, methods have to be implemented to ensure continued availability even in case of buggy code downloaded onto the hardware.

As distributed computing and parallel processing becomes more and more intriguing and applicable for the ever more networked computers and systems, different approaches have to be investigated. The often-cited networked refrigerator is an example for networked embedded systems where the reloading of tasks could be used to tailor the algorithm to build the list for food orders more to the need of the customer while preserving requirements for a small footprint of the resulting system as the new code is not required to reside at the target system.

The OSs discussed above enable programmers to implement their solutions in a simple albeit static way. Once the system has been configured it can not be changed easily anymore. Maintenance of such systems mostly involves shutting it down, change of firmware and restart of the complete system. In high available systems this behavior cannot be tolerated. Remote maintenance is completely rendered impossible with such a setup, but this would heavily decrease maintenance costs as no technician at site would be necessary anymore. To achieve these testing and maintenance facilities a mechanism has to be developed that implements the following features:

- downloading code onto DSP memory at run-time
- software fault tolerance principles
- rapid prototyping facilities

In this thesis a mechanism has therefore to be developed to provide the user with a dynamic behavior in that the system could change at run-time. Due to the ability to reload code onto DSP memory scheduling could then be changed and even the executive exchanged to a new one to include new application tasks or exclude obsolete ones. With this possibility even (unintendedly) changing hardware could be supported, once the changes can be detected (which can be as simple as timeout or watchdog functions). It is therefore necessary not only to implement the means for loading mechanisms but also additional properties to supervise these code exchanges as these exchanges are a crucial part for the stability and usability of the system. The next chapter, therefore, deals with (software) fault tolerance principles and how they could be implemented to increase overall system stability and make easy maintenance possible.

2.4 Fault Tolerance

2.4.1 Principles

Technical systems, if designed by humans, are not error free, especially if it comes to software in computer systems. There is no way to mathematically prove if software is bug-free.[13] Moreover all the errors in software are design faults and hence are not predictable. Therefore no provisions can be make to work around these faults as long as they are not known. Running software cannot know by itself that it is behaving in an unintended way. On the other hand, increased reliability is required by systems, especially if human lives depend on them. Hence, methods have had to be developed to cope with bugs, errors and faults in general in a way, where continued live support and reliability can be assured. The main point of fault tolerance therefore is, how to continue a service once a fault has occurred.

[13] It is in fact noted by [LYC99], that “Some faults will always remain, and the level of faults in released software are typically between 1 and 10 faults per 1000 lines of codes.”

To mitigate the effects of ill-behaving software or, to put it more generally, controlling entities in systems (which applies to hardware as well), concepts have been developed to allow for faults and recover from the impacts on the system. These concepts can be summarized under the topic of fault tolerance. To be able to decide on which aspects of fault tolerant systems shall be implemented during the work for this thesis, an overview of the theoretical definitions and notions shall be given here.

According to [LA90] and as can be seen in fig. 2.2 (p. 12), system lifetime can be divided into three phases: *design and implementation*, *debugging and testing* and the *in-service*.

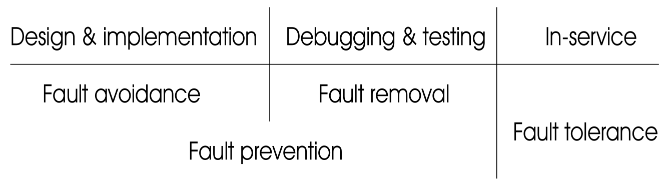

Figure 2.2: Phases of system lifetime

Design and implementation describes the specification phase of a system. Here the overall operation modes are defined. Most of the situations where faults can occur have to be eliminated early in the design. It is always better to search for alternative possibilities to implement a specific problem, than to have to check for too many possible errors later on. Here, *fault avoidance* has to be pursued.

Debugging and testing has to be performed thoroughly to be able remove as many errors and faults, that have been introduced while implementing the system, as possible. It is not always applicable and desirable to strictly divide the testing and implementation phases. As faults are encountered they have to be removed, hence this is also called the *fault removal* phase.

In-Service denotes the working phase of a system. It is running on its final target hardware, fulfilling the tasks it has been developed for. In this phase it is no longer possible to change the system in a simple way. Furthermore it should not be necessary to change it. This is the phase, where fault tolerance can and has to be applied, as in this phase the system has to cope with faults all by itself.

The first two phases can be collected under the topic *fault prevention*,[14] the third is where *fault tolerance* will be applied. Fault prevention has to massively utilize techniques to avoid bugs in systems while still in design phase. One way to do this is to partition the system in smaller and smaller parts which can easily be tested and only compile it afterwards into a system of increasing complexity but with tested modules. This way desired operation of complex systems can be produced much easier and more reliably [Vra96]. For this, the modules of course have to feature defined interfaces and have to be as independent from each other as possible, which in itself hinders application of synergy effects of cross-task register usage only if these are not well-defined. Another possibility (without having to strictly split modules from each other) is the introduction of points of control and observation, as suggested in [Vra96], where test stimuli can be applied and values be read out at a desired point in the system while running. Although this is mainly applied in hardware testing, it is also usable in software at the cost of, for example, interrupt routines reading out or inserting desired values at specific points.

As stated above, software can never be completely bug-free. To be able to handle different types of errors and faults, these have to be classified. A basic work on classification is by Lee and Anderson [LA90]. They describe two basic classes of faults: *anticipated* and *unanticipated* ones.

Anticipated faults are values out of range (typed in by the user for example) or divides by zero, for example, where effective countermeasures can be applied by prompting the user again to input a correct value. Furthermore failures of hardware components can be anticipated as they will eventually fail. Appropriate fault tolerance measures can be implemented into systems to detect when hardware failures occur and ensure continued service by, for example, switching operation to a spare unit.

Unanticipated faults are design faults, as the effects of such faults will be un-anticipatable. What is more, the only type of fault that exists in

[14] Also called “fault intolerance” occasionally [AK83].

software, is a design fault, as software does not degrade over time, as hardware does.[15]

2.4.2 Software Fault Tolerance

Fault tolerance is mainly applied for hardware faults [LA90, p. 205] [LGH80], but techniques have been developed to cope with software (design) faults. Two major concepts are usage of a *recovery block* and *n-version programming.* These concepts are not implemented into the system developed during this thesis, but as reloading of code is provided, n-version programming can easily be supported.

Recovery block scheme provides for automatic backward error recovery implicit by the mechanism, error detection by the acceptance test and fault treatment through the use of alternate modules. As backward error recovery is applied, no damage assessment is necessary. The scheme is implemented by establishing a recovery point prior to the call of a task and testing the output afterwards for acceptance. In the case of failure, the system will backtrack to the recovery point and start another, secondary, module with the same data provided on the inputs. This can be repeated as often as there are different modules available for this task to be computed. Also nesting of this scheme is possible.

N**-version programming** describes a process where routines are implemented not only once, but independently from each other in n versions. All of them are executed with the same set of data and their output compared to each other in a sort of replication check. Based on a majority vote, this check can then eliminate erroneous results. The difference to the recovery block scheme is, that every available module will be executed in every case and not if one of them fails. The advantage being constant execution time in all cases, even though it is longer in overall for most cases. The problem however exists, on how to decide on the correct result if the outputs are principally all correct but differ, for example because of different rounding of numbers or usage of different approximations.

[15] Although there have been investigations by Intel into 'bit decay' (or 'bit rot') due to cosmic rays, single-bit errors are mostly caused by design failures in RAM modules. See `http://www.tuxedo.org/jargon/html/entry/cosmic-rays.html`.

2.4.3 Implementational Issues

In implementing a system, the developer has to decide on where and how much fault tolerant techniques should be applied. The system's reliability should increase up to the the desired (or affordable) level, but it will be much too costly (time and money) to implement redundancy and tolerant measures in every bit of the system. As component failure can often be mathematically predicted (and is in most cases the underlying cause for failure), it is of use to apply probabilistic methods to assess these reliability parameters [LA90, p. 58].

This thesis will cover aspects of fault tolerance as indicated above at relevant positions. To provide for high availability, the system also has to contain deadlock detection and be able to re-synchronize with the host, once communication has been lost. It would be desirable to implement deadlock prevention, but that proves to be difficult to be implemented in a general manner [LA90, p. 135]. What will be implemented is the ability to detect deadlocks and react accordingly by sending error packets to signal the situation to the host (as will be done on detecting any other error, too), to re-synchronize if possible, and then retransmit the failed packet. This approach is similar to the one suggested in [SSBS99, p. 172], where in a concurrent system all pending transfers are stopped and the complete process has to be tried again. This of course adds to overall execution time, which very likely leads to a failure due to exceeded time-slots, as all these techniques mentioned above are developed for cyclic real-time applications and therefore the tasks have to be repeatedly executed in a strict schedule. In [AK83] a system is mentioned that has a deadline such that there is always time to execute a secondary (alternative) module, should the primary module fail, to at least provide a degraded service.

In the context of theoretical work, the concept implemented here is an implicitly defined recovery point, as on the occurrence of faults the process is rolled back in time and tried again. This of course only pays in the case of transient errors due to spikes on data lines, it does not help for more severe faults like bits of a data bus tied to a fixed level or floating lines.[16]

On the question of handling erroneous situations, it generally is good style to check all return codes for error values and act accordingly.[17] In [SSBS99],

[16]Described in [Pra80] as unidirectional and random errors respectively.

[17]It has to be noted, that in nearly all situations concerning calling of sub-routines or

however, it is stated, that "it is usually a waste of time to check the value." This remarkable sentence becomes more graspable after having read the complete paragraph. It is assumed that one can start sensible actions, once an error is detected. In parallel processes, as discussed in [SSBS99], there is not much that can be done after a communication error occurred, other than repeat the complete process. Therefore it is of little interest which value failing functions return.[18]

2.5 Reconfigurability

A lot of work has been done in the field of reconfigurability of FPGA systems up to the introduction of partially reconfigurable FPGAs like the new Xilinx Virtex[19] (and Spartan) devices. Not much can be found in the literature on reconfiguration of DSP systems at run-time, however. Since the FPGAs mentioned above are only reconfigurable in "columns" they are not that useful if one wants to have general freedom in reconfiguring devices or parts of one's design. Moreover, the internal organisation of the FPGAs has to be known very well to be able to divide the functionality into parts that can be re-loaded at all. This very device dependant behavior does not encourage developers who are trying to implement portable and open designs as they are bound to this device for a certain time.

