Embedded Systems Lab

The SpectrA Assignment 2010/2011

S.D Cotofana, T. Slats, X van Rijnsoever

Name student : _____

Studentnumber : _____

Delft University of Technology
Faculty of Information Electrical Engineering, Mathematics and Computer Science
Computer Engineering Group

September, 2010

Preface

In embedded system design, the challenge lies in determining a good mixture of hardware and software components and, of course, the design of such components. Furthermore, the designer must deal with design constraints such as low cost, high performance, short design time, and low power consumption.

The current lab assignment is setup such that the student can experience the embedded system design by mainly focusing on the performance design constraint. In particular, the student will face several difficulties/challenges when a software-only implementation must be sped up by utilizing software techniques and by shifting certain components' implementation from software to hardware. The goal of this assignment is to allow the students to learn what embedded system design is and to allow them to use their creativity to realize competitive designs. We hope you will enjoy it as much as we did when setting up the lab.

Finally, if you have questions or feel that something is missing in this guide, please don't hesitate to tell us! We always appreciate your opinion about this handbook and the lab in general. For questions about the organization of the laboratory you can contact **Ton Slats (A.M.J.Slats@tudelft.nl, 015 27 88787)**.

Regards,

Sorin Cotofana

Ton Slats
Xavier van Rijnsoever

Contents

1. Project description

1.1 Introduction

On current mp3 players' displays (projected on a computer screen or on an LCD screen) most of us might have seen those funny bars bouncing up and down. The displayed information relates to the amplitude of certain frequencies (or frequency bands) of the music that is being played at that moment. The calculations needed to display this spectrum related information can be considered rather complex. However, given that older audio (visual) equipment (such as an old VCR or old HIFI equipment) also can display such information, the required calculations can be performed by utilizing non-'state-of-the-art' processors.

The actual device responsible for the calculations and displaying the frequency (band) information is called a spectrum analyzer. Generally speaking, the steps required to converting analogue audio signals into their corresponding frequency spectrums and finally displaying the frequency spectrums can be summarized as follows:

- First a number of (*N*) samples of the analogue audio signal are read and digitized.
- Once digitized the frequency information can be acquired by computing a fast Fourier transform (FFT) over the *N* samples.
- To represent the spectrum, the amplitudes of the different frequency elements belonging to one frequency band should be averaged.
- Finally the average values of each band should be displayed.

1.1 Goal of the lab

In this lab, you will have to design a spectrum analyzer. It should be able to accept an analogue (electrical) audio signal and to display its corresponding frequency spectrum on an LCD display. The frequency spectrum is divided into 16 bands of equal width and each band can have different amplitude values from 0 to 16.

The spectrum analyzer should be fast enough to have only a small unnoticeable delay between the arrival of input audio signals and the displaying of the (corresponding) frequency spectrum. In the context of this lab only the performance requirement is considered and all other issues like power consumption, cost and size, although important in a real life embedded system design, are not to be considered.

1.2 Design constraints

Normally, an embedded system designer has complete freedom to choose a suitable combination of hardware and software development environments. In a "real-life" design scenario he/she would have to optimize cost, performance, time-to-market, etc. However, in this lab you don't have to do so. Our concern in this matter is to merely optimize performance given the use of some predefined hardware and software components. The language in which programming is performed is C, which constitutes the foundation for C++ and in a sense for JAVA.

Briefly summarized, the following HW & SW development platform should be utilized in the context of the embedded systems lab:

- 80C552 microcontroller board and associated software suite including a C compiler and a Windows based shell.

- UP1 board containing an Altera FLEX10K20 FPGA and Quartus II software suite.

Given this context, you are asked to design the spectrum analyzer (called SpectrA) by following the embedded system design trajectory described in the course textbook. Apart from that, you have to exercise co-design experiments, that is try to improve the system's performance by moving time consuming operations from software to hardware. Don't forget that you should be able to implement your Embedded System (ES) under the restrictions imposed by the development platform you have to make use of in the context of this ES design experiment. Therefore, solutions that cannot be implemented on the Altera FLEX10K20 chip have no value!

In order to complete your design, you should provide solutions to all the assignments presented in Chapter 3. Please note that this is not the only way in which such a system can be designed. Although other alternatives may exist we stick to the provided assignment list This enables the student assistants to keep track of the evolution of your work, check it, and help you when needed.

Apart from the assignments in Chapter 3, Chapter 2 of this handbook contains a description of the hardware and software development platform. To make the lab handbook self-contained we have also included documentation about the 80C552 microcontroller board and some documentation related to the software suites Quartus II and WinUptools.

We hope you will find the project interesting and have a lot of instructive fun in solving it.

Good luck!

2. The Laboratory Environment

In this chapter, we give a short overview of the available hardware and software you will have to utilize for this ES design lab exercise. When you need more information about the software or hardware, you can find it in the appendices, the online help or on the Internet.
On the PC's a small collection of digital tutorials will be available. In the following section, a description of the available hardware is given. The second section gives a brief overview of the software available for this practical course. The last section gives some notes on the available documentation .

2.1 Hardware Environment

The hardware environment of this project (depicted in Figure 2.1) consists of a PC running Windows, a FPGA board, and an 80C552 board with an LCD. The different hardware components are described in the following subsections.

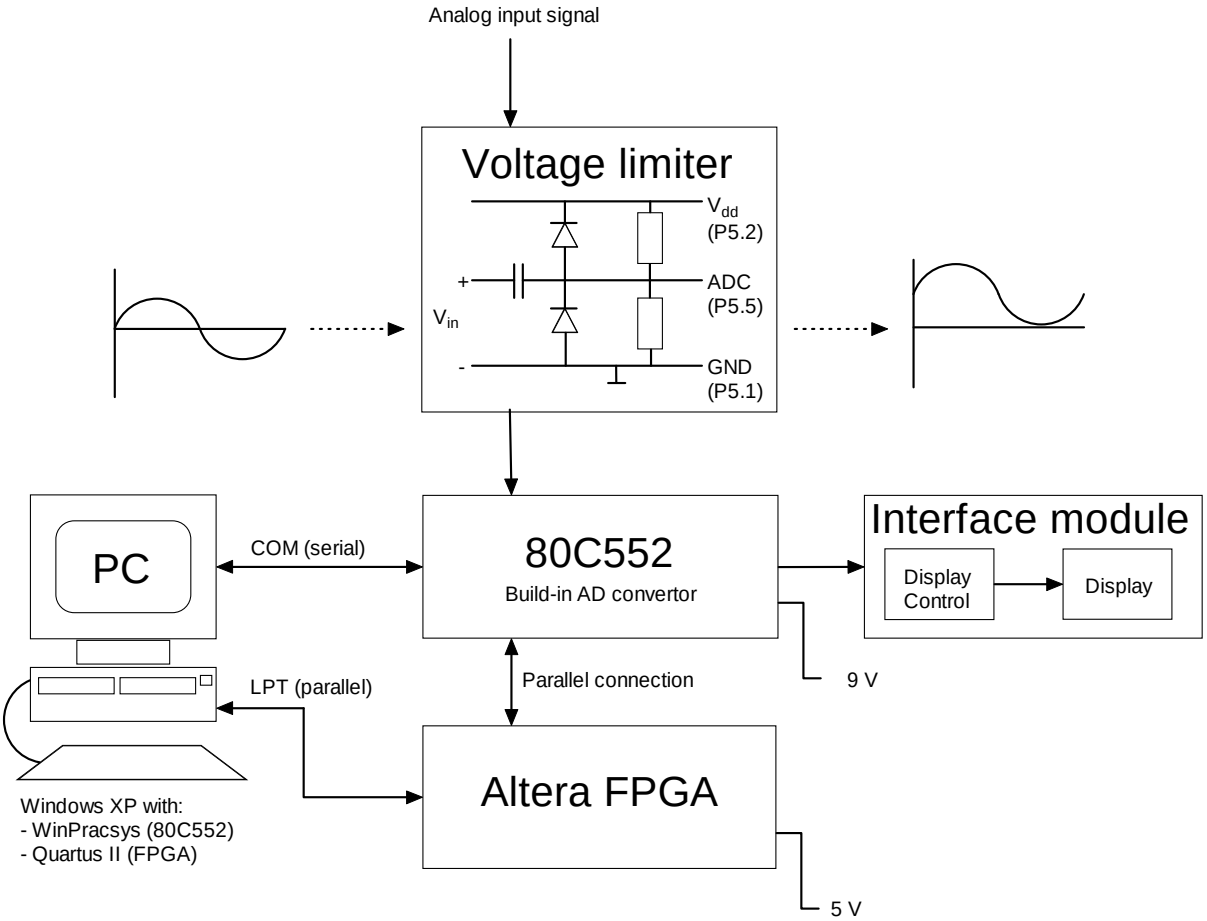


Figure 2.1: ES development platform.

