

# A Microcomputer Interfacing Laboratory

J.D. Garside

Department of Computer Science, University of Manchester

Oxford Road, Manchester, M13 9PL, UK

jgarside@cs.man.ac.uk

Keywords: Teaching, Microcontroller, Interface, ARM, FPGA

## Abstract

*The majority of computer processors sold today are used as embedded controllers; devices which rely on a mixture of hardware and software in a real-time environment. The teaching of this subject falls somewhere between computer science and electrical engineering. This paper describes a laboratory developed to provide an easy-to-use solution to fill this gap.*

## 1. Introduction

Whilst most laymen might still think of a “computer” as a large, plastic box with a keyboard and monitor, such devices represent only a small (and decreasing) proportion of the industry. The majority of computers are embedded controllers – devices which are built into an ever-increasing variety of equipment. Unlike ‘traditional’ computers these microcontrollers often rely on intimate cooperation between their hardware (for speed) and software (for cheapness); if there is time available some functions normally supported by dedicated hardware may be implemented in software and vice versa. For example Dynamic RAM (DRAM) refresh could be relegated to an interrupt routine whereas encoding of communications packets may be partially supported with a coprocessor. There are a number of trade-offs involved and an increasing demand for engineers with an appropriate background.

The teaching of embedded control in universities often lies between the remits of computer science departments – who typically approach the subject (if at all) with a heavy software bias – and electrical engineering departments – who usually have a primary interest in the hardware. However the topic crosses this boundary and it is therefore desirable to offer a course which blurs this divide.

In order to introduce students to this discipline a course has been developed within a new degree programme – Electronic and Computer Systems (ECS) – at the Computer Science Department at the University of Manchester. This particular course is somewhat unusual in that it is entirely

practical-based; engineering is a practical subject and the course should therefore force the students to produce their own designs and allow them to make their own mistakes. This naturally reduces the ‘breadth’ of the syllabus but increases the memorability of the lessons! After considering both the course requirements and the available equipment the hardware and most of the software used was custom-developed. This was a significant undertaking, but appeared justified at the time and the results seem to bear out this decision.

The wider intent was to create a set of exercises, together with supporting equipment, as a ‘kit of parts’ which may be used to build other university courses (or parts thereof). Naturally such a kit must be cheap, flexible and extensible. This work is ongoing and is being done in association with a small group of academics at various universities; the initial group includes Manchester, Imperial College, and the University of New South Wales, with expressions of interest from several other sources.

## 2. Intention

*“Things should be made as simple as possible  
– but no simpler.”      A. Einstein*

The aim of the laboratory is to give the students a solid experience of computer interfacing. This includes input and output, timing, interrupts and real-time operation. Some of these concepts can prove intimidating at first encounter and it is quite easy for students (and, indeed, professionals) to take the ‘easy’ way out. For example, it is ‘easier’ to poll a peripheral than set up interrupt routines; a software delay loop is ‘easier’ to write than programming a hardware timer. It is only when several simultaneous, unrelated I/O activities are required that the benefits of ‘correct’ design become apparent.

As is revealed from a short perusal of its data sheet an ‘off the shelf’ microcontroller can be a complex device and, frequently, not one amenable to the inexperienced programmer. An important consideration within the laboratory is therefore to introduce complexity

incrementally. To support this end it is desirable to have a range of simplified peripherals which provide just enough functionality for each exercise. It is also helpful to simplify the development environment (both for hardware and software) or find tools which are easy to use, or already familiar.

### 3. Background

In embedded solutions it is not always clear whether a hardware or a software solution is best for a particular function and, quite often, a mixed solution is employed. The object of this course is to practice development at this level and allow the option of mapping a design into hardware or software.

In an industrial environment microprocessors have been available as library components for several years now and standard ASICs have been large enough to incorporate them. Somewhat more recently semiconductor manufacturers have begun to produce FPGAs which incorporate microprocessor macrocells, one example being Altera's Excalibur™ which contains an ARM processor. Such devices reflect the existing ASIC design flows but incorporate the advantages of fast turn-around and reprogrammability inherent in FPGAs. It is clear that these devices can both act as a model for current ASIC development processes and, in the longer term, replace some of them.