Professor Villasenor,[20] from the UCLA Adaptive High Performance Scalable Dynamic Computing Department, states in the introduction to his proposal for dynamic computing architectures [Vil95] that throughout his proposal "Dynamic computing architectures, based on low-cost commercial reconfigurable logic technologies, will be developed for real-time military signal processing applications, with teraops computing requirements." and "Dynamically reconfigurable logic devices (FPGAs) will be re-used across several

functions, it is not only good style, but indeed necessary to check for the return values as only by checking on success or failure further problems, up to system crashes, can be avoided. This holds especially in concurrent and multi-tasking systems, where resources are not exclusively available. More details about proper error handling can be found in [Agl99].

[18] The user-level Message-Passing Interface, MPI, used in Beowulf is in fact shielding the application programmer from basic communications that no error handling at all is performed in the user program. If it does not work, the fault at hand is of such nature, that a program could not circumvent this problem all by itself in any case.

[19] URI: `http://www.xilinx.com/partinfo/databook.htm#virtex`

[20] URI: `http://www.ee.ucla.edu/faculty/Villasenor.html`

algorithms in real-time to increase functional density, while exploiting low-cost microprocessors to compute intensive tasks that do not map efficiently to FPGAs." Though Villasenor aims at the re-usability of FPGAs in space-, memory-, or device-restricted systems in connection with cheap microprocessors, the same principles are applicable to re-usability of memory attached to DSPs and the DSPs themselves.

In his proposal Villasenor also applies the term "dynamic computing" to FPGA systems [Vil95]. He divides dynamic computing into three parts:

- scalable dynamic architecture
- compilation tools for this scalable architecture
- tools for dynamic compilation

These terms are general enough to be used in other contexts, too. A scalable dynamic architecture is comprised through its definition as an architecture which is able to scale both performance and problem size. After upgrades it therefore not only speeds up execution time but is also able to solve larger problems. This is possible for many loosely tied multi-processor systems like a multi-DSP board or a network of workstations.[21]

Scalable architectures can be used simply as a conglomeration of workstations whose idle time is used or can be dedicated computing farms. In either case, special tools are needed to distribute the tasks at hand onto the available processing nodes. These tools can be simple scripts, starting tasks one after the other on nodes as they become available after completing the previous task, or they can be sophisticated compilation tools, implementing message parsing and employing concurrent tasks communicating with each other as they are executed on the computing nodes. These compilation tools especially have to consider partitioning as the computing nodes do not always provide the same processing power. Dynamic Compilation as a concept extends the concept from mere distribution of pre-compiled tasks to compilation at run-time.

At that point the available processing nodes are known and an optimal distribution of tasks can be dynamically assessed. This of course provides for the most flexibility in distributed computing as there are no restrictions

[21] One famous example of a network of workstations working to speed up problem solving is Beowulf clustering as described in [SSBS99].

for the tasks as they adapt dynamically to the available hardware at each execution cycle.

Since [Vil95] only targets his proposals on FPGA systems, the next step would be to use mixed architectures (DSP and FPGA both used simultaneously). In this thesis, however, the system is restricted to multiple DSPs attached to a host system.[22] Nevertheless, interesting aspects have emerged from research into mixed architectures [LEAP95, Müc01, FdAar].

[22]One can argue, however, that homogenous systems are simpler in programming and more straightforward, as only one target system is in operation and timing and synchronisation seems to be more compatible.

Chapter 3

Principles of Operation

3.1 Overview

In this chapter and overview of PEPSY and the context of this thesis to PEPSY is given. Principles of operation of the developed kernel are given as well as a description of the data transfer mechanisms from host to the DSPs and back to the host.

Development with PEPSY can be divided into different phases, shown in fig. 3.1 (p. 20). Inputs to the optimizer are the user defined application model and the system defined hardware model. Together with mapping constraints and optimizing parameters the optimizer computes mapping and scheduling. Code synthesis tools compile the user provided sources together with system specific communication and transfer tasks and link them to an executable code.

The rapid prototyping system PEPSY as it exists at the institute for Technical Informatics is based on accurate performance estimation [RRS00]. This concept will also be used here to provide compatibility to the existing tools. Accurate performance estimation is only possible due to the fact that the amount of execution cycles an assembler command needs on a specific target hardware is known before the schedule has to be constructed. This information can either be gained by compiling the sources and summing up the needed cycles command after command as listed in a look-up table, or by timing the execution while having the task run on a live target. The second method is supposed to be more accurate as live information can be gained, if one can determine the exact overhead imposed by the time keeping itself. The first means to trust data-sheets.

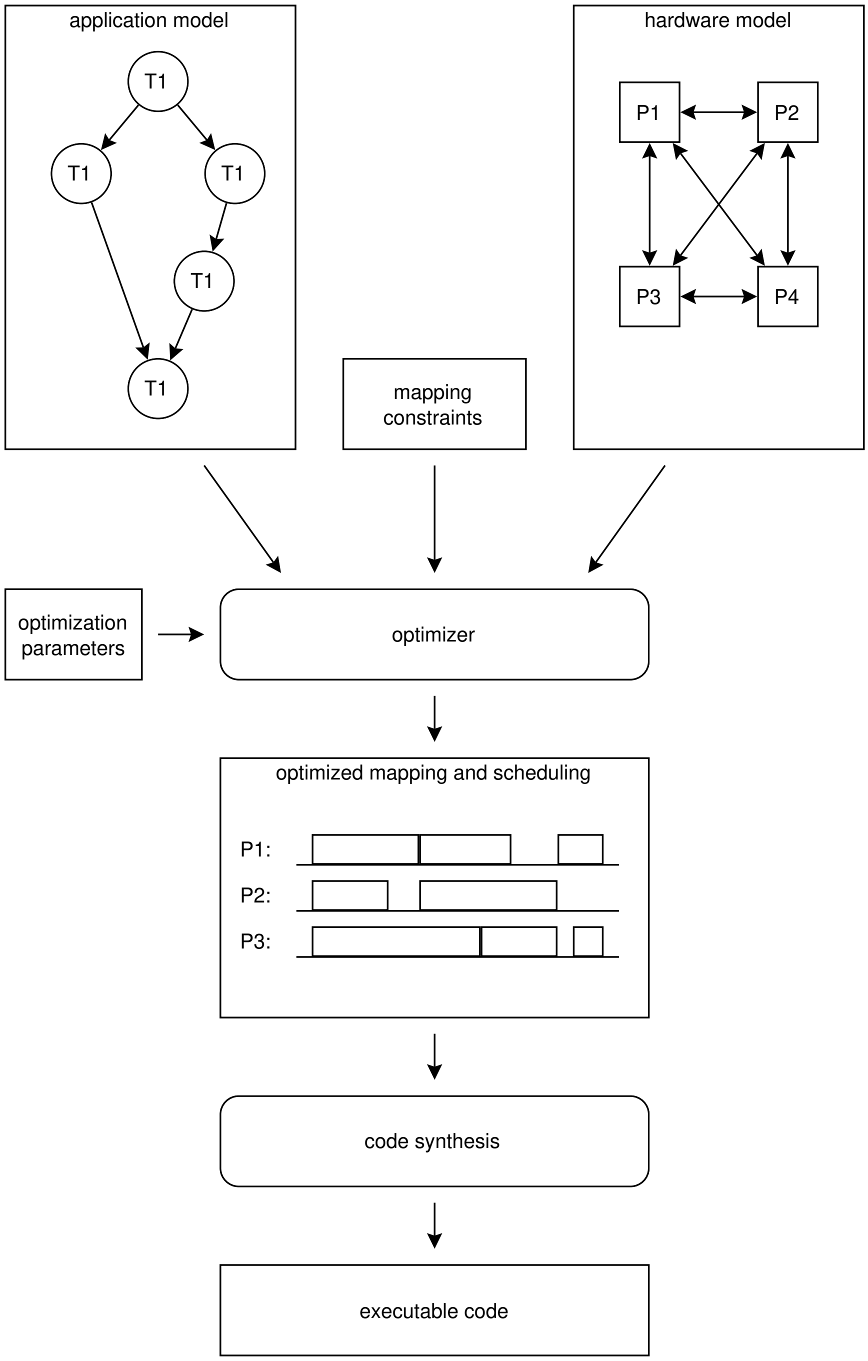

Figure 3.1: Design flow of PEPSY

In this thesis an extension to PEPSY is developed which allows task reconfiguration and data exchange in a run-time environment. It provides the possibility to add tasks to the executive and also to automatically kill tasks exceeding their initial time limit. Therefore a small kernel runs on the DSPs. This kernel has to be compiled into the executive as an additional task. As the application model is specified in human readable form, this addition can easily be managed. Despite the kernel's ability to run on each DSP in the system, there is no need for that as it can make much sense in a tight scheduling loop just to have the kernel run on a single DSP.

The kernel itself will be called as a task (but can also be run in a stand-alone mode) and then looks for commands transmitted from the host. If there are none, the application task the kernel supervises will be executed according to the flags provided by the user. If there are commands (like code downloading or uploading or starting of tasks) from the host they will be executed and control will be given back to the calling executive afterwards. With this the kernel is enabled to download code (or data) from the host into the DSP RAM and also upload data back to the host. It can also start and stop application tasks as well as having them run in a supervised mode to provide for increased system stability. Data transfer from and to the host computer is strictly packetized and consists of 16 words in each packet. In this, data transfer takes more time but real-time constraints can still be observed as only a fixed and small amount of data has to be transferred. Furthermore each packet is check-summed via CRC to provide data integrity and be able to recognize synchronization errors.

3.2 Data-flow model and code requirements

The existing system is modelled on research in rapid software prototyping based on accurate performance estimation as described in [RRS00], where "Each function has a unique name which is also specified in the application model." Data transfer from one task to another one is done via buffers located in DSP memory. This yields a simple structure throughout the system. To further increase simplicity and therefore testability of parts of the system, "a function implementing a task must not return a value" [RRS00]. Its parameters will be transferred to and from the task by the input and output buffers defined in the function prototype. The structure can be seen in fig. 3.2 (p. 22). Here `B1` through `B4` denote data buffers used for input and output. The location of these buffers will be passed to the tasks `T1` through `T3` by either parameters (if programmed in C) or as pointers on the stack (if programmed in assembler). This allows for a simple data driven structure of the system without any need of additional synchronization as all necessary synchronization is already achieved due to the fact that tasks have to wait for full input buffers before commencing work. For example, task `T3` can only start once buffers `B2` and `B3` are filled up by tasks `T1` and `T2`. A possible Gantt diagram could then show dependencies as in fig. 3.3 (p. 22), where `C` denotes communication tasks copying data from one processor `P1` to another processor `P2` and back to the first one after ending of task `T2`. The communication tasks (`C`), which move buffers, are pre-written by the system developer and provided for the user who writes the application tasks (`Tx`). This is because these communication tasks not only have to be highly target specific and optimized as much as possible, but also have to be provided in order for the rapid prototyping system to be able to automate the code generation process. As can be seen in fig. 3.3, task `T3` only commences once it has all data, despite of task `T1` having finished earlier. What is not shown in fig. 3.3 is the executive due to clarity. The executive of course also loops at the end of the execution of tasks.