2.1.1 The PC

On the PC a shell is running: WinUptools. This shell enables you to program the 80C552 board. It controls the C compiler, assembler, and linker. Furthermore, the shell can upload the compiled program at a specified address in the memory of the 80C552 board. This is done via a serial cable on the COM port.

In addition, the program Quartus II is installed. This program can be used to compile VHDL code and to program the FPGA chip on the UP1 board.

2.1.2 The 80C552 Board

The 80C552 board consists of several main components from which we mention only the ones that are relevant in the context of this lab. The main component is the 80C552 processor itself. A schematic view of its architecture is depicted in figure 2.2 below.

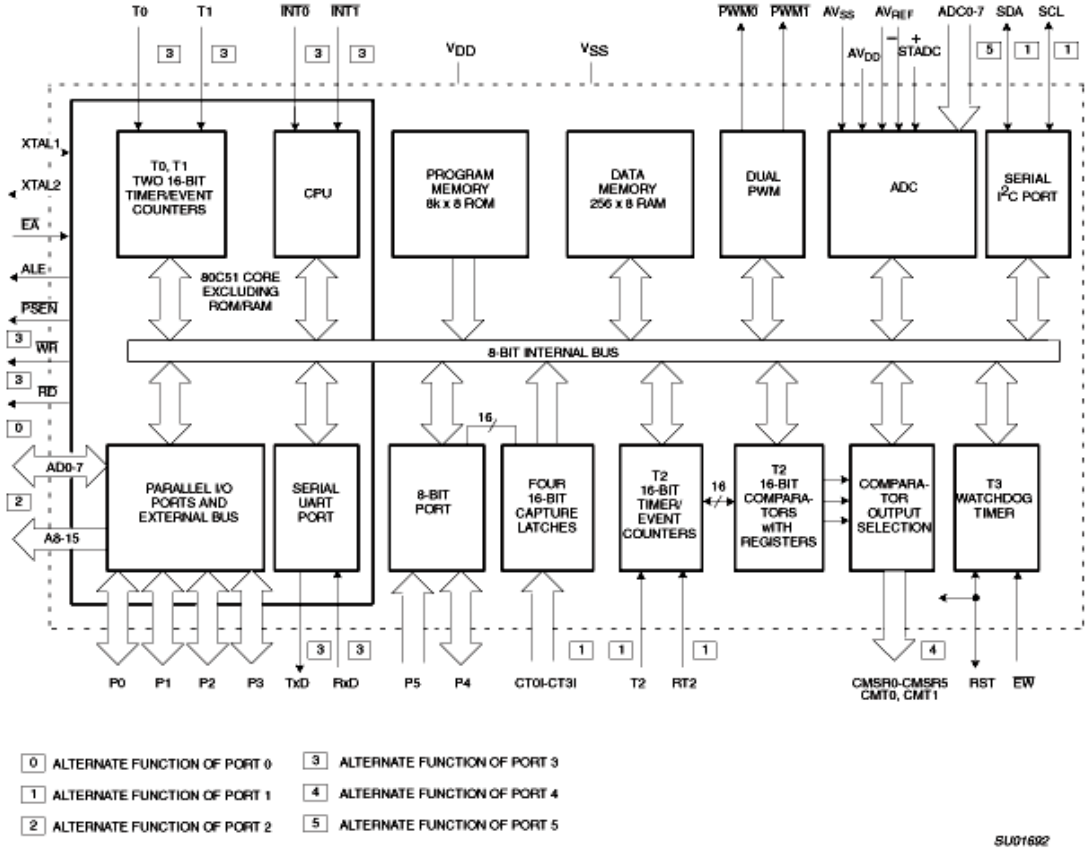


Figure 2.2: Schematic view of the 80C552 board

Additionally the board includes:

- A 8kb SRAM chip. In this SRAM, both data- and program memory are located.
- Several ports which can be used to send data to the display and FPGA and to receive data from the soundcard, FPGA, and PC.
- A reset button, of course, which needs to be pressed before the board can be programmed.

Just like a normal PC development environment, the 80C552 board needs a way to provide the user with debug information and such information is provided in our case by the VMON program running on the microprocessor. This program manages the reception of data from the PC, the display control, and so on. Addresses 0 - 1FFF of the SRAM are reserved for the VMON program.

2.1.3 Interface Module

The interface module is designed to connect an alphanumeric display and keyboard to the 80C552 microcontroller. Utilizing these peripheral devices, it becomes possible to design

applications that can be controlled by the keyboard, while output data can be presented on the display. In this lab, we don't utilize the keyboard, because the nature of the considered application doesn't require the user to provide any input data except for a reset signal. On the other hand, the LCD module is used to visualize the output of the SpectrA. The LCD module (HD44780) has 2 lines with 16 characters each. This should be adequate for most text oriented applications, however it isn't ready yet for your design. This is because you would like to display graphical and not alphanumeric information. Fortunately, the LCD module supports programmable user defined characters, thus you can display the bars associated to the SpectrA system by utilizing some special characters.

Display controlling routines

Programming the LCD-display is straightforward, but it can be very time-consuming. Therefore, you are given some procedures (written in C and assembly) that can be used to control the display, as well as a procedure that creates the special characters that are required in order to display graphical information. Those special characters make it easier to write a program to display bars.

The file that contains these procedures is `LCD.A51`. This file should be added to your project when you want to make use of the functions described in the table 2.1. Furthermore, the header file `lcd.h` should be included in the file where you want to make use of the display functions.

Table 2.1: Special display functions in `LCD.a51`

Procedure Name	Description
`void display_init(void)`	Initialize the display. First it puts the display on, turns the cursor off and disables the blinking of the cursor. After that the special characters (to display the bars) are created.
`void display_clear(void)`	Clears the display and puts the cursor in the upper-left corner.
`void display_return_home(void)`	Puts the cursor in the upper-left corner.
`void display_next_line(void)`	Puts the cursor on the lower-left corner.
`void display_putchar(char)`	Displays a character on the display.

Special Characters

In Figure 2.3 you can see the different characters that are created with the function `void display_init(void)`. To use these characters you should call the procedure `void display_putchar(char)`. The argument of this procedure should be the number of the special character you would like to use. This value ranges from 0 to 7, and follows the numbering convention in Figure 2.3.

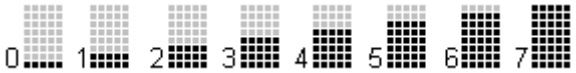


Figure 2.3: Special characters for the bar elements

The space character (ASCII character 20h) completes the full range of characters to be utilized to display the bars required for the spectrum analyzer.

Example code

To demonstrate the bar display procedure we include an example program. This program displays the special characters in a certain way and the result of this program can be seen in Figure 2.4.

```
#include “lcd.h”

void main()
{
	int i;

	/* initialize the display and create the special characters */
	display_init();

	/* this for-loop displays the first 8 characters on the first line.*/
	for(i=0; i<8; i++)
	{
		display_putchar(i);
	}

	/* this for-loop displays the last 8 characters on the first line.*/
	for(i=0; i<8; i++)
	{
		display_putchar(7-i);
	}

	/* now it puts the cursor in the lowerleft corner.*/
	display_next_line();

	/* this for-loop displays the first 8 characters on the second line.*/
	for(i=7; i>=0; i--)
	{
		display_putchar(i);
	}

	/* this for-loop displays the last 8 characters on the second line.*/
	for(i=7; i>=0; i--)
	{
		display_putchar(7-i);
	}
}
```

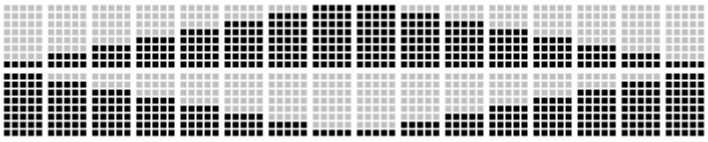


Figure 2.4: The output of the example program

2.1.4 The Altera FPGA Board

The most important piece of the FPGA board is the Altera FLEX10K20 FPGA chip itself running at 25 Mhz. We must note that this FPGA chip has a volatile character, therefore, it requires reprogramming when the power supply has been disconnected. Furthermore, two ports are used to send/receive data to/from the 80C552 microcontroller board.