Although such state-of-the-art parts can be expensive and in short supply in an educational context they can easily be replaced by a less integrated solution. Therefore the course has been developed around an experimental circuit board which provides the following facilities:

- an ARM-based microprocessor system
- a user-configurable FPGA
- a customised software environment

#### 3.1. ARM

The ARM [1], a UK-designed microprocessor, is now established as the leading 32-bit microcontroller architecture; ARM programming therefore provides experience relevant in the marketplace. Being a commercial architecture there is extensive software support available. Lastly (but significantly) ARM Limited is also keen to support the promotion of their products within UK universities.

Programming a completely 'blank' system necessarily requires the use of assembly language. ARM has a straightforward, (almost) orthogonal instruction set, unlike most 8-bit microcontrollers. The architecture is heavily RISC-inspired, but retains some features, such as condition codes, which are features of CISC processors; it therefore

occupies a position from where migration to other architectures should be relatively easy. The processor is fully interlocked so features such as register dependencies and branch delay slots are not exposed at instruction set level. It is therefore a straightforward teaching vehicle. These reasons had already led to its adoption as the example architecture taught at Manchester University.

Finally there are development tools available so that the hardware can be adapted for use with other programming languages and the interfaces between C (for example) and assembly language are defined and supported. This in turn allows the equipment to support other laboratories or project work.

#### 3.2. FPGAs (Field Programmable Gate Arrays)

FPGAs offer an electronically programmable replacement for the prototyping 'breadboards' once used in teaching digital electronics. These parts are large and cheap enough to encompass any feasible student design exercise. Parts are available which are indefinitely reprogrammable and, for the performance levels required, CAD tools can compile a schematic to an FPGA netlist adequately, without user intervention.

Preparing circuits this way has a number of educational advantages over hand-wired designs: there are no problems introduced from bad or intermittent wiring contacts and faulty devices are very rare and easily diagnosed and replaced. It is also possible to concentrate teaching effort on the design process by employing hierarchical design instead of having to construct many copies of some basic circuit by hand. This allows much more ambitious designs to be attempted within a given time, an important consideration within an industry that is progressing so rapidly. To achieve even larger circuits it is possible to supply some functional units pre-built for only a 'one-off' start-up cost; to do this with a breadboard required many hours of time for a skilled technician.

Finally it is possible to recompile the netlists already demonstrated using FPGAs using a standard cell library to allow the production of an ASIC. Although this does not add greatly to the educational experience – except for the few students with enough time that they can be exposed to some of the ASIC back-annotation and verification – it is a powerful motivating factor. With current levels of integration many student designs can be accommodated in a single device, thus reducing the cost overall.

At Manchester University we have been using such a design flow for over ten years. It has proved extremely successful in allowing students to produce complex designs. A standard second year exercise is now the implementation of a simple RISC microprocessor. A number of student designs prototyped this way reach silicon each year.

### 3.3. Software environment

As well as the target hardware it is important that there is software support available. Development tools for the hardware (FPGA) development are already available at educational rates via Europractice [2]. Software compilers and assemblers are available from ARM Limited, and from other sources. In addition to this software has been developed to integrate the user's view of the environment and also to simplify the system to allow teaching to concentrate on the generalities of what is being taught rather than system implementation detail. This is described later.

## 4. Alternative Systems

Instead of the microprocessor/FPGA based hardware a number of other possible schemes could be employed. Here some of the rejected alternatives are briefly described and the reasons for their rejection given.

### 4.1. Simulation

A software emulation of the entire microcontroller system could be produced and run on a workstation. This has one great advantage: it removes the need for physical hardware with a consequent reduction in system cost, especially in large laboratories. However there are also significant disadvantages to this approach.

The simulation would need to be a considerable system. It would require both a processor emulator to execute the software and an integrated hardware simulator to run the gate models. While both of these are available as stand-alone products the integration of these, together with an I/O interface to allow user interaction, would present a considerable development problem.

While this is surmountable (with difficulty) the simulation is always going to lack one thing – the allure of the physical system to the student. It is undeniable that there is more satisfaction in producing a working artefact than in a mere simulacrum. Furthermore – with appropriately designed exercises – it can be easier to spot problems with a real implementation, both for the student and the laboratory supervisor. A simple example of the latter case is a seven-segment display decoder, where a logic error anywhere soon becomes very obvious!