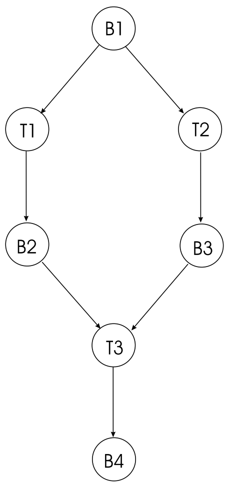

Figure 3.2: Dataflow in the original task model

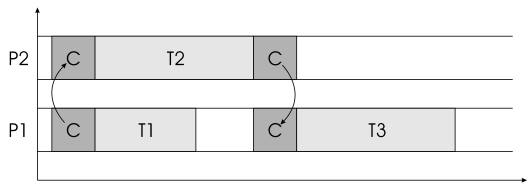

Figure 3.3: Execution order in the original task model

The synthesis tools to generate executable code have several requirements that have to be met by the programmer, for them to be able to produce multi-processor code for the DSPs. First, source code for all application tasks has to be provided by the user. Communication and synchronization routines are system dependent as is the memory allocation map and an executive. These have to be provided by the system developer. All but the first requirement can be automated, as the application always has to be provided by the user.[1]

The matter of providing sources for the different parts of the system seems trivial, but it has to be considered, that code for the communication tasks (depicted by arrows in the hardware model in fig. 3.1) is hardware-specific and has to be provided for an individual multi-processor system by the developer for that system as the user must not need to know all the internals. This system's principal intention is to introduce layers of abstractions between the hardware and the implementational issues the user has to deal with. They only have to provide code for the application tasks, everything else is part of the framework provided by the development environment. Only with these premises rapid software prototyping becomes possible.

The user provided source preferably is written in `C` and has to be syntactically compatible to the structures used within PEPSY. An example task prototype that can be used as a template for new developments is shown in fig. 3.4 (p. 23). This example task `task1` reads data from buffer `b1` and after computation writes its results into buffer `b2`. The definition of the sizes of the buffers and their allocation is one offline by PEPSY's optimizer.

```
D_TYPE in_buf[BSIZE_IN];
D_TYPE out_buf[BSIZE_OUT];

void task1 (D_TYPE in_buf, D_TYPE out_buf);
```

Figure 3.4: Example task Prototype

The communication and synchronization routines provide the intercommunication means for the tasks and implicitly implement the correct data flow as intended, as interprocessor communication is realized by introducing a dedicated sender task on one processor and a receiver task on the other

[1] The term user shall here denote the programmer actually using the rapid prototyping software, not the end-user employing the prototyped software.

processor, as is shown in the example executive source in fig. 3.5. Tasks only commence work once all necessary data has been gathered or arrived at their input buffers, which have to be allocated by PEPSY alongside the scheduling phase as the memory map is decided upon.

In PEPSY memory allocation for buffers can either be static or dynamic. Statically allocated buffers result in a faster execution of the executive. Dynamically allocated buffers are more memory efficient, since buffers can be released after the last receiving task has read the buffered data. Although memory efficiency would be desirable, dynamic memory allocation cannot be applied here as the buffers have to stay alive after execution of a task, because the dynamically introduced tasks could use any of them, but it will only be executed at the end of the predefined cycle, as can be seen in chap. 4.5 (p. 47).

The executive builds the main loop for cyclic calling of tasks and runs on each target processor and controls the execution of the tasks. An example C source fragment for an executive is shown in fig. 3.5 (p. 25). In this example communication buffers are statically allocated. A unique buffer is used for each arrow in the data flow graph used to generate that executive. The entry point (function name) for each task is derived from the task list generated by the optimizer. In this example `receive (b1, p1)` receives data from processor `p1` and writes it into buffer `b1`. `task 1` uses the data in `b1` to generate its output and writes it into buffer `b2`. `send (b2, p2)` sends then data in buffer `b2` to processor `p2`.

The order of task calls in the executive is defined by the schedule generated by PEPSY. It's optimizer is based on simulated annealing which allows a formal specification of different optimization objectives. This ensures optimal usage of available resources and fastest possible execution under given constraints and importance of different cost factors like time, speed or energy consumption.

The execution time of the tasks (which have to be known for the optimization to be able to work) are measured on a simple system with only one DSP attached. Once execution parameters of each tasks are known, PEPSY can build a system-wide schedule consisting of all tasks and assign remaining time slots to the newly developed mini kernel's watchdog facility.

For portability issues, XML has been used with PEPSY to describe the data flow and intercommunication of tasks and will also be used in this thesis to provide seamless integration into existing systems. Also human readability

```
D_TYPE b1[BSIZE1];
D_TYPE b2[BSIZE2];
D_TYPE b3[BSIZE3];

void executive()
{
    receive (b1, p1);
    task1 (b2, b2);
    send (b2, p2);
    task3 (b2, b3);
    send (b3, p3);
}
```

Figure 3.5: Example executive source

is pertained by the use of a non-binary format and thus changeability as well as adaptability by a user is provided.

The whole system will be automatically scheduled offline by PEPSY, based on performance estimation gained by analyzing the code to be compiled. The scheduling algorithm provides an optimal distribution of tasks onto available resources. Furthermore, the system can be partitioned into different executing entities by assigning costs (in the speed, time and energy domain) to each implementational method [Müc01].

In this model, SDF has been applied to describe the behavior of the system. This provides for simple mechanisms in automating the implementation by scheduling and partitioning tools, but it limits the possible applications as no dynamic aspects can be introduced into the system. All tasks have to be known at compile time and once the system is running, nothing can be changed without resetting it. To circumvent that restriction, this thesis develops further mechanisms to expand the abilities of the system.

The code generation tools communicate to each other by means of XML to provide for portability, human readability and ease of manual changes to the intercommunication. All the information they need to work properly can be found in these describing XML files, part of which are generated automatically by more basic tools. The input these tools need are the sources of the tasks and their logical connection to each other, i.e. the data flow in the system. They output readily compiled packages, which only have to be downloaded onto the DSPs to execute in the correct way and provide the intended behavior.

No provision has been taken, however, to prevent single tasks from wrecking havoc in the running system as only the tasks are running on the DSP and no supervising entity is available. It has been tried to address this in this thesis by application of simple memory protection and use of watchdog facilities to increase overall system stability and availability. On the other hand no further protective mechanisms have been implemented as this would unacceptably increase processing overhead and is one of the causes a small kernel has been favoured instead of a readily available real-time kernel as is further described in the next chapter.

3.3 Reconfiguration Mechanism

3.3.1 The Kernel

The kernel is the main part of the work done throughout this thesis as it comprises the main functionality introduced into the task model. This kernel is not only able to execute a task starting at a specific location in its memory space, it also downloads code chunks into its memory while protecting its own memory space, supervises the execution of the task and stops it if necessary.

The kernel can either be run in a standalone mode, taking over the complete DSP it runs on, or be switched into a special task mode to stay compatible to PEPSY and accompanying tools, where it will execute the task only once and return control and results back to the calling executive loop.

As a main feature, this kernel can be used in a multi-DSP environment with multiple instances of the kernel running in parallel, one on each DSP. This is made available and heavily simplified by the possibility to just pass on commands to the next DSP in a tree structure and return the up packet from that DSP. No local computation has to be executed to facilitate this as only forwarding of data is applied.

As there is no necessary communication between instances of the kernel on different DSPs, it will become possible to run this kernel on each DSP (as long as they are connected in a tree and the root DSP to the host) and only have commands passed through the other kernels. This system is freely scalable and adapts itself to available or actually used hardware.

To understand the communication taking place between host and DSP, the following sections explain in detail the commands passed on and returned on the line down from the host to the target DSP. As each packet is passed along with its corresponding CRC, integrity of data is ensured by this checksum over the complete pass as only the first and the last entity have to and will check that integrity.[2]

3.3.2 The Host Tool

The host computer, connected to the root node in the DSP tree, is the master of all communications. It is the only initiating entity, only the host can send commands. DSPs are merely able to answer commands or pass them on to the next DSP down the tree structure[3] (see fig. 3.6, p. 27). Only one DSP can be attached to the host as the host interface is realized via a single communication port. As the DSPs feature several communication ports (for instance 6 on a TMS320C40 DSP from Texas Instruments) the number of connections between DSPs is only limited by the amount of free communication ports.

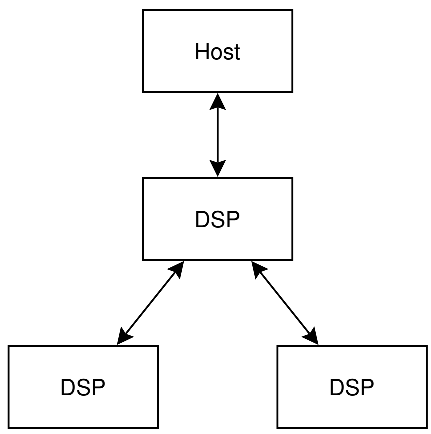

Figure 3.6: Tree structure of DSP connections

[2]That behavior can be compared to ATM, where only the endpoints of a network connection test on data integrity and not each forwarding router as was (and still is) true for other network protocols like FR.

[3]This is only TRUE for command execution and answers. Normal communication between DSPs as intended by their respective application is in no way influenced.

The host tool is able to show and use all of the facilities implemented into the mini kernel to extend the system to the desired level of abilities. With this tool the kernel on one of the connected DSPs is addressed and receives commands that are then executed by the DSP and an answer packet is sent back to he host. The mechanism implemented for command execution can be seen in fig. 3.7 (p. 28).

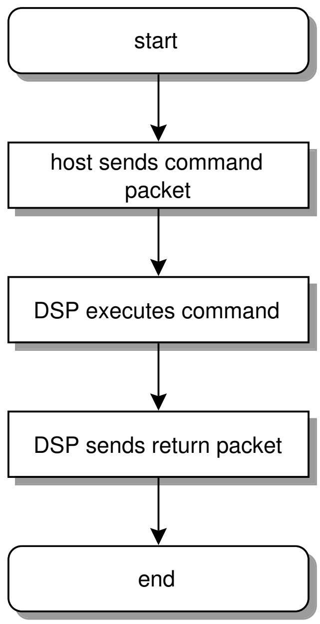

Figure 3.7: Mechanism of command execution

3.3.3 Communication Protocol

There is a protocol needed for the host to communicate to the DSP attached to it. Other DSPs connected to the first one cannot directly talk to the host, they have to be configured and provided with data and commands sent by preceding DSPs. These however will in return receive the commands from the host and only pass them along as is described in the routing bits of the command notifier (bits 32–8 in word 0 of each down packet).