The interface between the 80C552 microcontroller board and the FPGA chip is depicted in Table 2.2. Note that not all pins are used and that the internal pin assignment is different from the physical pin layout.

It should go without saying that care has to be taken while handling the hardware. The board has no cover so all components are out in the open and very sensitive to student fingers!

Table 2.2: The interface between the Altera UP1 and the 80C552 microcontroller board (pin assignment)

Port 1				
Altera UP1 dev. board		80C552 board		
FPGA	FLEX EXPAN A	PORT 1		
		Bit	Pin	
56	24	7	10	reserved
55	23	6	9	reserved
54	22	5	8	
53	21	4	7	
51	20	3	6	
50	19	2	5	
49	18	1	4	
48	17	0	3	
46	16	n.a.	2	VDD (+5 Volt)
45	15	n.a.	1	GND

Port 4				
Altera UP1 dev. board		80C552 board		
FPGA	FLEX EXPAN A	PORT 4		
		Bit	Pin	
78	40	7	10	
76	39	6	9	
75	38	5	8	
74	37	4	7	
73	36	3	6	
72	35	2	5	
71	34	1	4	
70	33	0	3	
68	32	n.a.	2	VDD (+5 Volt)
67	31	n.a.	1	GND

2.2 Software Environment

To program the hardware you will make use of software that runs on a Windows XP platform. For the 8051-development board a program called WinUptools has been developed. You will use the Quartus II suite to program the Altera UP1 board.

In this section we give a brief introductory description of both programs. The most attention is given to WinUptools, because the tutorial for Quartus II software can be found on the Altera website. In Subsection 2.2.1 the program WinUptools is explained. Subsection 2.2.2 covers the basics of Quartus II.

2.2.1 WinUptools

WinUptools is especially developed for this lab and it is a Windows port of the DOS PracSys 80C552 development environment (also developed at the TU Delft). WinUptools is actually an easy-to-use interface to invoke the programs you have to use to develop software for the 80C552 development board. These programs are from the company Keil, a well-known supplier of embedded development software.

The main reason why we developed WinUptools was to speed up the design process. One of the things we try to save you from is the trouble of entering long commands at the DOS-prompt. Because WinUptools is still under development the features are subject to changes. We hope we can (during the lab) adapt the program to your needs. Of course there is one drawback, some unpleasant bugs may still exist in the version you'll be using. When you encounter one, please mention this. We can then try to fix those problems. In the next subsections the WinUptools environment is explained.

The WinUptools interface, as it appears directly after the environment has been started, is depicted in Figure 2.5.

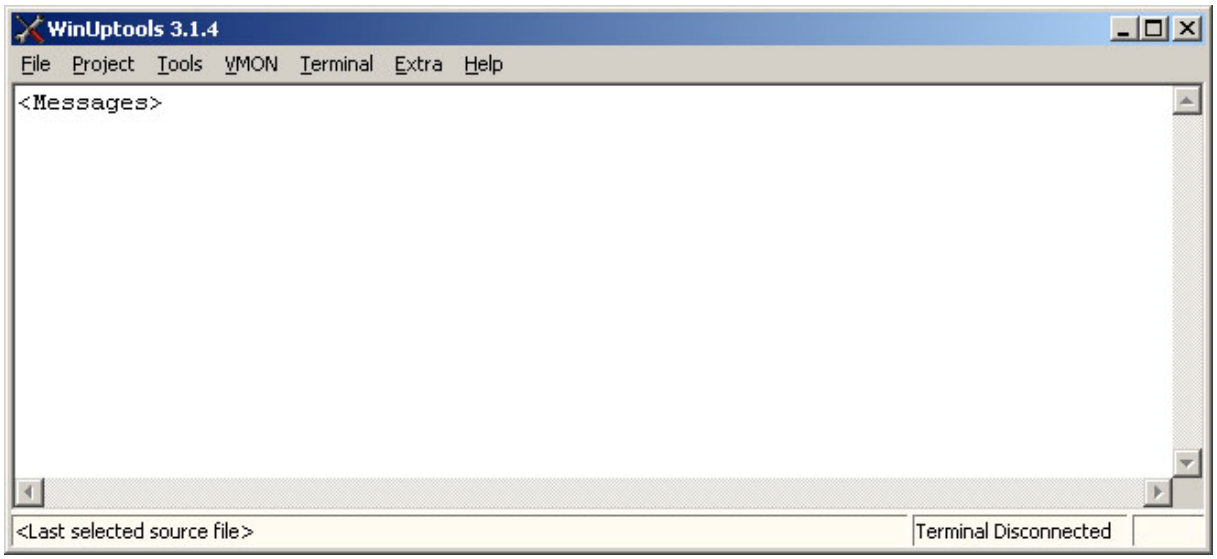


Figure 2.5: The WinUptools main screen

To make WinUptools working you have to copy the map WinUptools3 from the directory J:\ewi\Courses\ET3115D2 in your own Personal Data directory (H:)The WinUptools program is located in the WinUptools start directory. Be sure that project files and source files can only be located in the WinUptools start directory or in one of its subdirectories, preferably in the subdirectory 'projects' a subdirectory of 'projects'.

- The command **View Project** opens the project files window. In this window source files, object files and header files can be added to the project. Specify the file type when importing this files. Make your first program by using the directory Framework with the project 'SpectrA' this project can also be found in the map ET3115D2. Copy it into the WinUptools\projects directory. Look in the header file SpectrA.h and write your c code in Spectra.c.
- The command **Project -> Build Project** (F9) compiles assembles, and links the whole project.
- The command **Terminal ->Terminal Window** (Crtl+W) split the main window into a message window and a terminal window.
- The command **Terminal -> Connect Terminal** (Crt+T) open a serial connection between the WinUptools and the Microcontroller board.
- To run a program on the microcontroller board use the **VMON** menu and the **Upload and Run** (F6) command.

We note here that sometimes the linker message is not specific enough. You can find more information about linker messages in the *.M51* file (near the end of that file).

2.2.2 Quartus II

In this subsection we will cover some basics of Quartus II. An extended manual is available on the PC you will utilize for the lab, so when you need more information you can find it there. Additionally we'd like to mention that you can download a web version of Quartus II at the Altera website (http://www.altera.com/).

When you start working with Quartus II you'll have to set your project. This can be done with the '`New project wizard`…' under the '`File`' menu. First you have to select a working

directory. Make sure you pick a directory that is empty. You also have to specify the name of the toplevel entity.

In the second dialog you can add files to your project. These can be VHDL entities that you want to use in your design. Please keep in mind that a large component library that includes many standard hardware components is also available.

In the third dialog you can specify third party software that you want to use for simulation, synthesis, etc. Usually you can leave these options unspecified, in that case the default Quartus II tools will be used.

In the following dialog it is possible to specify the target device family. This will be the FLEX10K. Answer `yes` on the question "`Do you want to specify a specific device?`". In the following dialog it is then possible to select a device type. This should be the FLEX10K20RC240-4 or the FLEX10K70RC240-4 (check the Flex-chip on your board).
The project is now configured. You can now click '`Finish`' (or '`Next`' for an overview of the selected options).

In this lab you will mainly utilize the following functions of Quartus II:

- VHDL-editor
- Schematic design interface
- Compiler
- Simulator
- Programmer.

These functions can be found under the 'processing' or 'tools' menu. Quartus II also contains a syntax-highlighting editor for VHDL files and a schematic design interface. This schematic design interface can be used to connect VHDL entities together on a structural level.

VHDL-editor

The build-in VHDL-editor of Quartus II is a syntax-highlighting editor.

Schematic design interface

In Quartus II you can easily create your logic design from basic building blocks or blocks created by yourself by connecting them in the schematic design interface.

Using the schematic editor will save you a lot of time.