### 4.2. PC and I/O board

The next step from a pure simulation would be to simulate the processor on a PC and map some of its I/O space onto a dedicated circuit board for user interaction. Ideally such a circuit board would be external to the PC – to allow the use of different 'plug in' interfaces – and

communicate through some standard means so that it was not tied to a particular host system.

If this is done the most obvious approach is to fit a microprocessor to the I/O board and communicate via a serial link. If this is accepted then it is a small step to running the user's application remotely, freeing it from the vagaries of the PC.

### 4.3. 'All FPGA' solution

Modern FPGAs can easily hold hundreds of thousands of gates. One credible solution would therefore be to use an existing FPGA development board and integrate the processor into the FPGA itself. Since this project was begun, such a solution has been provided by Altera's Nios embedded processor [3].

This solution has three disadvantages. The first is cost; while small FPGAs are quite cheap large ones are still expensive and this must be multiplied by the capacity of a laboratory. The second disadvantage is that the processor model must be obtained or developed. This represents a considerable expense in time but is only a 'one-off' cost and so is not necessarily significant. The desire to retain software compatibility with other courses may restrict the options here, though.

The third disadvantage to this scheme is the compilation time of the hardware. Student designs will typically require a few hundred gates and are relatively quick to compile; however if the processor has to be added to this each time the overhead will be significant leading to a long design turn-around time and rapid disillusionment.

### 4.4. Integrated processor/FPGA devices

A number of devices (such as 'Excalibur™') are becoming available which contain both a CPU and an area of FPGA. These appear ideal for the application in hand. However at time of writing the cost and availability of these is uncertain. Even if the hardware is available at an economic price there must also be library support for the development tools available through a subsidised process.

### 4.5. ARM's development tools

As would be expected ARM Limited provide development tools for loading and debugging software both in simulation and on remote target systems. Superseded versions of the toolkit are available free for education purposes. Whilst it is anticipated that this should be supported by the hardware it is less than ideal for teaching purposes.

ARM's tools are intended for experienced developers; such users can be expected to treat the system with some respect. This is necessary because, for example, resetting

the target system can require a restart of the host software. Such abuse will be common in a laboratory, where an ideal system should be able to recover seamlessly.

More seriously the target system is not ‘clean’; the debug monitor requires certain facilities which the user interferes with at his peril. A notable example is the interrupt system where a user must *supplement* the existing interrupt service routines – overwriting them will cause a loss of the host link! An experienced user can be expected to do this routinely but it is hopelessly confusing for a neophyte.

The system would also need extensive modification to support the programming of any other devices (specifically FPGAs) which are hosted off the same serial line. A protocol which shared the communications link would therefore need adding to both ends of the software, in itself a considerable undertaking.

Lastly, while the compilers, assemblers etc. are all available in command line versions, the windowed system is currently only available under Microsoft’s Windows. This renders it incompatible with Manchester University’s existing FPGA development routes which are all under Linux. Command line driven versions of the software are available but tend to produce the ‘wrong impression’ on modern students. However such tools can be executed in the background by scripts and are therefore useful for supporting the finished system.

## 5. Environment

### 5.1. Basic hardware

The development environment hardware (figure 1) comprises an ARM microprocessor system with a separate FPGA accessible in its I/O space. The ARM processor connects to the host system via a serial line which can be used to download programmes for both the hardware and the software.



**Figure 2. Laboratory PCB**

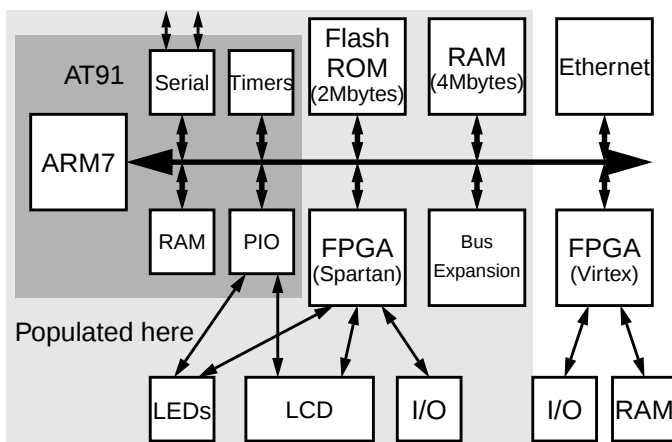
The devices chosen are the Atmel AT91 microcontroller [4] (32 MHz ARM7) and the Xilinx Spartan XCS10XL [5]. The processor provides a highly integrated, low-cost solution to the microprocessor system. Its performance is not high, but it is far more than adequate for the applications in mind. The FPGA is a small, cheap part but with a capacity of a few thousand gates is quite capable of holding an adequate variety of I/O interfaces simultaneously. The system is supported by a 2 Mbyte Flash ROM and 512 bytes of RAM.