This protocol has to provide secure and reliable data transfer. Data is transferred in binary without additional encryption, the protocol itself is a lookup code consisting of packets of a dedicated number of words to fill up the input and output FIFOs in both communication ports at the sending as well the receiving side of the communication. In the case of loss of sync between the two communicating parties, this situation shall be detected and cleared without user interference necessary. To address kernels on more distant DSPs, data forwarding has to be implemented to be able to serve DSPs connected to the host in a tree.

The first word of the transmitted packet determines the operation and the route the packet has to take as it passes along the DSPs to find the target. All communication originates on the host, the DSP is merely a slave on the communication port to the host.

Downloading of new code is also organized in packetized transfers. The number of these packets is determined by the host program to suit the kernel on the DSP as it has to fit into the real-time constraints imposed by already running tasks. The code will therefore be split into several packets, each transferred to a new start address as data transfer and a subsequent packet is sent to the DSP to start the newly downloaded task.

In the next chapter implementational issues are discussed and internal principles will be explained.

Chapter 4

Implementation

4.1 Target System

The target system used is a 19" industry PC with two Transtech[1] boards, each equipped with four TMS320C40 TIM modules (as can be seen in fig. 4.1 (p. 30) as a block diagram), and an ADAC analogue input/output card[2] specially designed for communication with the C40. Fig. 4.2 (p. 31) shows the connection between the Transtech board and the ADAC card.

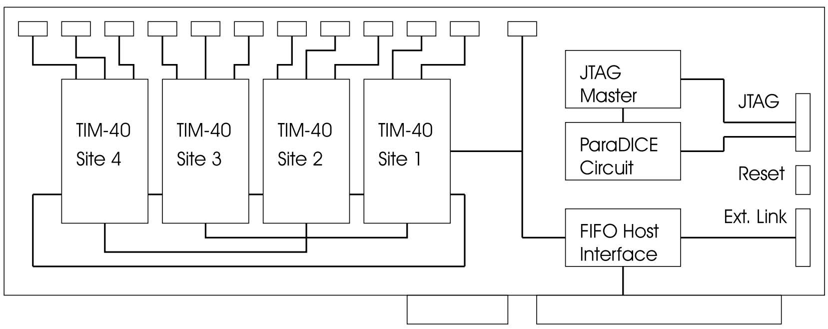

Figure 4.1: Block diagram of Transtech TDMB412

The TMS320C40 processor, called C40, used in this thesis is a digital signal processor made by Texas Instruments[3]. It has been designed for parallel processing and therefore offers several distinguished features:

[1]URI: `http://www.transtech-dsp.com/c4x/tdmb412.htm`
[2]URI: `http://www.adac.com/catpages/5403dhr.html`
[3]URI: `http://www.ti.com/`

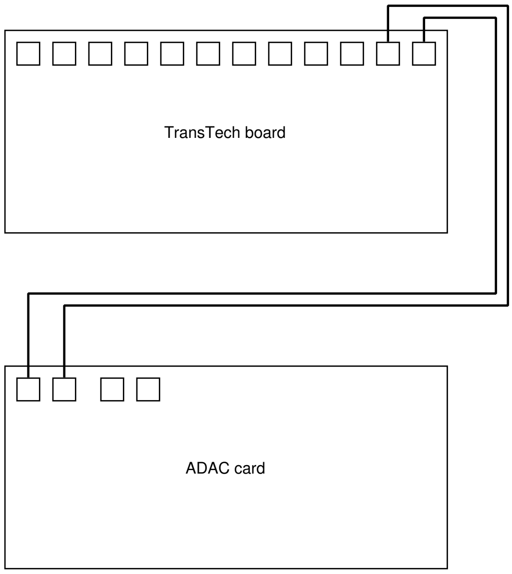

Figure 4.2: Connection between Transtech board and ADAC card

- The CPU has a floating point and integer multiplier, a 32 bit barrel shifter, internal registers and internal busses for data transfer. It is able to execute 32 bit integer and 40 bit floating point multiplications in a single CPU cycle.

- Memory can be up to 4 Giga words, each 32 bits. It is linearly addressed and can also consist of read only memory.

- There are internal busses on the C40 for parallel operations:

  - programme busses
  - data busses
  - DMA busses

  Additionally there are two busses for external memory.

- A very important feature of the C40 is its 6 communication ports which offer very fast bidirectional interfaces. They have a FIFO queue with a depth of 8 words. The maximum transfer rate is 5 megawords per second. They provide a simple means to connect several C40s to a multi-DSP system.

- The DMA coprocessor located on the C40 has access to all of the RAM via 6 channels without having to interact with the CPU.

The DSPs on the Transtech boards are already pre-configured in their connections to each other via several communication ports, but they also provide at least two free ports available for interfacing external hardware like I/O ports or DSPs on the other board. The internal structure is designed as such that tree as well as star topology can be realized for the system.

The Transtech boards also feature JTAG emulator ports to control the DSPs at run-time via the development and debugging tool Code Composer. That way developers can run their program while simultaneously reading out memory locations or DSP registers. One has to take care, however, not to kill the kernel once it is running by trying to download a task via the Code Composer as the DSP will be restarted at that point.

The root DSP in the DSP tree (fig.3.6, p. 27) is connected to the host only via communication port 1. On the host side this port is emulated by byte-wide accesses to specific I/O addresses. As each communication port on the DSPs feature an 8 word long FIFO, so does the host emulation of this port. This yields a FIFO depth of 16 words. This length of 16 words was also chosen as packet length for communication between host and DSP as it fills up the FIFOs on both sides of the communication and therefore eases detection of synchronization problems.

4.2 Protocol Commands

The commands, as listed in tab. 4.1, are packets consisting of 16 words each, the first being the command denotifier and the last a checksum over the packet. Unused words are filled up with zeroes. Only the host can initiate communication with the DSPs, the latter are mere slaves, only receiving commands or sending data on demand.[4]

Only the last two 4-bit nibbles (denoted `c` in fig. 4.3) contain the command. The first 6 nibbles (24 bits) describe the route to the target DSP.

[4]Again, this is only valid for DSP–host communication. The normal operation of the DSPs is not influenced by this strict client-server model.

command word	operation
0x00000000	request status
0x00000001	initialize download of code
0x00000002	download packet
0x00000003	start task
0x00000004	stop/kill task
0x00000005	send buffer contents to host
0x00000006	set mode
0x55555555	error packet

Table 4.1: Commands used in DSP-host communication

This concept provides for addressing individual DSPs as long as they are connected to each other in a tree structure and the root communicating with the host.

bits	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	cccc	cccc
nibble	5	4	3	2	1	0		

Figure 4.3: Structure of command word

As the command passes down the tree, each DSP getting the message checks nibble 0 (see fig. 4.3). This nibble describes the next target. If it is zero, the message is for the local DSP, otherwise it will be passed on via the communication port described in the nibble, and the nibble itself will be cleared by moving all remaining nibbles 4 bits to the right filling up the word with zeroes. Fig. 4.4 (p. 34) shows an example configuration with DSPs connected via various communication ports. To start a task on DSP 5 the command word would have to be `0x00009703`.

Word 15 in each down packet and words 14 and 15 in the up packet consist of a CRC, as calculated by the sending entity. Word 15 in the down packet (sent by the host) is the CRC over the 14 preceeding words, word 14 sent by the DSP is the CRC over the (down) packet sent by the host to the DSP, as calculated by the DSP. Word 15 in that packet is the CRC over all of the preceeding words, including the CRC for the down packet (word 14).

Below all the implemented commands are listed with a detailed descrip-

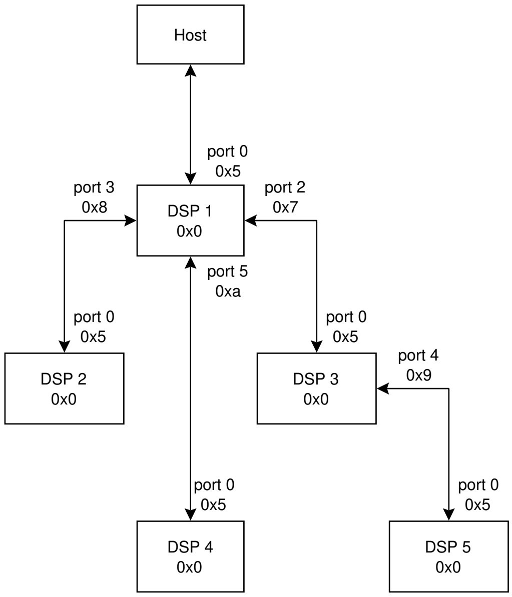

Figure 4.4: Addressing of DSPs

tion of their workings and a tabular enumeration of the contents of the accompanied command packets as they are transmitted from host to DSP and back to the host.

4.2.1 0x00000000 - request status

This command requests a status packet from the kernel running on a DSP connected through the tree to the host. The kernel version and some test value are transmitted with this command. It is implemented mainly for testing and debugging purposes. Furthermore the status of the watchdog and the timer interval is read back.

kernel version vv.rr.ss.ss: v... version, r... revision, s... sub-revision.

test value a value transmitted to the host for debugging and testing.

schedule flag

0x00000001	interrupt occurred, task too long
0x00000002	interrupt did not occur, task OK

word	contents
00	0x00000000
01	0
02	0
03	0
04	0
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	0
15	CRC (host → DSP)

Table 4.2: request status: data from host to DSP

word	contents
00	0x00000000
01	kernel version
02	test value
03	schedule flag
04	time interval
05	run mode
06	start mode
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	CRC (host → DSP)
15	CRC (DSP → host)

Table 4.3: request status: data from DSP to host

time interval the time span after which the watchdog fires if not reset before. If the watchdog fires, a task has utilized too much processor time. The watchdog is switched off if this value is set to zero.

run mode

0x00000001	run standalone
0x00000002	behave like a task

start mode

0x00010001	download only
0x00010002	run once
0x00010003	run continously

4.2.2 0x00000001 - initialize download

Due to the variable size of packets containing code, communication has to be done in two steps. First, the kernel on the DSP has to gain knowledge on how many words to await from the host, the data itself is sent in a second packet.[5] These two commands (`0x00000001` and `0x00000002`) are also used

[5] See chap. 4.2.3 (p. 37).

if memory or buffers on the DSP have to be filled up with predefined data.

word	contents
00	0x00000001
01	# of section
02	size of section
03	load address
04	start mode
05	start address
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	0
15	CRC (host $\rightarrow$ DSP)

Table 4.4: initialize download: data from host to DSP

word	contents
00	0x00000001
01	# of section
02	size of section
03	load address
04	download status
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	CRC (host $\rightarrow$ DSP)
15	CRC (DSP $\rightarrow$ host)

Table 4.5: initialize download: data from DSP to host

# of section if code has to be split into multiple parts, this value denotes the number of this part of the code.

size of section number of words to be downloaded for this section.

load address the address where data has to be stored.

start mode

0x00010001	download only
0x00010002	run once
0x00010003	run continously

start address the address where program execution has to jump to, to commit execution of the new task.

download status

0x00010001	allowed
0x00010002	denied

4.2.3 0x00000002 - download packet

Due to the variable size of packets containing code, communication has to be done in two steps. First, the kernel on the DSP has to gain knowledge on how many words to await from the host[6], the data itself is sent in this packet.

word	contents
00	0x00000002
01 – (size)	data
(size+1)	CRC (host → DSP)

Table 4.6: download packet: data from host to DSP

word	contents
00	0x00000002
01	# of section
02	size of section
03	load address
04	download status
05	task status
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	CRC (host → DSP)
15	CRC (DSP → host)

Table 4.7: download packet: data from DSP to host

# of section if code has to be split into multiple parts, this value denotes the number of this part of the code.

size of section number of words downloaded for this section.

load address the address where data (of this section) has been stored.

download status

0x00020001	successful
0x00020002	unsuccessful

[6]See chap. 4.2.2 (p. 35).