Compiler

The compiler generates the necessary design files to program the FPGA. Before you start the compilation you have to complete the pin assignment procedure. This can be done with the option '`Assign pins…`' under the '`Assignments`' menu item.

During the compilation process you can see the different steps the compiler goes through. During every step messages are generated. These can be hints, warnings, and errors. On encountering errors the compiler won't start with the next phase. These messages can be pretty difficult to understand, but a much more useful description can be found in the build-in help. To bring up this help-function, select the error-message and press the F1 key. After correcting (some) of the errors given by the message window, you can recompile the project.

Use the help-function before asking assistance!

When the compilation is complete, you can simulate your design or program it into the FPGA.

Simulator

The simulator runs a simulation of your design by using a waveform as the input file. To create a 'Vector waveform file', you can use the menu item '`New file`'. The name of this file

should be the same as that of the top entity. In the Wave Form Editor you can also view the simulation results generated by the simulator.
The Waveform editor is best suited for sequential and repeating signals. The editor allows you to copy, cut, paste, repeat, and stretch waveforms.

Programming

Programming the FPGA is very simple. You just choose the programmer from the '`Tools`' menu, select the '`Program/Configure`' checkbox behind the file you want to program and click on '`Start Programming`'. After the programming phase, the FPGA is programmed and ready to use.

2.3 Documentation

In order to be able to finish this lab, you will have to use several software programs, implement algorithms and familiarize yourself with the hardware. The information you require for this lab can be found in the provided documentation. However, this information is distributed over several documents. It is up to you to find out which parts are useful for your implementation.

2.3.1 Appendices

This manual has several appendices that contain (abstracts of) manuals for the programming tools, descriptions of algorithms and technical information on the hardware. In the appendices you can also find an introduction to programming in C and an overview of the provided VHDL building blocks.

2.3.2 Online documentation

During the lab, full versions of several manuals are available on a server. You can find the serverfiles on the T-drive under the code of this course.

Of course, the internet is valuable source of additional information too. The information provided in the appendices is sufficient for this lab, but additional information can lead to a better design.

3. The Lab Assignments

3.1 Introduction

The lab project is divided into several kinds of assignments:

- Homework
- Lab tests
- Practical assignments

You will have to do all these assignments to successfully pass this lab.

3.1 Homework

The homework exercises must be completed ***before*** starting with the lab assignments. The main purpose of the homework exercises is to make sure you understand the material required for the practical assignments.

3.2 Lab tests

The lab tests are handed out at the beginning of the lab session. The tests consist of a number of multiple-choice questions. The lab tests must be completed before you start working on the practical assignments. The results must be discussed with one of the student assistants.

The purpose of these tests is to point out the most common pitfalls that can occur in the practical assignments.

Doing these tests will save you a lot of time with the practical assignments

3.3 Practical Assignments

The practical assignments are the “real” work. In this section you will find the steps you have to follow in order to complete the ES lab design. The assignments will guide you through the design process. After you have finished an assignment, you should check your result with the student assistant. Of course you can try to solve the next assignment when the assistants are busy and cannot check your solution immediately, but you still must check it as soon as possible.

Experience has shown that this lab course has a tight schedule. The schedule is shown in Table 3.1. You have to keep up with this schedule! You can only do this by doing the homework exercises at home and reading the assignments before coming to the lab.

Table 3.1: The nominal project schedule

Session	Homework assignment	Practical assignments
1	Homework 1	1 and 2
2	Homework 2	3 and 4
3	Homework 3	5 and 6 (+ optional assignment)
4	Homework 4	7
5		8 and 9
6	Homework 5	10
7		11 and 12 (+ optional assignment)

Software layout

The assignments provide you with function prototypes. Although you are allowed to deviate from these prototypes, it is highly recommend using them. If you choose to deviate from the conventions, you should take into account that the student assistants can't help you with debugging your code.

Also try to keep a clean programming style and create regular backups of your files.

Note that your files are ***not*** kept on the computer you're working on!

Constants

To allow easy changes to the number of samples your implementation uses, you should use the constants defined in Table 3.2.

Table 3.2: Constants used in the software implementation

Constant	Meaning
N	The number of samples used in a cycle
LDN	$\log_2(N)$
PI	3.14159265359

Global datastructures

To store the samples on which the computations are performed, you have to use an array of complex numbers. A complex number is not a standard C data type, so we define it as follows:

```
typedef struct
{
  long r;         /* the real part of the complex number */
  long i;         /* the imaginary part of the complex number */
} complex;
```

Furthermore, the array has to be defined globally as `complex xdata samples[N];`

The `xdata` directive tells the linker that we should place the array in the extended memory. This is necessary because the internal memory of the chip is too small. Unfortunately accessing the external memory is slower than the internal memory.

Hardware layout

The hardware can be built out of several available components. All the components available from the standard library of Quartus may be used. In addition some components are provided during the lab. A description of some of these components can be found in Appendix F. It is allowed to write your own components to suit your needs. There is one condition; you must write the component yourself. In all other cases you should consult the student assistant if it's allowed to use the component.

Session 1, Band processing and Data acquisition

Homework 1, Assignments 1 and 2.

In this session, you will have to think about the global structure of the SpectrA. The system consists of several blocks that are ordered in a certain way. You will have to think about the functionality the blocks should provide and the way they are ordered.

Once the global structure and the functionality of the blocks are clear, you will start implementing the blocks. The blocks must be implemented as separate modules. This allows you to test each block independently from the other block.

To test these blocks, you will have to create a small (specific) test program. This program should only contain the block itself and a main-function that calls the block with some relevant input data sets.

Homework 1: System requirements and description

Based on the problem description in Chapter 1, derive the spectrum analyzer requirements and create a block diagram of the Spectrum Analyzer. Let the student assistant check your requirement list and block scheme diagram before you start with the first assignment.

Hint:

- The block diagram should contain the components you have to implement for the software implementation as you can find in the assignments (there are four main blocks).

Assistant signature:

Assignment 1: Band processing module

Design and implement in C the band-processing module. The function should have the following prototype:

```
void band(void)
```

The function operates on the values in the `samples[]` array and the output should be a set of 16 values. Think carefully where you should place these values and discuss your opinion with the student assistant.

Hint:

- The values for each band should range from 0 - 16.

Assistant signature:

Assignment 2: Data acquisition module

Design and implement in C the data acquisition module. The sampled values should be placed in the `samples[]` array. To test your module, you should display the sampled data on the PC terminal. The function to start the sampler should have the following prototype

```
void AD(void)
```

The function should exit when the samples array is filled with 256 samples.

Hints:

- The A/D converter is 10-bit wide, but you should only use the 8 most significant bits.
- To read the data samples you have to use the interrupt generated by the A/D-converter.

Assistant signature:

Session 2, FFT and Display

Homework 2, Assignments 3 and 4

In the previous session you've implemented and tested two of the four building blocks of SpectrA. In this session you will have to create the other two blocks, the FFT and the display module. The FFT is the main procedure of the SpectrA and is a large and difficult routine. To make things easier, you can use the algorithms provided in appendix D.

The homework assignment will ask you to select one of the different implementations of the FFT. You will also have to draw a flowchart for the display module. When you've discussed your choice and the flowchart, you have to implement the routines in C.

Test these routines with a small test program. To verify that your program works correctly, you can use the program `Matlab` and compare the results of your FFT with the results of `Matlab`. You can use the following command in `Matlab` to print the results of the FFT of the numbers 1 to 16:

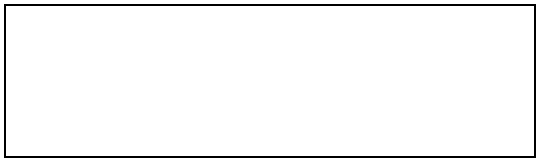
```
fft(1:16)'
```

This command will print a table of results of the FFT of the numbers 1 to 16. When you make your test program output the results in the same way, you can easily compare them.

Homework 2: FFT and Display module design

Before continuing with assignment 3 and 4 you should finish the following homework assignments. Make sure that before you start implementing assignments 3 and 4, the student assistant has verified your homework.

- Find for the next assignment the FFT algorithm you want to use. Some algorithms can be found in Appendix D. It is wise to choose a radix-2 algorithm (a higher radix is more difficult to implement). Explain why you have chosen a recursive or iterative algorithm.
- Make a flowchart for the software of the display module.