The circuit board (figure 2) is capable of accepting more devices: the RAM can be expanded up to 4 Mbytes, an Ethernet interface attached and another, larger FPGA can be fitted to extend the environment to encompass larger student projects or postgraduate work; however for this laboratory a restricted, low-cost solution is preferred.

### 5.2. Software

A set of four DIP switches on the PCB allows the selection of one of sixteen possible software configurations following reset. Some of these are used for system support – for example one configuration is used to support an in-circuit ROM reprogramming utility and another is intended for self-test diagnostics – the others are available for users. Whilst ARM’s ‘Angel’ (the embedded software for their developer suite) can be installed it is not used in the laboratory described here for the reasons outlined above. Instead a purpose-built ‘back-end’ is used. This is designed to be robust and has a protocol which allows the FPGA(s) to be downloaded and can also support user data across the same link, whilst still providing the system monitor functions. A matching ‘front-end’ user interface (Komodo) is provided on the host system which allows the control and monitoring of the systems status, the downloading of software and hardware configurations, and the execution of user programmes (figure 3).

In order to provide the necessary control and reliability the students’ code is executed in a fully virtual environment running on the target processor (figure 4). This results in the slightly bizarre circumstance of a processor running an



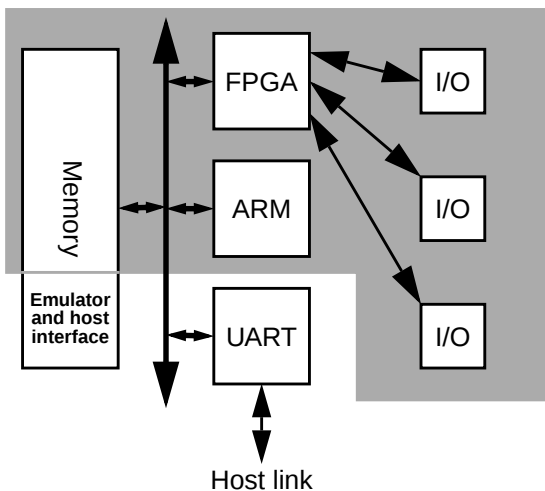
**Figure 1. Hardware configuration**



**Figure 3. User's view via Komodo**

emulation of its own machine code; however there are a number of advantages to this scheme:

- The students' view of the target machine can be completely 'clean'. None of the interconnection software is visible in the virtual environment. This also means the link cannot be damaged by target software crashes, which can be observed, single-stepped etc. just like any other software.
- The student can interrogate the target system at any time – including while programmes are running. This allows the display of registers and memory locations as they change.
- The simulator can simulate hardware which does not exist. Thus a software upgrade can extend the processor's architecture to the latest versions of ARM while still running on cheap hardware. If desired memory management, coprocessors, DSPs etc. could be added or even a different instruction set employed without changes to the hardware.



**Figure 4. Virtual ARM Environment**

- Sophisticated debugging features (breakpoints, watchpoints etc.) can be included easily.
- Simplified hardware interfaces can be provided whilst protecting hardware from potential damage (e.g. multiple different outputs driving the LCD bus).

The only real disadvantage of this environment is that the performance is much lower (about 0.5%) of the native performance. Although this sounds significant in practice it means that the target 'processor' can still deliver about 0.12 MIPS which is far more than is required for any of the intended experiments. (Due to the slow, 16-bit wide external RAM on the microcontroller, the performance delivered running code from the external RAM would be about 4 MIPS, so the emulator, in fact, internal RAM, actually delivers ~3% of the true performance.)

Considerable other software is used to support the laboratory. The assembler and linker from ARM's standard 'Software Development Toolkit' (SDT) are called from a script invoked from Komodo (although independently compiled programmes can also be loaded using the standard 'Executable and Linking Format' (ELF) [6].