	0x00020001	downloaded only
task status	0x00020002	started once
	0x00020003	running continously

4.2.4 0x00000003 - start task

Once a task has been downloaded onto the DSP's memory, it has to be started, if it is not already running.[7]

word	contents
00	0x00000003
01	start address
02	start mode
03	time interval
04	stack address
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	0
15	CRC (host → DSP)

Table 4.8: start task: data from host to DSP

word	contents
00	0x00000003
01	start address
02	task status
03	time interval
04	stack address
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	CRC (host → DSP)
15	CRC (DSP → host)

Table 4.9: start task: data from DSP to host

start address the address where program execution has to jump to, to commit execution of the new task.

	0x00030001	downloaded only
task status	0x00030002	started once
	0x00030003	running continously

[7] It could already be started by using the start mode entry in the download packet command as described in chap. 4.2.2 (p. 35).

time interval The time span after which the watchdog fires if not reset before. If the watchdog fires, a task has utilized too much processor time. The watchdog is switched off if this value is set to zero.

stack address The address where the stack pointer has to be set to, to provide for correct initialization of task.

4.2.5 0x00000004 - stop/kill task

If a task does not behave well, takes too long to execute or is finished and its RAM will be needed, the task must be stopped and killed. A task that is only stopped can be run again by issuing the start task command.[8] A killed task is not in the DSP RAM anymore and cannot be started again.

word	contents
00	0x00000004
01	start address
02	kill mode
03	0
04	0
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	0
15	CRC (host → DSP)

Table 4.10: stop or kill task: data from host to DSP

word	contents
00	0x00000004
01	start address
02	task status
03	0
04	0
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	CRC (host → DSP)
15	CRC (DSP → host)

Table 4.11: stop or kill task: data from DSP to host

start address the address where program execution had to jump to, to commit execution of the task.

[8]See chap. 4.2.4 (p. 38).

kill mode

0x00040001	stop task
0x00040002	kill task

task status

0x00040001	stopped
0x00040002	killed

4.2.6 0x00000005 - send buffer to host

This is essentially the same functionality as in downloading code (or data) to the DSP, but in the other direction. As all commands are initiated by the host it is not necessary in this case to employ a two-phase protocol. Instead, just the command is sent to the DSP, which in turn causes the specific amount of data being sent back to the host.

word	contents
00	0x00000005
01	# of section
02	size of section
03	start address
04	0
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	0
15	CRC (host → DSP)

Table 4.12: send buffer to host: data from host to DSP

word	contents
00 – (size-1)	data
(size)	CRC (host → DSP)
(size+1)	CRC (DSP → host)

Table 4.13: send buffer to host: data from DSP to host

# of section if code has to be split into multiple parts, this value denotes the number of this part of the code.

size of section number of words downloaded for this section.

start address the address where data (of this section) has been stored.

4.2.7 0x00000006 - set mode

To allow applications the use of all timers, watchdog functionality can be switched off. In order to do so, a mode switch is implemented to set and get the mode of the watchdog and set the timer according to desired values.

word	contents
00	0x00000006
01	time interval
02	run mode
03	0
04	0
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	0
15	CRC (host $\rightarrow$ DSP)

Table 4.14: set mode: data from host to DSP

word	contents
00	0x00000006
01	time interval
02	run mode
03	0
04	0
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	CRC (host $\rightarrow$ DSP)
15	CRC (DSP $\rightarrow$ host)

Table 4.15: set mode: data from DSP to host

time interval The time span after which the watchdog fires if not reset before. If the watchdog fires, a task has utilized too much processor time. The watchdog is switched off if this value is set to zero.

run mode

0x00060001	run standalone
0x00060002	behave like a task

4.2.8 0x55555555 - error packet

If an error is detected, this packet is sent back to the host instead of the answer packet as described at the various commands. This packet signals the host that communication has failed and the host can react accordingly, for example simply try again.

No down packet.

word	contents
00	0x55555555
01	token word
02	0
03	0
04	0
05	0
06	0
07	0
08	0
09	0
10	0
11	0
12	0
13	0
14	0
15	CRC (DSP $\rightarrow$ host)

Table 4.16: error packet: data from DSP to host

To further increase fault tolerant behaviour of the system, this packet would be filled up with information on where a fault of which type occurred. Information could be passed on concerning lost connections to parts of the DSP tree or even regained communication notifies.

token word the word found at the position where a command denotifier was expected. This helps in determining how far off the packet is from the desired borders and how many words would therefore have to be flushed from the FIFOs.

4.3 The Kernel

The kernel is the basic part of the system. It therefore has to feature tight loops while testing for data on its controlling communication port and leaving as much time as possible for the task to execute. To ensure this, several provisions and design decisions have been taken as described below. The flow chart of the kernel is shown in fig. 4.5 (p. 43).

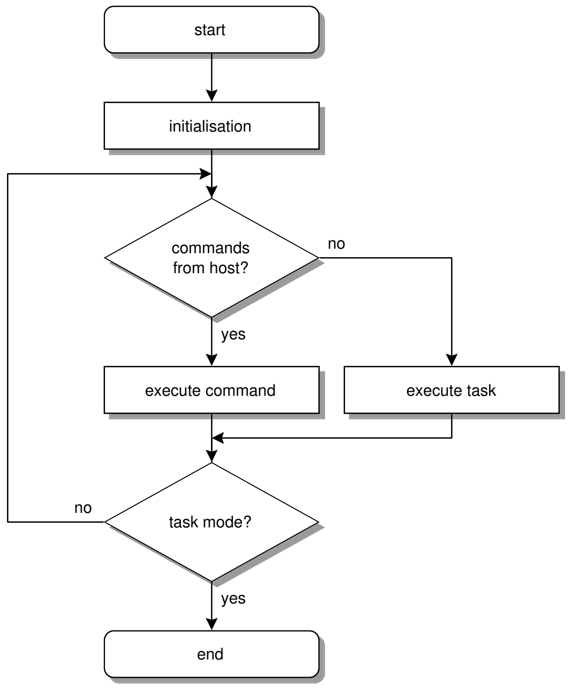

Figure 4.5: Flow chart of the DSP kernel

The kernel has been written entirely from scratch in assembler with help of Code Composer's online debugging facilities. Hence it features small footprint and fast execution. Due to modularity in the source it is easily extensible with additional commands and abilities. To keep the source small and execution times within limits, all administrative tasks have been transferred to the host. Therefore not only scheduling and mapping but also memory allocation has to be done offline. The kernel itself does not have any memory protection but for his own code segments. As the kernel is only a task if called by an executive, there is no way in knowing on where buffers and other application tasks are located. Therefore the memory map has to be manipulated on the host computer.

In the process of writing the code special attention was paid to constructing it in a completely relocatable manner. By recompiling the kernel source against a new command file, its memory map can be rearranged and its location altered. Even protection of its own memory space is relative to relevant entry points and features relocatability. While it is possible to put the kernel into local or global memory, for speed considerations it should be located in the on-board memory of the DSP.

As the tasks are downloaded, they constitute only data as far as the DSP is concerned. They only turn into executable code once the task is started on purpose by the user. This way, data chunks can be downloaded as well without the need to actually start anything at that point. This is useful for filling up buffers before testing new tasks or overloading data into corrupted memory in the case of faults to achieve at least degraded service by providing some default data.

The memory protection implemented in the kernel protects its own code and auxiliary memory locations from being overwritten while downloading code or data. Nothing can be done, however, to prevent tasks from doing harm once they are running as no protection can easily be built on processor level.

As one main objective has been fast execution and small footprint while conserving compatibility to existing systems, the other major aim was to implement fault tolerance principles. These principles applied to software development are again detection of errors and to react to them to avoid faults in the execution of code. As all packets are checksummed by a CRC recognition of transfer errors is made simple by checking this sum. Errors in communication can occur due to loss of synchronization and therefore shifting of packet borders.

Once communication becomes unreliable or packets are not scanned at the correct borders any more, automatic resynchronization will be started. Error packets will be sent, followed by a flush of the built in FIFOs, which effectively restores communication. If there is still some spurious data left, this process is repeated again until communication has successfully been restored. It was shown while developing that this concept reliably works even in severe cases of resets of only parts of the hardware or unreliable software operation.

As a further security against data loss or wrongly detected commands, each packet is checksummed by a CRC and both the host and the DSP test against this CRC for correctness of the packet. This also effectively enables both of the parties to detect communication problems and act accordingly by either resending the packet or flushing all the FIFOs and trying again. This not only works in DSP–host communication but also on the internal DSP communication lines.

4.4 The Host tool

The host tool has to work together with the kernel and so has to feature similar features like robustness and incorporate the same principles concerning fault tolerance. Once errors in the communication are detected it has to execute the same actions as the kernel does to regain communication. Due to flushing the FIFOs before trying again, these actions do not have to occur exactly at the same time, but the principle provides for some robustness.

As the host runs under Windows, development tools for this OS had to be used. This, however, proved to be problematic, as these tools not always worked as expected. Especially the software reset feature of the combined system of DSP boards and ADAC board did not work until a new tool had been written from scratch. This tool, however, reliably resets all three boards and provides a clean basis for further initializing of the system.