Assistant signature: 

Assignment 3: FFT module

Design and implement in C the FFT module. The module should be parametrical, so you can feed the FFT with `LDN` (with `LDN` = $\log_2$(`N`)) samples (see Table 3.2). You have to test it with `N` = 16 samples, with their values increasing from 0 to 15. The FFT module should contain the following function prototypes:

```
void fft(void)
void revbin_permute(void)
int revbin_update(int)
void swap(complex*, complex*)
```

Remember to create a separate project to test this function.

Hints:

- Optional: To speed up computations you can use a lookup table that contains the values of the `sine` and `cosine` functions. The values of the `sine` and `cosine` functions are between -1 and 1. So if you want to use an integer lookup table, you should scale them up. Remember to scale them down after the

multiplication! Note that you only need to use half a period and that the lookup table is not parametrical!. In the textbook you can find an example on page 202.

Assistant signature:

Assignment 4: Display module

Design and implement in C the display module. You have to test it by displaying all possible bars on the LCD display. Make sure your display module can correctly display all levels from 0 to 16.

Hints:

- You can use the display routines in Chapter 2.1.3.
- Use the special characters, which can be created with the procedure `void display_init(void)`.

Assistant signature:

Session 3, SpectrA v1 integration

Homework 3, Assignments 5 and 6 and optional

In the last two sessions you've implemented the building blocks of SpectrA, now it is time to put it all together and create the first implementation of SpectrA.

Before you can do this, you should first think about the flowchart required for the `main` function. This has to be done as homework assignment 3. When you're flowchart is checked by a student assistant, you can implement it. Test the application by using the signal generator as an input device. Think about what you should expect as output with a certain input selected at the generator. Also think what should happen when you increase or decrease the frequency.

If SpectrA is working correctly, you have to measure the speed of your implementation. If you have time left, you can try to increase the performance of SpectrA using several speedup methods.

Homework 3: SpectrA v1 integration flowchart

Now that all the modules are completed they should be integrated as a whole. There is a main component that controls the modules. This main component is implemented as the function **`void`** `main(`**`void`**`).` Design the flowchart for this function.

Assistant signature:

Assignment 5: SpectrA v1 integration

Integrate the already implemented modules into one system. SpectrA v1 has to be able to process blocks of 256 samples. To test the SpectrA v1 you can use the signal generator.

Hints:

- Create a new project called SpectrA_v1. Copy all necessary files in the new directory.
- Pay attention to data communication between the modules.
- Don't forget to comment out the code you wrote in order to independently test the modules.

Assistant signature:

Assignment 6: SpectrA v1 performance evaluation

Evaluate the performance of SpectrA v1 by measuring the time required by one SpectrA cycle and one FTT computation. Discuss the result with the student assistant.

Hint:

- To measure the execution time you can set an output pin just before executing the FFT function. Reset the output pin when the FFT function is finished. Now you can use an oscilloscope to measure the execution time.

Assistant signature:	
SpectrA time: FFT time:	

Optional: Increasing performance via software techniques

Investigate some alternative ways to increase the performance of SpectrA v1 via software techniques. You do not have to implement all your ideas, just discuss them with the student assistant. Implement the most promising ideas if you (and the assistants) believe you have enough time to complete the rest of the assignments too. Remember, it's only good to be the fastest, if you reach the finish.

Assistant signature:	

Warning:

- Don't spend too much time optimizing your design. Don't forget that there are other assignments waiting.

Session 4, Hardware multiplier

Homework 4, Assignment 7

Now you've got a working version of SpectrA, you'll have to try to increase the performance by implementing the most time-consuming parts in (faster) hardware. The first implementation will use a hardware multiplier.

In the homework assignment you're asked to design such a multiplier. Think carefully about the way the communication should work. When a student assistant has checked your work, you can implement it in VHDL using the Quartus II software. Note that you can find an example project in Appendix G. You can use this example to get familiar with the Quartus environment.

When your implementation works correctly in the simulation, test it with the 80C552 board. Create a separate test program to test the hardware multiplier.

Homework 4: Hardware multiplier design

To improve the performance of the SpectrA you can implement some of the software parts in hardware. First the multiplication will be done in hardware. Design a 16-bit multiplication hardware facility. Make a block diagram of the hardware design you proposed.

Your design should include the following parts: a part for the communication between the 80C552 board and the FPGA board, the 16-bit multiplier itself, and a control unit (FSM).

Before you start thinking about the communication protocol, read Section 2.6 of appendix A. This section describes the way the I/O ports of the 80C552 behave when they're connected to another device. For the hardware design there are some components available. A description of these components can be found in Appendix F.

Hints:

- The easiest way to set up communication between the 80C552 microcontroller board and the FPGA is with a handshaking protocol. Try to keep things as simple as possible.
- In table 2.2 in Subsection 2.1.4 you can find the description of the cable you have to utilize in order to connect the two boards.

Assistant signature:

Assignment 7: Implementation of the hardware multiplier

Implement in VHDL a 16-bit multiplication hardware facility (MUL16). Simulate your hardware design to verify its correct operation. Synthesize your design on the UP1 board. Use the FLEX10K device. Implement in C a function that can communicate with your hardware multiplier. This function should have the following prototype:

```
long mult16(long, long)
```

Make a test program to test your hardware multiplier.

Assistant signature:

Session 5, SpectrA v2 integration

Assignments 8 and 9

In this session you're going to integrate the hardware multiplier you've created in the previous session in a new version of SpectrA. Because you're going to edit a working version of SpectrA, make sure you create a backup of your original version.

When the integration is done, it is time to measure the performance of this new version of SpectrA. Since all the ports are used, you will have to use pin 1 of jumperlocation J5 on the interface module. You can write to this pin using the label `INT1`. You can use pin 1 of the I^2C-bus as GND. The locations of these pins are shown in figure 3.1.

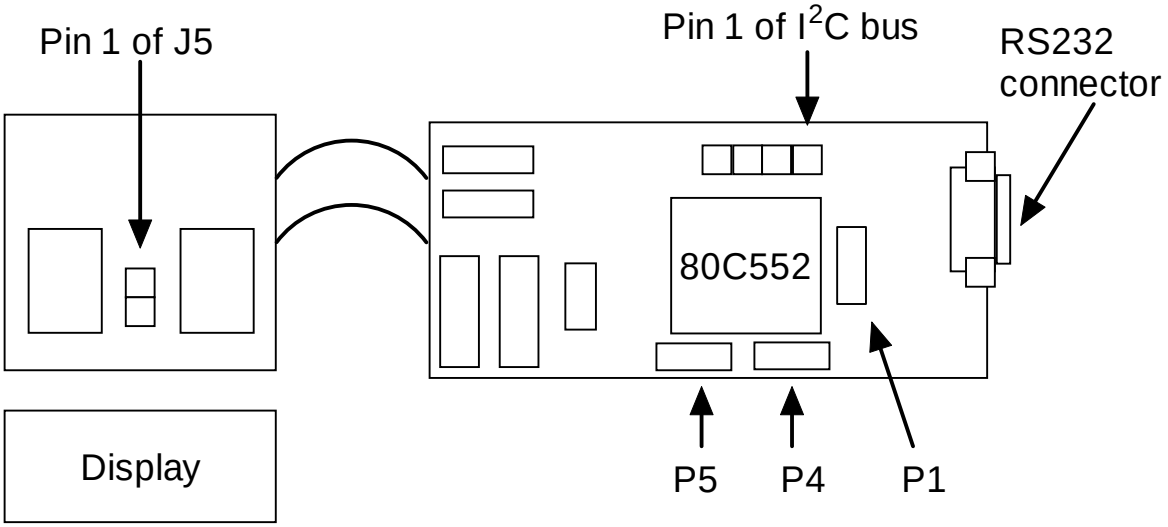


Figure 3.1: Pinlocations needed for measuring the speed of SpectrA v2

Assignment 8: SpectrA v2 integration

Rewrite the FFT module (backup the SpectrA v1 first!) in order to make use of the MUL16 hardware facility. Subsequently, perform again the system integration step. This should result in a new design version: SpectrA v2. To test SpectrA v2 you can use the signal generator.

Hint:

- Make a new project named SpectrA_v2. Copy the files of the SpectrA_v1 in this project. Modify only the copied files and leave your SpectrA_v1 project intact.