Hardware design is supported using the Mentor Graphics design suite (although other tools are equally applicable) and compiled with Xilinx' tools. Komodo can pick up the bit file for downloading with a browser.

### 5.3. Interface expansion

It is obviously desirable for a piece of laboratory equipment to serve as many different roles as possible. Although the FPGA allows the digital hardware to be very flexible there is still an issue of the interfaces to external devices. For example a 'serial line' often refers to some subset of RS232 which requires buffering to particular voltage levels.

It is not always feasible to predict in advance what interfaces will be required. Instead a number of customised interface modules, with a common interface to the FPGA, are produced. Thus, for example, the FPGA can provide a serial interface at the voltage levels used by the digital circuits and a plug-in module can buffer these and provide a standard RS232 connector. This approach offers the added advantage of separating the parts most likely to suffer from physical or electrical damage from the more expensive PCB.

Possible interface modules include:

- Serial drivers
- Parallel port buffer
- Stepper motor drivers
- Analogue inputs/outputs

and others can be designed and added quite cheaply.

These expansion modules can be interchangeable so that not all the possible units need be fitted at any time. Each expansion connector occupies sixteen digital I/O pins on an FPGA; this is adequate for driving the parallel interface to a character LCD display, or four individual stepper motors, or an 8-bit DAC plus some bits from analogue comparators (allowing analogue to digital conversion algorithms to be explored) etc. If necessary an expansion module can be built which spans two or more I/O connectors.

## 6. Exercises

The exercises are primarily concerned with microprocessor interfacing. This is a subject not easily taught (practically) in any other way. Subjects covered include parallel and serial interfacing, timers and, of course, interrupt programming. Extensions into analogue interfacing are also planned.

The students already (allegedly!) have some background knowledge in that they will already have taken courses in both ARM assembly language programming and basic hardware design. Basic familiarity with the instruction set and the tools used is therefore assumed.

Exercises are intended to be short (normally intended to fit within one or two 2 hour sessions). Each exercise is intended to introduce one or two basic concepts with these ideas reinforced by later reuse. All the exercises are described in a laboratory manual which includes a large amount of tutorial and background material. Most exercises contain some suggestions for extensions so that the enthusiast can take things further. On the other hand there are numerous ‘side boxes’ which can be ignored if a student is getting behind.

Although the manual is substantial it deliberately omits a number of facts which may be very useful, if not essential. For example the LCD controller (HD44780) command set is deliberately not described; details are readily available on the WWW and students must also learn to seek the data they require themselves.

### 6.1. Basic experiments

The simplest form of output can be introduced with a simple ‘traffic lights’ programme which can be extended to accept user inputs. Initially this can use a simple delay loop which can later be replaced with a hardware timer.

Once simple I/O has been experienced more complex interfacing can be introduced. For example a subsequent session introduces the interface to a character LCD display (bidirectional parallel interface, strobing data) as a simple ‘Hello world’ programme (what else?). A later experiment uses the display drivers to implement a digital clock (timers, text formatting) and a further exercise places this under interrupt control.



**Figure 5. Predefined FPGA Configuration**

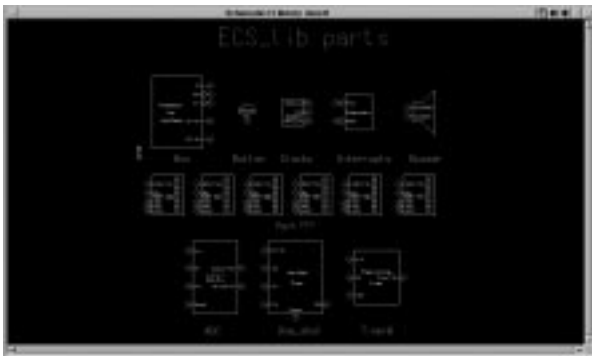
In this way the students also learn something of code reusability and the need for some documentation – probably discovering this the hard way, but this often conveys the best lesson. What is more, this incremental approach ensures that most exercises cannot be left half finished because they will be needed again. The ease of marking is a bonus for the laboratory supervisor.

A simple FPGA configuration (figure 5) is loaded at power on which is suitable for the early experiments. This default configuration is defined by the ‘boot’ option chosen on the board. The bit file is about 16 Kbytes long, so several different options can be supported, for example for other laboratories.