The host tool itself is written by use of the Borland C compiler entirely in plain C. It features commands as described below. These commands exactly resemble the possibilities built into the kernel running on the DSP as the host tool mainly has to show the features and enable the user to explore the capabilities of the system. To integrate the system into PEPSY, more tools and scripts have to be written to automatically control the kernels running on each DSP.

This tool is a command line tool, able to show and use all of the facilities implemented into the mini kernel to extend the system to the desired level of abilities. Several arguments are parsed by the tool. The template, as can be seen in fig. 4.6, shows the commands described in detail in chap. 4.4 (p. 45).

Below a description of each command presently implemented in the host tool is given with an explanation of the syntax.

INIT reliably initializes both the DSP board and the connected ADAC board. Although this is also issued by the host, it does not send commands to any of the DSPs, as it only writes into registers accessible via the ISA bus.

REQUEST_STATUS issues command `0x00000000`, displaying the returned information on screen.

DOWNLOAD_TASK issues commands `0x00000001` and `0x00000002`, downloading a piece of code or data, read from `filename.out` into

```
Usage: host <command>
  command         one of the following commands:
        INIT
        REQUEST_STATUS
        DOWNLOAD_TASK <filename.out>
        START_TASK <start addr> <stack pointer> <time interval>
        STOP_TASK
        SET_MODE <time interval>
        READ_BLOCK <start addr> <length> <filename>

  filename.out: COFF file of task to be reloaded
  start addr: entry address for task to start;
             start of block to upload
  stack pointer: address of stack
  time interval: max. time (in ticks) task is allowed to
               execute, 125 ticks == 10 us == 100 kHz
  length: number of words to read from DSP RAM
  filename: name of file to save block (if none, view
           block on screen)
```

Figure 4.6: Argument template of host tool

DSP RAM. If the destination space conflicts with kernel positions in the DSP RAM, download will be denied.

START_TASK issues command `0x00000003`, starting a task downloaded before from address `start addr`. It can be started once (if the kernel is used as a part of an executive loop) or continuously (for the kernel running standalone). It is also necessary to set the appropriate stack via `stack pointer` to the same position as denoted in the command file for the task to ensure correct execution. If `time interval` is a value in microseconds, this value initializes and starts the internal watchdog to fire after that time to kill the task if it has not returned before. If this value is set to 0, the watchdog will be switched off.

STOP_TASK issues command `0x00000004`, stopping a task from execution. As the memory allocation is handled offline, the task is effectively killed by this command as no internal provision for preserving the code space will be taken.

SET_MODE issues command `0x00000005`, setting several mode flags as described in chap. 4.2.7 (p. 41). The parameter `time interval` has the same functionality as with starting a task.

READ_BLOCK issues command `0x00000006`, reading in `length` words from DSP memory, starting at `start addr` and displaying them on the screen or writing them into the file, if the optional argument `filename` has been given.

4.5 Expansion to PEPSY

The expansion to the existing prototyping system basically consists of restricting PEPSYs buffer management to static allocations and introducing the execution of the kernel at the end of the executive. This enables the system to have access to all buffers for reading them out or overloading them with new values as well as executing new tasks if time remains at the end of the execution cycle. The task flow and execution order of the new model can be seen in fig. 4.7 and fig. 4.8 (p. 48) respectively.

These figures basically provide the same information as in figs. 3.2 and 3.3 (p. 22). The difference is the addition of the execution of the kernel `K` at the end of the executive just before it loops. There has to be provision in the executive for the kernel, i.e. the kernel has to be called. That can be as simple as a function call as the kernel in task-mode behaves exactly like an application task so no changes to the structure of the executive are necessary. As can be seen in these figures, the kernel is called and then calls itself the newly added task `Tx`. This task can, since the buffers are allocated statically, use any of the previously allocated buffers. Additionally it can read and write own buffers.

With the facilities provided by the kernel it is even possible to change the executive loop itself to a new one and to introduce tasks in a more permanent way once they have been tested and approved. Tasks entitled for permanent execution can be introduced into the system in three steps:

1. calculating new schedule including new task

2. downloading code of new task into DSP memory

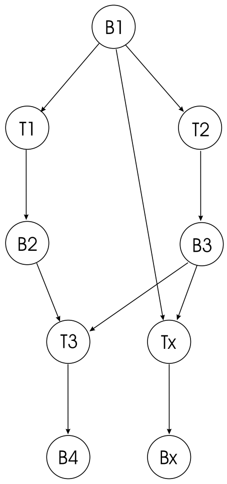

Figure 4.7: Dataflow in extended task model

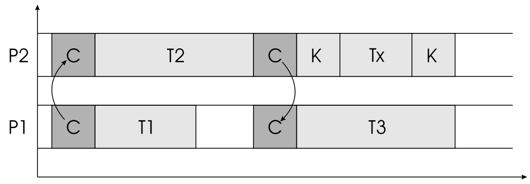

Figure 4.8: Execution order in extended task model

3. download new executive code and branch to new entry point

Tasks entitled for permanent execution have to fulfil several requirements as described in [BELP96, pp. 401, 405, 406]. They describe how to prove that a new schedule (calculated offline, as has to be done first with the utilities provided here) is still valid. The schedule can be found by first constructing a single-processor schedule for one iteration of the graph (using the optimizer developed in [Sch98]). A valid multi-processor schedule can then also be found. The liveness of this schedule (i.e. a loop within the flow graph) depends only on the consumption- and production-behavior of the tasks, but it has to be considered that the cyclo-static specification does not allow for any overlap between subsequent invocations of the same task. With these restrictions and mathematical framework as provided by [BELP96], new schedules can be approved and a new executive built thereupon. Once a new schedule has been found and approved the necessary additional code of the new task has to be downloaded into the DSP memory. A new executive built after the newly calculated schedule is also necessary and has to be downloaded, too. To switch the system to the new state, the host issues a `START_TASK` command to the DSP pointing it to the entry point of the new executive.

The limitations to the new model are of course, that the new execution time still has to fit into the (old) existing (and already running) schedule and enough memory has to be available. These requirements could be enforced by developing a completely new schedule with the tools developed by the partitioning tool developed at the institute [Müc01], but it remains to be seen if that reconfiguration of the system can be done fast enough in order not to influence proper execution of the system already in operation.

4.6 Writing New Tasks

4.6.1 Development

Writing new tasks is always a tedious job as newly written software never works on first attempt. Hence, rapid prototyping with fast turn around times is required to enable programmers to finish their jobs within deadlines. This thesis furthers the attempts already made at the institute to implement such a system on a multi-DSP board.

While the kernel developed here has been debugged with Code Composer's facilities for online supervising of DSPs and their codes, the same principle cannot be applied in such a straightforward manner for developing tasks, as that would throw off the kernel from the DSP.

Several provisions have to be made to ensure continued availability of the kernel running on the DSPs depending on the environment used for developing new tasks: Code Composer or the shell tools as provided by TI. If Code Composer is used for development of tasks, it must under no circumstances be connected to a DSP with a kernel running on it. The other possibility would be to use the provided `make.bat`[9] shell script to compile the task sources from within the shell environment, as has been done extensively while developing the core and accompanying test tasks.

In the case of Code Composer used for development of new tasks, testing of the new task has to be performed offline on an additional system (or at least a DSP not actually used by a running system) or within certain restrictions as depicted above on the target-system itself (which would have the advantages of yielding real timing). The kernel itself has been developed

[9]Small adjustments have to be made to the script depending on whether the sources are in assembler or C.

with Code Composer as this provides extensive debugging features, but tasks have either to be written offline, for they would interfere with Code Composer once it is connected to the target, or they have to be written in an external editor and compiled by the provided `make.bat` script in their own directory.

It is stated by [DLP97], that 30% of time and resources is used by nodes which are not processing data, but merely moving them. In order to reduce this waste of resources, restructuring of the nodes and their functions has been considered. Although this is more a design process than an implementational issue for the kernel itself, it is interesting to see, that improvements in cost and speed very much depend on the topology of the system (as shown by [DLP97]) and the mapping choices available. Furthermore these choices interact very much with other synthesis choices in a nontrivial way. Nevertheless for simplicity issues a tree structure has been chosen for connecting the DSPs as this topology is general enough since only up to five communication ports are available at each node, if C40s are employed.

While developing new tasks, one has to keep in mind that a system is already running on the target hardware and therefore a deadline is given. On the other hand that schedule is known, therefore the latest termination point is also known and the new task can be developed accordingly as it has to fit in this schedule to execute properly.

The kernel itself is fully relocatable and so no restrictions on its placement in DSP RAM if linked against a new command file are present. On has to keep in mind, however, that fast execution is only guaranteed for the kernel situated in on-board RAM. By using the precompiled kernel that property will be ensured.. Furthermore the kernel is written in a way, that no restrictions on use of registers in tasks are imposed, as relevant values are all stored in RAM adjacent to kernel code (if mapped correctly in command file) and therefore shielded by the kernel's memory protection.

Still several restrictions have to be kept in mind while writing new tasks. These will be described in the following section.

4.6.2 Restrictions

It would of course be preferable if one could implement a supervising kernel not imposing any restrictions whatsoever on the tasks to be run. This, however, is not possible if the hardware does not provide for such supervising functions other than via alternative access and command modes (JTAG in the case of a C40). The kernel developed in this thesis runs on the DSP itself and is not meant as a replacement for the online debugger as implemented in the Code Composer. Therefore it cannot prevent all possible problems and two main restrictions have to be imposed on tasks: no (or almost no) use of timer 0 and strict avoidance of use of memory space already allocated by the kernel.

The kernel itself has to stay small to fulfil requirements for small footprint and low overhead and therefore cannot itself check the validity of the newly downloaded tasks. All it can and will do is check for over-extended execution times of a task and kill it if needed, or at least delay it for later execution. This leads to the problem that this timer is already used for administrative purposes and therefore it cannot be used by applications anymore. As there is no arbitration scheme implemented on processor level, developers have to take care either not to use this timer at all or to switch off that watchdog function in the kernel.[10] Major problems arise if the watchdog function is not switched off and the timer still used within the application, as internal registers would be negatively affected and necessary function vectors would not point to their intended target anymore.

A more severe restriction is the avoidance of memory space already used by the kernel. Again, no arbitration scheme is implemented on processor level (as store commands would have to be supervised). Therefore the developer has to consult the command file for the kernel before compiling the task to ensure that no part of the task uses these memory spaces. This is also true for the stack pointer or system variables. Furthermore the vector table position is also predefined if the watchdog as described above is used. Lower priority interrupts could be used by the task nevertheless as long as the timer 0 interrupt is preserved. This, however, has to be done either by determining the actual position of the interrupt vector table as used by the kernel (not recommended because it would lead to memory accesses in kernel space), or by redefining the interrupt vector table at another position and redefining the interrupt vector table pointer (IVTP in the case of C40s). In the data flow model, however, no interrupts are necessary as the program flow is only dependent on presence or absence of data.