Assistant signature:

Assignment 9: SpectrA v2 performance evaluation

Evaluate the performance of SpectrA v2 by measuring the time required by one SpectrA cycle and one FFT computation in the new scenario. When compared with SpectrA v1, how much improvement did you get? Explain the results to the student assistants.

Assistant signature:	
	SpectrA time: FFT time:

Session 6, Hardware compound ($2M^2A$)

Homework 5, Assignment 10

In this session you're going to implement a larger portion of the FFT algorithm in hardware to speedup your SpectrA. This hardware facility can efficiently compute the most time-consuming operations in the FFT.

The homework assignment requires you to design this computing facility and the required control unit. Make sure your work is checked by a student assistant before you start implementing.

When your design is correct, implement it in VHDL and the FPGA board. Test your implementation with a test program.

Homework 5: Hardware compound ($2M^2A$) design

Design a computing facility ($2M^2A$) that can perform the following computations: a*b-c*d and a*d+c*b. These are two compound operations that constitute the main loop of the FFT algorithm. Make a block diagram of the design and draw a FSM for the control unit. Also think about the communication between the 80C552 board and the FPGA board.

Hints:

- Speedup is important but do not forget that you should be able to implement your design under the current design constrains. Thus you should carefully trade FPGA area for performance.
- In order to speed up the design you may try to do things in parallel.

Assistant signature: ____

Assignment 10: Implementation of the compound multiplier ($2M^2A$)

Implement the compound multiplier in VHDL and implement in C a function that can communicate with your $2M^2A$ hardware facility. This function should have the following prototype.

```
complex compound(long, long, long, long)
```

Make a test program to test your compound multiplier.

Assistant signature: ____

Session 7, SpectrA v3 integration

Assignments 11 and 12 and optional

In this session you're going to integrate the compound into SpectrA. Again, make sure to backup your previous version first. If the integration is completed, measure the speed of the implementation. If you have time left, you can try to even further increase the speed of the implementation.

Assignment 11: SpectrA v3 integration

Rewrite the FFT module (again backup the project first!) in order to make use of the $2M^2A$ hardware facility. Subsequently, perform again the system integration step. This should result in a new design version: SpectrA v3. To test SpectrA v3 you can use the signal generator.

Assistant signature:	

Assignment 12: SpectrA v3 performance evaluation

Evaluate the performance of SpectrA v3 by measuring the time required by one FFT computation in the new scenario. When compared with Spectra v1 and Spectra v2 how much improvement did you get? Explain the results to the student assistant.

Assistant signature:	
	SpectrA time: FFT time:

Optional: Can you do it even faster?

Try to make the Spectra v3 even faster by moving more complex operations into hardware. Design and implement the new facility, and integrate it into the system. You can also think about optimizing something in software. Evaluate the performance. How much improvement did you get? Discuss the results with the student assistant.

Assistant signature:	
	SpectrA time: FFT time:

A Appendix A: The 80C225 board

80C552

Computer Board
80C552 Microcontroller
Monitor VMON552 Version 3.1
Development Software

Delft University of Technology
Faculty of Information Technology and Systems
Practical Group

Author: Leo van Velzen (original), Stephan Wong (editor)
October 2001

Table of Contents:

Introduction

The industrial importance of 'the microcontroller' has grown tremendously during the last decade. Typical features of this type of microprocessor, compared to the 'classical' microprocessors, like program memory, data memory and several hardware functions have been constructed together with the CPU in the same integrated circuit. Typical hardware functions are timers, serial interfaces, analog/digital converters, and pulse width modulation circuits. Program memory in ROM, (E)EPROM, or FLASH PROM is often added. As a result of this integration of data memory, program memory and interface functions a compact, stand alone and cheap computer system can be made, which is in particular suited to be used in (embedded) control and measurement systems.

The first industrial microcontroller was the Intel 8048. Apart from RAM and ROM this controller has three parallel input/output ports. It can be considered old-fashioned, as its functionality is very limited, according to modern standards. Its successor, the 8051, has become an de facto industrial standard that served as a basis to develop several microcontroller families equipped with the same 8051 CPU-kernel and hardware functions but with different extra's.

In this manual we will discus the 80C552 microcontroller, which is an extended version of the 80C51. The common 80C51 is the CMOS version of the 8051. The 80C552 and the 80C51 have the same CPU. The 80C552 that comes with the microcontroller board has no internal program ROM. However versions with build-in ROM exists.

In Chapter 1 the microcontroller system board will be discussed. If you want to build the computer or if you want to develop expansion hardware you need to read this chapter.

In Chapter 2 the 80C552 itself is discussed. Complete technical documentation can be found in DATA HANDBOOK IC28 (Philips). Many internet sites are dedicated to the 80C51. You may look at the next page to find a list of interesting URLs.

In Chapter 3 the monitor program of the microcontroller system is described. This monitor has many useful system routines that may be called by a user program.

Chapter 4 has the ASM51EP assembler documentation. This rather basic assembler produces a HEX-file directly so linking assembly and C modules is not possible. Use for instance the Keil tools instead if you want to do serious programming in C and/or assembly.

In appendix 1 an overview of the instruction set is given, with details about execution time, instruction length, and instruction operation.

An optional interface module is available with LCD and matrix keyboard support. You can obtain this module and a separate manual titled,

LCD Module & Matrix Keyboard, Interface Board, Monitor Routines

by contacting the author.

Websites

Many websites are dedicated to the 8051 microcontroller family. While gathering information using a search engine like Google, you will be overwhelmed by interesting but also often by useless pages. A few interesting URL's are listed below, however do not hesitate to explore the internet yourself.

Philips
http://www.semiconductors.philips.com/catalog
80C552 product catalog; data sheets; application notes

http://www.semiconductors.philips.com/mcu/products/selguides
Overview of microcontroller selection guides; datasheets; application notes

Atmel
http://www.atmel.com/atmel/products/prod20.htm
8051 datasheets; application notes

Vault Information Services - 8052.COM
http://www.8052.com
8051, 8052; frequently asked questions (FAQ); tutorials; chips; links; code library
This site has no data sheets

Intel Corporation
http://www.intel.com/design/mcs51
MSC51, MCS151, MCS251 controllers
This site has no data sheets

Embedded Systems Academy
http://www.esacademy.com/automation/faq.htm
8051 FAQ

ePanorama
http://www.epanorama.net
Selection of links

The EE Compendium
http://ee.cleversoul.com/8051.html
Mainly commercial site, books, hardware, software, projects, links

1 The microcontroller board

1.1 Detailed Schematic

The next pages show the schematics of the 80C552 microcontroller board, the components layout and the corresponding part list.
A brief description of the schematic follows below.

Power source
The power source consists of the 5 Volt voltage stabilizer U4, Diode D4, and capacitors C9, C10 and C11, a resistor R6, and a LED D5. This LED is on when power is present.

Reset circuit
SW1 is the RESET-switch. By pressing this switch the microcontroller is forced to restart. The reset circuit is made of R9, C13, and inverter U5C. Inverter U5D buffers the reset signal to the expansion bus.

Reset the processor by holding the RST-pin high during 2 machine cycles (24 oscillator cycles). During reset the outputs of the ALE and the PSEN pin are high. External circuitry may not change these signal levels.

The RST-pin can be activated by watchdog-timer T3. The T3 output pulse length is 3 machine cycles. This pulse is output through the RST-pin and can be used this way to reset external hardware. However, the current configuration of the reset circuit prevents the use of the reset output.

Power-on reset
After switching on the power source the processor is reset automatically. The initial voltage on capacitor C13 is zero. The capacitor is loaded through resistor R9. The processor is in the reset state as long as the capacitor voltage is lower than the trigger voltage of Schmidt-trigger U5C. The minimal required reset time is equal to the oscillator start-up time plus 2 machine cycle intervals (24 oscillator cycles). This will work if the power source voltage rise time is lower than 10 milliseconds. If this is the case, increasing the value of C13 could solve reset problems.

Expansion bus
The long row with 2 times 20 pins is the expansion bus J8 of the microcontroller system. The expansion bus carries all signals that could be needed to construct an expansion board. All address and data lines are present, however they have not been buffered. The power lines V_{DD} and V_{SS} (ground) are present to power small circuits from the microcontroller board. In addition the chip select signals CS2 to CS7 facilitate easy memory mapping. (CS0 and CS1 are being used to select on board EPROM and RAM). The external interrupt signals INT0 and INT1, the external timer T0 input, the reset signal, the read and write signals and the PSEN signal PSEN are all essential signals. The following figure shows the pin functions of connector J8.