### 6.2. Other experiments

Once the basics have been mastered various more satisfying exercises are attempted. It is impossible to give an exhaustive list here, however here are some suggestions, some of which have been incorporated into the current syllabus:

- A piezzo-electric buzzer can be used to generate a programmable tone; timer interrupts control the duration of a note and a simple interpreter can generate tunes.
- An interrupt controller to help support increased system complexity.
- A stepper motor driver can be added to allow the control of external moving parts. This could give an added appeal to some exercises and lead into other, robotics-type courses. Extensions to illustrate control theory could be added if required.
- With an external DAC it is relatively simple to construct a successive approximation ADC, which can be controlled by either a hardware or a software algorithm.
- Serial communication can be introduced and used to link to another system (such as another port on the host). This forces compliance with an external standard. Both transmission (easy) and reception (harder) could be attempted.



**Figure 6. Extra FPGA Library Components**

- An infra-red communications module allows boards to communicate with each other remotely.

For many of these exercises the ‘best’ results can only be obtained by developing some hardware support. For example tone generation in software is crude – the emulator does not provide enough performance to give much resolution – and subject to ‘interference’ from interrupt service routines. A programmable hardware counter overcomes these problems. To assist with building these, students have access to the component library supplied with the Xilinx tools – which includes registers, counters etc. – and a local library (figure 6) which provides pin definitions for the circuit board and some more specific assistance.

It is also possible to combine several of these exercises in a modular fashion to form ‘mini-projects’. For example a regular clock interrupt could be combined with an analogue conversion exercise (e.g. speech sampling) and with a communications exercise to produce a form of telephone!

## 7. Status and Observations

At the time of writing the laboratory has just run for the first time. The first generation circuit boards are in use and have so far proved thoroughly reliable; a second generation system is likely, but this is intended to provide a more modern, cheaper ‘large’ FPGA option (the *Virtex-E*) and will not affect the laboratory described. Enhancement via new ‘plug-in’ interface modules will be a continuous process with small incremental costs.

Software development is ongoing, but the first generation system has proved the concept perfectly feasible. The emulator is reliable and adequately fast and the host interface is robust enough to survive the abuse of inexperienced users without crashing.

Future developments will include some more streamlining of the embedded code, support for some more on-board devices, and improvements to the ‘look-and-feel’ of the front end, for example making it easier to customise

to the user’s satisfaction. It is intended that ‘source-level’ debugging will be incorporated in future.

Although the hardware and software have proved largely ‘right first time’, the exercises still need further development. Whilst the students raced through some others caused severe consternation, for no readily apparent reason. The first major check occurred when servicing system calls, an exercise requiring little more than the setting up of a system stack and an understanding of ‘user’ and ‘supervisor’ modes, all of which was taught in a preceding course. Clearly relying on material from the earlier ARM course – only months before – was a bad idea. Such exercises will be broken into smaller steps in future.

As this course has only been running for one year, and therefore student numbers are small, student feedback is necessarily limited. The students’ response to the course appears largely positive, attested to by the high attendance at laboratory sessions even in the absence of strictly imposed deadlines. Progress has clearly been checked by ‘teething troubles’ with the flow of exercises; while there will be ‘tweaking’ we intend to continue the course ‘as is’.

## 8. Conclusions

It is believed that a real, physical environment for real time development is the best way for a student to learn about a real-time environment. The system described here provides precisely that, whilst retaining a very simple user view of the system, allowing concepts to be introduced and reinforced gradually. The low cost and user expandability of the circuit board and the software provides a customisable interface for the exercise designer. It is hoped that this will provide a useful tool for universities for some years to come.

## 9. Acknowledgements

The author wishes to acknowledge the contribution of Charlie Brey in the production of the Komodo software.

## 10. References

- [1] S.B. Furber, “ARM System-on-Chip Architecture”, (Addison Wesley Longman 2000) ISBN: 0-201-67519-6
- [2] <http://www.europractice.com>
- [3] <http://www.altera.com/literature/lit-nio.html>
- [4] <http://www.atmel.com/atmel/products/prod35.htm>
- [5] <http://www.xilinx.com/partinfo/databook.htm>
- [6] Tool Interface Standards (TIS) Committee  
“Executable and Linking Format (ELF) Specification”  
(<http://x86.ddj.com/ftp/manuals/tools/elf.pdf>)