Overall, the restrictions imposed on tasks are not that severe, especially if a data-flow model is used as these models do not require application of timers or interrupts. Even for data reading or writing, no timers or interrupts

[10] See chaps. 4.2.7 (p. 41) and 4.2.1 (p. 34) for further information.

have to be initialized as the DSPs are stalled if the input FIFOs on the communication ports are empty or the output FIFOs full.

Chapter 5

Experimental Evaluation and Results

5.1 Performance

It is one of the major objectives of this thesis to implement a technique for reloading code onto running systems with as little additional overhead as possible. Ideally, the overall system behavior should not be influenced. Therefore the test if new commands coming from the host are to be served has to be very tight. This is achieved by the small inner loop as shown in fig. 5.1 (p.54). This loop adds only very few code words to the code to be executed. As a system should not already be running at the limits of its capabilities to avoid excessive transient failures, a few extra cycles can be added to the code. In the case of necessary reloading, however, the extra cycles to download batches of code into the DSP RAM have to be available anyway in order to make the system work at all, therefore the aforementioned extra cycles will not matter.

While running without any reloading activity present, the overhead is minimal as can be seen in the code extract in fig. 5.1 (p. 54). This is the main loop that will call the task to be run in a very tight loop, or, if running as part of a larger system, calls the task only once and returns control to the calling environment. Although there is additional code necessary to decide on the different run-time modes and intended behavior, that has been omitted from the listing for clarity reasons. This code chunk shows the important parts of the inner loop where tests on presence of words from the host are

```
LOOP:
LDI     *+AR0(0),R1 ; test if word is present on input
LSH     -9,R1       ; bits 9-12 contain # of words
    ; bits 13-31 read as zero
BNZ     CTRLENV     ; call CTRLENV if words present
CALL    TASK        ; call TASK if no words
BR      LOOP        ; loop back
```

Figure 5.1: Code sequence for test on presence of commands from host

performed, followed by a conditional jump to either the control environment to handle the requests or to execute the task.

In this figure, `CTRLENV` is the entrypoint for the control environment that communicates with the host (or the DSP connected to the upstream port) if there has been data present on the respective upstream communication port and is able to execute the commands described in chap. 4.2 (p. 32), whereas `TASK` points to the entry location of the task to be executed.[1] This entry point needs not necessarily point to a new task, it can also be the entry point of a new executive loop if it has been downloaded before. This provides for the possibility to adapt dynamically to changing environments or permanently incorporate downloaded tasks into a new schedule. In that way the kernel would then be free for another new task to be tested under its control.

A very important point in using this kernel is performance as the system in question is a real-time multi-DSP system. Therefore profiling has been used while implementing and it successfully showed that the overhead for tests on commands present from host is very minimal. Also transferring code between host and DSP is limited in its time usage as only packets with a length of 16 words are transferred in each cycle. The impact of these few operations on overall behavior is minimal and it can be said that the implementation of the kernel has successfully shown the possibility of implementing quasi-dynamic data-flow into an already existing environment.

[1] It would of course be possible to download one word at precisely the location, where the task is called and therefore modify the kernel to jump to another task, were it not for the basic memory protection that the kernel would not allow any download address and range reaching into its own memory locations. Instead a command is implemented for this reason.

It has been tested that at full speed of the ADAC board, which works at a maximum of 100 KSps, full operation of the kernel can still be ensured, while a task is executing. In units of the parameter passed to the `START_TASK` or the `SET_MODE` command,[2] a task can only complete in all cases if it will be allotted a minimum of 73 units. This means that the kernel uses $125 - 73 = 52$ units or $\frac{52}{12.5} = 4.16\ \mu s$. This conforms with observations made on the oscilloscope screen where a delay of roughly 5 μs has been shown.

As an estimation it can be said, that each word transferred will need 0.25 μs in a worst case scenario including overhead. That duration amounts to a maximum data rate of 4Mwords per second or 8MBps and is well under the maximum throughput of 20MBps (equals 5 Mwords each 32 bit per second) the DSPs are capable of. As only packets to 16 words each are transferred, the delay decreases with increasing packet length. But on the other hand, communications will take too long and tasks limited to too little processor time. So it has been found that 16 words not only fit nicely to fill the FIFO queues on both the sending and the receiving side, but also serve as a good trade-off between transfer speed and packet overhead in the time domain.

It would be possible to list individual delays for different packet sizes and FIFO queue availability. The packet length has been chosen in a way that influences due to interference of packet size and queue depth are ruled out, however. This ensures constant delays and provides predictability which otherwise could not be ensured due to variable delay lengths. In that the delay that is imposed on to the system is of constant length and only occurs if data transmission from or to the host takes place. This behavior of constant delay has been chosen to enable simplified incorporation of the kernel into PEPSY since offline schedulability remains possible. In the case of dynamic delays this offline schedulability would only be possible in either assuming constant worst case delay or being able to decide beforehand on the contents of the packet to be transmitted. The latter is not possible and the earlier results in the same system structure as fixed delay. Therefore the approach of fixed delays has been chosen.

[2]As the DSP is clocked by a 50 MHz oscillator, the internal clock frequency is 25 MHz in the case of TI DSPs. This frequency is again divided by two to yield a clock of 12.5 MHz for the timer counters. As the unit passed to the commands is connected directly to the clock frequency of the internal timers, this relates on the used hardware to a value of 125 for a 100 kHz looping frequency. This equals a value of 12.5 units per μs, but only integer values can be passed to the command.

5.2 Scalability

Scalability is a very important topic in developing multi-processor and multi-node systems. Overall requirements are rising by time and so it must be necessary to be able to upgrade the system without having to write new code for all of it.[3] This can only be achieved reliably by separating each node from the others. In this, influences from changes on one node to the others are limited and upgrades can be introduced step by step. In only applying small changes, the system behavior can be monitored more closely and errors introduced are observed more immediately. That serves for more ease in system maintenance and bug hunting.

Since the kernel developed in this thesis is completely transparent to the outside world (i.e. beyond the boundaries of one node), it does not matter on how many nodes in a multi-processor environment it is running. There is no interference between the structures on one node and on another one. Only the buffers transmitted over the communication ports are a connection, but they are handled by the tasks themselves so they do not constitute a change in the overall system behavior. Hence, scalability is not limited by the extensions made to the existing task and data-flow model.

Nothing can be said on the topic of overall overhead, however, as this depends on the tasks implemented and the intended use of the system. It very much depends on the application itself, whether processing power really increases by applying more nodes, as the administrative overhead in the system will rise as well.

5.3 Fault Tolerance

Fault tolerance by itself is a means to increase system stability and availability. It is implemented here to provide for fault free (though not necessarily error free) execution of code on DSPs while making possible to transfer data between different entities in the system. In this the implementation has succeeded as it is now possible to apply the already existing tools with added functionality.

Communication on a level besides task interaction takes place on the same hardware level, but is logically separated, so neither interaction nor

[3] On the other hand this also introduces problems with data and code integrity as pointed out by [Agl99, p. 94], but this shall not be an issue here.

interference is possible. As the kernels on different DSPs do not necessarily need each other, they can (but do not need to) communicate if there is more than one. However, this communication is only a passing along of commands issued by the host, and if the target DSP or DSP along the path to the target DSP are or become unavailable, no timeout mechanism has been implemented to prevent deadlock of this situation. The advantage of this separation is that there is no need to backtrack the complete system once an error has occurred in one node while communicating with the host.[4]

For further tolerance against such faults, this timeout would be necessary to signal the host that hardware does not react as expected and the host can (and should) redistribute the tasks according to the new tree. This functionality has not been implemented in this thesis, however, but that is a matter of recompiling the host sources, as all computing intensive parts like constructing a news schedule shall be kept away from the DSPs.

[4]It has to be noted that fault tolerant behavior possibly implemented in the application system is not part of the kernel and its communications and is therefore completely independent.

Chapter 6

Discussion

6.1 Conclusion

It has been shown in this thesis, that, backed by theoretical aspects, a small kernel running on a DSP can be implemented in order to introduce quasi-dynamic data-flow into an already existing system without changing its overall behavior. Furthermore multi-DSP capabilities have been implemented to make the system more versatile. It could also be shown that it is possible to implement this kernel in a way to stay compatible to the existing rapid prototyping system PEPSY and in the same time provide additional important functionality like task reconfiguration and basic memory protection, while still maintaining offline scheduling and even increasing security and availability of the running system. This kernel not only offers implementation examples of theoretical aspects, but also shows the separability of task inter-communication and kernel communication between DSPs and a host computer.

Theoretical research has shown that there are several data-flow models in existence, none of them providing for the flexibility needed and eventually implemented here. All of them are either too restrictive or too extensive to fit the needs. A new model, quasi-dynamic data-flow, has therefore been introduced in order to circumvent the restrictions imposed by existing systems. This model is especially suited for applications where data-flow is not known in all extent at compile-time, but offline scheduling still has to be possible. It resembles SDF in a way, that data-flow is rather restricted in its way and amount through the system while it being the timing defining parameter, but on the other hand provides the additional functionality for the extended feature set. It expands SDF's definition by introducing dynamic elements into the data-flow model, but does not go as far as a completely dynamic model, where offline scheduling is not a possibility anymore.

Several rapid prototyping systems have been reviewed and their usability tested against the requirements posed by the system design. The extensions to the existing system have been implemented to stay compatible to PEPSY, the institute's system. Despite the functional extensions, the overall timing of the system does not change if restrictions imposed by the data-flow model and implementational details have been kept in mind. Furthermore there is no interference with other additional tools added to PEPSY, as the kernel can be switched in fully transparent mode and acts as a task afterwards.

In addition to that, fault tolerant principles have been applied to increase availability and stability of the system. These methods have not been extensively exhausted but instead used on a problem specific basis in accordance with the need of the system, as it is never necessary and often too expensive in the time, effort and money domains to implement all features possible. Especially while implementing under tight timing constraints, only basic important functionality is implementable as more computing intense tasks would need too much time to complete. Therefore application of these methods has been limited in this thesis and hand optimized assembler code with few powerful features used instead.

While implementing the system, performance and a small footprint has always been an issue. Both has been achieved by the kernel, while in the same time allowing for a fair number of powerful commands. Features not needed on the kernel have been moved to the host tools, again to keep the kernel small and fast. The performance numbers have shown only a small increase in overhead due to administrative code and communication port handling. The overall performance can be said to be sufficient if for example audio applications are to be implemented. Even high-end audio with sample rates of up to 96 kHz is not a problem for the kernel as could be seen in the timing results. Of course the complexity of the tasks running on the kernel have to be indirectly proportional to the amount of data to be processed in a given time unit.