+5V	1	2	+5v	A0 to A12	– de-multiplexed address lines
A0	3	4	A1	D0 to D7	- not buffered data bus
A2	5	6	A3	PSEN	- program select enable pulse
A4	7	8	A5	RD\	- read pulse output
A6	9	10	A7	WR\	- write pulse output
A8	11	12	A9	INT0	- external interrupt input
A10	13	14	A11	INT1	- external interrupt input
A12	15	16	GND	RST\	- reset output
D0	17	18	D1	T0	- timer T0 input
D2	19	20	D3	CS2\ to CS7\	– chip select signals
D4	21	22	D5		
D6	23	24	D7	\ means “active low”	
GND	25	26	PSEN		
WR\	27	28	RD\		
INT0	29	30	INT1		
RST\	31	32	T0		
CS2\	33	34	CS3\		
CS4\	35	36	CS5\		
CS6\	37	38	CS7\		
GND	39	40	GND		

Oscillator circuit

A for the most part internal oscillator circuit drives the microcontroller. The only external components are crystal X1 and capacitors C5 and C6.

The memory circuit

U8 is the 8 Kbyte EPROM in which the monitor program is residing. U7 is the 8 Kbyte external RAM component. U1 is the 'address latch' is used to de-multiplex the combined address-data bus of P0. The address decoder U6 generates the various ‘chip select’ signals. U5A and U5B function as an 'or-gate' for the signals PSEN and RD. That is why the external program memory and the external data memory in the system board share the same memory space. This hardware feature allows the execution of a program loaded in RAM.

Memory organization

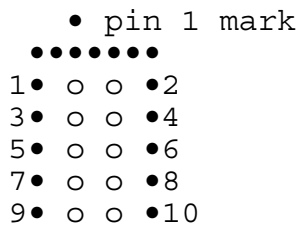
The 80C552 has 256 bytes of internal RAM memory. The lower 128 bytes (address range 0 to 7FH) are accessible through both direct addressing and indirect addressing. The upper 128 bytes (address range 80H to FFH) can only be accessed using indirect addressing. Direct addressing is used to access the "special function registers" (address range 80H to FFH).

The microcontroller is able to address in principle up to 64K bytes of external data memory and 64K bytes of external code memory. With special instructions the external memory can be addressed. Both memory areas have been divided into 8 blocks of 8K bytes. For each block a chip-select signal is generated (CS0 to CS7). On the board only CS0 and CS1 are being used to select the 8K byte EPROM and the 8K byte RAM respectively.

Chip Select	**Address area**	**Selects**
CS0	0000-1FFF	Monitor EPROM
CS1	2000-3FFF	Data/program RAM
CS2	4000-5FFF	External
CS3	6000-7FFF	External
CS4	8000-9FFF	External
CS5	A000-BFFF	External
CS6	C000-DFFF	External
CS7	E000-FFFF	External

P1, P4 and P5

Ports P1, P4 and P5 are accessible through a 10-pin header on the system board. An arrow marks pin number 1. (See the following figure)



Pin numbers of connectors P1, P4 and P5

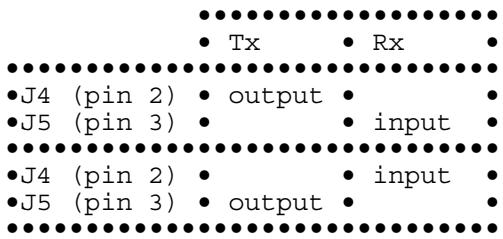
Pin	Function
1	GND (ground = 0 Volt)
2	V_{DD} (power voltage = +5 Volt)
3	bit 0
4	bit 1
5	bit 2
6	bit 3
7	bit 4
8	bit 5
9	bit 6
10	bit 7

The serial interface

U2, C2, C3, C4 and C12 make up the standard RS232 serial interface circuit. J6 is a 9-pin D-type connector. With jumpers J4 and J5 the serial input signal and output signal of connector J6 can be switched.

Tx and Rx jumpers (J4 and J5)

J4 and J5 selects the receive pins or transmit pins of the 9-pin D-connector. With jumper J4 the function of pin 2 is selected and with J5 the function of pin 3. The jumpers can be placed in position Tx or Rx. In position Tx the corresponding pin of the connector is the serial RS232-output of the computer. Position Rx indicates the serial RS232-input. Only two combinations are possible:



9-pin D-connector

The microcontroller board has been equipped with a 9-lead D-connector (female). A three-lead cable is used between this connector and the COM-port of a PC. There are two COM-port types: a 25-pin D-connector (male) or a 9-pin D-connector (male). A 25-pin connector has the serial output on pin 2 and pin 3 is the serial input. In a 9-pin connector input and output have been reversed. The 9-pin connector on the controller board has only 3 connections: pins 2, 3 and 5 (ground).

If the PC has a 9-pin serial connector you can use a straight cable (in which a pin number on one side is connected with the same pin number on the other side). J4 has to be in position Tx and J5 in position Rx. Only pins 2, 3 and 5 have to be wired.

If the PC has a 25-pin connector you can use a 25/9-pin adapter. Likewise J4 has to be in position Tx and J5 in position Rx.

If you have a crossed cable (pin 2 on one side connected to pin 3 on the other side), then J4 has to be in position Rx and J5 in position Tx.

The I^2C-interface

The resistors R1, R2, R3 and R4 make up the I^2C interface together with the zener diodes D1, D2 and D3. J10 is the interface connector. This 4 pin carries the lines SDA (data, pin 4) and SCL (clock, pin 3) and GND (ground, pin 1). Jumper J11 is used to feed pin 2 a voltage of +5 Volt. An arrow marks pin 1.

Reference voltage of the A/D converter

The voltage stabilizer U3 and the capacitors C7 and C8 form the reference voltage source for the A/D converter.

MISC

Connector MISC (4 pin connector J7) carries the pulse width modulation signals PWM0 (pin 1) and PWM1 (pin 2) and the start-ADC input signal (pin 3). Pin 4 is the low active RESET-input that can be used as RESET-output as well. Pin 1 is marked with an arrow.

Watchdog timer jumper

With jumper J9 the Watchdog Timer T3 can be switched on. Timer T3 is running if this jumper is in place. More information on the Watchdog timer can be found in the next chapter.

2 The 80C552 Microcontroller

2.1 Introduction

The 80C552 microcontroller has been designed to use in real-time environments, like instrumentation, industrial control and specific automatic control applications. The 80C552 has all standard 80C51 functions, including the instruction set. The controller has a number of special hardware functions to support the applications mentioned above. Extra Special Function Registers have been added to control the build-in special hardware functions.
The discussion below about the 80C552 is of course mostly relevant to the 80C51. If the description is exclusively relevant to this controller, the differences between both microcontrollers have been marked by [80C552].

2.2 Features of the 80C51 and the 80C552 Microcontroller

The following list shows the most important features of the 80C51 and 80C552. Most of them will be discussed in this manual. Additional information can be found in device specifications of manufacturers.

FUNCTION ↓ **TYPE →**	**8051**	**80C552**
8051 CPU	Yes	Yes
Internal RAM	128 bytes	256 bytes
External RAM	64K bytes	64K bytes
External ROM	64K bytes	64K bytes
8 bit input/output ports	4	5
8 bit input ports	-	1
Standard 16 bit timer/counter	2	2
16 bit timer/counter with capture/compare registers	-	1
Interrupts with two priority levels	5	15
On-chip Watchdog timer	-	1
Multiplexed analog inputs	-	8
Pulse width modulated outputs	-	2
I^2C-interface	-	1
Standard full duplex UART	1	1
On-chip oscillator and timing circuits	Yes	Yes

The 80C51 comes in the classical 40 pin DIP (Dual In-Line Package) witch has two rows of 20 pins and in the modern square shaped 44 pin PLCC (Plastic Leaded Chip Carrier) package.
The PLCC package of the 80C552 has four rows of 17 connection leads each (in this manual also called pins). This component has to be mounted in a special socket. The next table shows an overview of the 68 pin functions.