6.2 Further work

Several features have not been implemented as this thesis shall provide the theoretical background and basic functionality implemented to show the principles of operation of code reloading facilities. Therefore it has to be mentioned that not all functionality can be exploited by use of the current host tool. Adding additional features, already implemented in the (more important) kernel, is only a matter of recompiling the host sources, however, and do not pose any basic implementational problems anymore.

Further work would include optimizing of the kernel sources to fully utilize the DSP's parallel processing capabilities as well as incorporation of the kernel and the host tool into PEPSY providing for easy development and a consistent system available at the institute. As the host tool has basically been programmed to show the capabilities of the kernel it is intended more for educational purposes than for development processes. Integration into PEPSY will render this host tool superfluous and a common development environment will exist once this integration has happened.

Bibliography

[ADAar] ADAC. *5404DHR User Manual.* ADAC Corporation, unknown year.

[Agl99] Thomas Aglassinger. *Error Handling In Structured And Object-Oriented Programming Languages.* Master's thesis, Department Of Information Processing Science, University of Oulu, 1999. `http://www.giga.or.at/~agi/thesis/`.

[AK83] Thomas Anderson and John C. Knight. A framework for software fault tolerance in real-time systems. *IEEE Transactions On Software Engineering*, SE-9(3):355–364, May 1983.

[BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions On Signal Processing*, 44(2):397–408, 1996.

[Bha99] Bishnupriya Bhattacharya. *Parameterized Modeling And Scheduling For Dataflow Graphs.* Master's thesis, Institute for Advanced Computer Studies, University of Maryland, 1999.

[DLP97] L. De Coster, R. Lauwereins, and J. A. Peperstraete. Data routing in dataflow graphs. *Proceedings Of The 8^{th} International Workshop On Rapid System Prototyping*, pages 100–106, 1997.

[FAdar] Virginie Fresse, Mustapha Assouil, and Olivier deForges. *Rapid Prototyping Of Image Processing Applications Onto A Multiprocessor Architecture.* Laboratoire Artist, Rennes, unknown year.

[FdAar] Virginie Fresse, Olivier deForges, and Mustapha Assouil. *Rapid Prototyping For Mixed Architectures.* Laboratoire Artist, Rennes, unknown year.

[FMF98] Franz Fischer, Annete Muth, and Georg Färber. Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping environment. *Proceedings Of The 6^{th} International Workshop On Hardware/Software Codesign*, pages 35–39, March 1998.

[Go 98] Go DSP. *Code Composer User's Guide*. GO DSP Corporation, 1998.

[LA90] P. A. Lee and T. Anderson. *Fault Tolerance – Principles And Practice*. Springer-Verlag Wien New York, second edition, 1990.

[LEAP95] Rudy Lauwereins, Marc Engels, Marleen Adé, and J. A. Peperstraete. Grape-II: A system-level prototyping environment for DSP applications. *Computer*, 28(2):35–43, 1995.

[LGH80] P. A. Lee, N. Ghani, and K. Heron. A recovery cache for the PDP-11. *IEEE Transactions On Computers*, C-29(6):546–549, June 1980.

[LYC99] Christopher J. Lloyd, Paul S. F. Yip, and Kin Sun Chan. Estimating the number of faults: Efficiency of removal, recapture, and seeding. *IEEE Transactions On Reliability*, 48(4):369–376, December 1999.

[Müc01] Manfred Mücke. *Ein Programm zur automatischen Partitionierung von heterogenen Multiprozessor-Systemen*. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2001.

[Pra80] D. K. Pradhan. A new class of error-correcting/detecting codes for fault-tolerant computer applications. *IEEE Transactions On Computers*, C-29(6):471–481, June 1980.

[RRS00] B. Rinner, B. Ruprechter, and M. Schmid. *Rapid Prototyping Of Multi-DSP Systems Based On Accurate Performance Estimation*. Technical Report, Institute for Technical Informatics, Graz University of Technology, 2000.

[Rup01] Bernd Ruprechter. *Rapid Prototyping von Multi-DSP Anwendungen anhand einer audiotechnischen Anwendung*. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2001.

[Sch98] Martin Schmid. *Parallelisierung und Implementierung von Simulated Annealing auf einem Multi-DSP System.* Master's thesis, Institute for Technical Informatics, Graz University of Technology, 1998.

[ŠRU97] Jurij Šilc, Borut Robič, and Theo Ungerer. *Asynchrony In Parallel Computing: From Dataflow To Multithreading.* Technical Report, CSD-97-4, Computer Systems Department, Institut Jožef Stefan, 1997.

[SSBS99] Thomas L. Sterling, John Salmon, Donald Becker, and Daniel F. Savarese. *How To Build A Beowulf.* MIT Press, Cambridge, Massachusetts, 1999.

[Tex96] Texas Instruments. *TMS320C4x User's Guide.* Texas Instruments Inc., 1996.

[Tex97] Texas Instruments. *TMS320C3x/C4x Assembly Language Tools.* Texas Instruments Inc., 1997.

[Tra94] TransTech. *TDM411 EDRAM TIM-40 User Manual.* Transtech Parallel Systems, 1994.

[Tra95] TransTech. *TDMB412 Rev.2 User Manual.* Transtech Parallel Systems, 1995.

[Vil95] J. Villasenor. *Proposal For A Dynamic Computing Architecture.* URI: `http://www.icsl.ucla.edu/Reconfigurable/Papers/proposal.html`, 1995.

[Vra96] Harald P. E. Vranken. Design for testability in hardware-software systems. *IEEE Design And Test Of Computers*, 13(3):79–87, Fall 1996.

Appendix A

Acronyms

ATM	**A**synchronous **T**ransfer **M**ode
CSDF	**C**yclo-**s**tatic **D**ata-**f**low
DDF	**D**ynamic **D**ata-**f**low
DSP	**D**igital **S**ignal **P**rocessor
FIFO	**F**irst-**I**n **F**irst-**O**ut buffer
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FR	**F**rame **R**elay
Grape-II	**Gr**aphical **Ra**pid **P**rototyping **E**nvironment
OS	**O**perating **S**ystem
PEPSY	**P**rototyping **E**nvironment for multi-DS**P** **Sy**stems
RAID	**R**edundant **A**rray of **I**nexpensive **D**isks
RTOS	**R**eal-**t**ime **O**perating **S**ystem
SDF	**S**ynchronous **D**ata-**f**low

Appendix B

Terms

Data-flow graphs are used to visualize the order of events an algorithm has to trigger to execute properly. These graphs describe the flow of data in the course of programs execution. A program, therefore, is represented as a directed graph, where the vertices correspond to application tasks and the edges indicate data dependencies between these tasks. Basically, data transfer can be buffered (and so provide for un-synchronisation possibilities) or unbuffered (which means that sender and receiver have to be tightly synchronized not to lose data). [LEAP95, BELP96]

Homogenous (single-rate) data-flow as the simplest model of data-flow methods behaves like a FIFO of unlimited length. As soon as data is present at each data input, the vertex (task) may be executed and can fire output data. The fire rule is implicit in this model as it does not have to be specified by the programmer. This model makes the execution intervals of vertices depending on the frequency of data in the way that all data arrives at the vertex with the same frequency and so the vertex fires with the same frequency at its output as it will be called exactly once if here is at least one data token at all its inputs. [LEAP95, BELP96, Bha99]

Synchronous (multi-rate) data-flow denotes the property of vertices to only fire if all required data is present. Vertices are to wait for its input and cannot become active without sufficient data. Almost all implementations are synchronous as there is no point in wasting resources by executing code without sufficient data. Exceptions are situations where it is simpler to implement or yields better stability in the time domain by executing exactly the same commands in each loop. This model, in contrast to single-rate data-flow, can handle incoming data at different frequencies as it will pull different amounts of data tokens from each input for each of its executions. The data output frequency can again be different from the input frequencies. [LEAP95, BELP96, Bha99]

Cyclo-static data-flow expands the possibilities of synchronous data-flow even more as it can describe situations where it will be necessary to alternately serve different outputs at different frequencies, while the single-rate and multi-rate models can only operate at a single output, or at multiple outputs with the same frequency of data on each of them. A possible application for this model would be a de-multiplexer unit. [LEAP95, BELP96, Bha99]

Dynamic data-flow comprises the most general model in which consumption and production is unknown at compile time and therefore needs a run-time scheduling mechanism. Thus, a large run-time overhead is required. Many problems, however, can be implemented more efficiently as their behaviour is more determinable and the universality can be restricted to special cases which in turn will be executable under simpler data-flow models. Dynamic data-flow is powerful, but better avoided in tight real-time applications due to its unpredictability. Hence, a lot of extensions to the synchronous data-flow model have been proposed. [BELP96, Bha99]

Grape-II is a system-level prototyping environment for developing DSP applications with general-purpose reusable hardware. It automates the prototyping methodology by offering tools for resource estimation, partitioning, assignment, routing, scheduling, code generation, and parameter modification at run-time.[1] Grape- II consists of a set of tools which interact with two central databases, one containing application specific data, the other describing the features of the target systems. The environment can handle mixed (DSPs and FPGAs) as well as homogenous architectures. One large advantage of this system is the ability for the developer to use an optimised code generator to enable register usage across task boundaries. [LEAP95]

[1] Grape-II uses a heuristic branch-and-bound algorithm.

PEPSY represents an environment to automatically build multi-processor solutions for real-time applications. At the core of PEPSY is an optimiser, which is distributing the discrete functions (tasks) of an application onto a parallel multi-DSP system in an optimal way.[2] The result will be used by a code generator to gain C code for calling the given functions. For the exchange of data files between the components of PEPSY, special XML formats are used. [RRS00]

SynDEx as another rapid software prototyping tool distributes and schedules the data-flow graph on a multicomponent hyper graph while satisfying real-time constraints. The distribution and scheduling are calculated offline for static execution sets. SynDEx generates an intermediate macro code, which is a direct translation of the obtained abstract distribution and schedule. Finally, a macro processor generates the appropriated executive for the target architecture.

Code Composer from Texas Instruments is an integrated development environment featuring not only a compiler but also debugger facilities as well as online supervising and maintenance facilities on a very low level. It is possible to watch and change certain CPU registers and memory locations. Code Composer also provides the programmer with multi DSP capabilities to develop code for several connected DSPs all at once and have them running and supervised all at the same time.

[2] PEPSY applies simulated annealing to solve the mapping problem.

Appendix C

Kernel Code Listing

The latest kernel sources can be found at:
`http://www.rockus.at/gerler/mastersthesis/`