2.3 Pin functions of a [80C552]

Pen	Function	Description
2	V_{DD}	**Power.** +5V has to be present during normal operation, Idle mode and Power-down mode.
3	STADC	**Start ADC.** Input to start A/D conversion.
4	PWM0	**Pulse Width.** Modulation output 0.
5	PWM1	**Pulse Width.** Modulation output 1.
6	EW	**Enable Watchdog.** Active high. This input activates the watchdog timer T3 and switches off the power-down mode.
7-14	P4.0-P4.7	**Port 4.** 8 bits bi-directional I/O port.
15	RST	**Reset.** Input to reset the 80C552 microcontroller.
16-23	P1.0-P1.7	**Port 1.** 8 bits bi-directional I/O port.
24-31	P3.0-P3.7	**Port 3.** 8 bits bi-directional I/O port.
32,33		**Not connected.**
34	XTAL1	**Crystal 2.** Output of the inverting oscillator amplifier.
35	XTAL2	**Crystal 1.** Input of the inverting oscillator amplifier.
36,37	V_{SS}	**Two digital** ground pins.
38		**Not connected.**
39-46	P2.0-P2.7	**Port 2.** 8 bits bi-directional I/O port.
47	PSEN	**Program Storage ENable.** Output pin. Low active read pulse to access the external program memory.
48	ALE	**Address Latch Enable.** Active high output. Strobe pulse to write the LSB of an address during external memory addressing.
49	EA	**External Address.** Active high input. Enables instruction fetches from an external ROM, under the condition that the address is greater than 8192 (8K bytes).
50-57	P0.7-P0.0	**Port 0.** 8 bits bi-directional I/O port.
58	AV_{REF-}	**Low End.** ADC reference resistor.
59	AV_{REF+}	**High End.** ADC reference resistor.
60	AV_{SS}	**Analog** ground.
61	AV_{DD}	**Analog** power, +5V.
62-68,1	P5.7-P5.0	**Port 5.** 8 bits input port.

2.4 Special Function Registers

The address range of the Special Function Registers (SFR) lies between 80H and FFH. The addressing mode has to be direct only. However SFR Special Function Registers may not be used as memory locations; they are not. A few of these registers play a role during the execution of instructions by the CPU. These are registers A, B, DPTR, PSW and SP. The other SFR's are related to special on-chip hardware, like timers, I/O ports, A/D converter, serial interface and I^2C-interface. Not used address locations do not offer access to the hardware.

A short description of the Special Function Registers is given in this section. However, in sections to come the most important SFR's are dealt with more thoroughly. Again, complete information about these and not mentioned registers can be found in the manufacturers manual.

Accumulator (ACCU)
The accu (short for accumulator) is the most general processor register. Many instructions use or change the content of the accumulator. In instructions A, like in CLR A, indicates the accumulator.

Register B
The CPU uses this register during execution of the instructions MUL AB and DIV AB. Other instructions that use direct addressing may access register B as an extra memory location.

Program Status Word (PSW)
The PSW register holds a number of bits with the following function:

7	6	5	4	3	2	1	0
CY	AC	F0	RS1	RS0	OV	-	P

Bit	Mnemonic	Function:
0	P	Parity flag
1	--	User definable flag
2	OV	Overflow flag
3	RS0	Register bank Select control bit
4	RS1	Register bank Select control bit
5	F0	Flag 0
6	AC	Auxiliary carry
7	CY	Carry

RS1	RS0	Bank
0	0	0 -> (00H-07H)
0	1	1 -> (08H-0FH)
1	0	2 -> (10H-17H)
1	1	3 -> (18H-1FH)

Stack pointer (SP)
The SP register is eight bits wide. It points to an internal memory location called 'top of stack'. During execution of a PUSH/POP instruction its value is increased/decreased by 1. An LCALL or an ACALL instruction writes a return address to the stack and the SP is increased by 3. The RET and RETI instructions pull a return address from the stack and decrease the SP by 3. Upon a reset the SP value is 07H. An application program may change the stack pointer.

Data pointer (DPTR)
The DPTR register consists of two bytes: DPL and DPH. Together they form a 16-bit address register. DPTR can be used as a 16-bit register in instructions that read or write external memory, but DPTR can also be accessed as two separate 8 bit registers.

Port 0 t/m Port 5 (P0 t/m P5)
P0, P1, P2, P3, P4, and P5 are the microcontroller's 8 bit I/O-ports. Ports have a memory function. They can be written and read without changing direction.

Serial Data Buffer (S0BUF, S0CON)
The serial data buffer is constructed from two independent registers. A byte written to S0BUF will be transferred to the serial ports transmit latch. From there it will be transmitted serially.
A received byte is stored in the serial port's receive buffer. By reading S0BUF the receive buffer is accessed. Register S0CON is the control register of the serial interface.

Timer T0 and T1 registers (TL0, TH0, TL1, TH1, TCON, TMOD)
The registers TL0/TH0 and TL1/TH1 form the two 16 bit registers of Timer T0 and T1 respectively, each divided into two separately usable 8-bit registers. The registers TCON and TMOD have control over the timers.

Timer T2 registers (TML2, TMH2, STE, RTE, TM2CON)
Registers TML2 and TMH2 form together the 16 bit count register of timer T2. The registers STE, RTE and TM2CON have control over this timer.

Capture registers (CTL0 to CTL3, CTH0 to CTH3, CTCON)
This registers form in pairs the 16 bit registers needed by timer T2 to function in capture mode. Register CTCON is the control register of the capture function.

Compare registers (CML0 to CML3, CMH0 to CMH3)
This registers form in pairs the 16 bit registers needed by timer T2 in compare mode.

I^2C-bus registers (S1CON, S2STA, S1DAT, S1ADR)
These four registers are controlling the I^2C-bus interface. The microcontroller can be programmed to communicate with subsystems and IC's are equipped with such an interface.

Control registers (T2CON, S0CON, PCON)
The registers IP, IE, TMOD, TCON, T2CON, SCON, and PCON contain special information about respectively the interrupt system, thee timer/counters and the serial port.

Interrupt registers (IEN0, IEN1, IP0, IP1, TM2IR)
With Registers IEN0 and IEN1 various interrupt sources in the 80C552 can be switched on and off. IP0 and IP1 control the interrupt priorities. TM2IR is the interrupt register of timer T2.

The A/D converter registers (ADCON, ADCH)
Register ADCON is the A/D converter control register. In ADCH a part of the conversion result is to be stored.

Pulse width-modulation registers (PWM0, PWM1, PWMP)
These registers are used to generate pulse width modulated signals. PWMP is a prescaler.

Watchdog timer register (T3)
Timer T3 is a watchdog timer. T3 is also the name of the count register of this timer.

2.5 Addresses of Special Function Registers

The following list shows the addresses of the special function registers (SFR's). The * in the first column indicates an 80C51 register. The * in column 'Bit' marks a bit addressable register.

Address	Mnemonic	Bit	Function
80H *	P0	*	Port P0
81H *	SP		Stack pointer
82H *	DPL		Low byte (LSB) of DPTR
83H *	DPH		High byte (MSB) of DPTR
87H *	PCON		Power control
88H *	TCON	*	Timer T0 and T1 control
89H *	TMOD		Timer mode register
8AH *	TL0		Timer T0 low (LSB)
8BH *	TL1		Timer T1 low (LSB)
8CH *	TH0		Timer T0 High (MSB)
8DH *	TH1		Timer T1 High (MSB)
90H *	P1	*	Port P1
98H *	S0CON	*	Control register of the serial port
99H *	S0BUF		Data register of the serial port
A0H *	P2	*	Port 2
A8H *	IEN0	*	Interrupt enable register 0
A9H	CML0		Compare register 0 LSB
AAH	CML1		Compare register 1 LSB
ABH	CML2		Compare register 2 LSB
ACH	CTL0		Capture register 0 LSB
ADH	CTL1		Capture register 1 LSB
AEH	CTL2		Capture register 2 LSB
AFH	CTL3		Capture register 3 LSB
B0H *	P3	*	Port 3
B8H *	IP0	*	Interrupt priority register 0
C0H	P4	*	Port P4
C4H	P5		Port P5
C5H	ADCON		Control register ADC
C6H	ADCH		Low byte result of ADC conversion
C8H	TM2IR	*	Timer T2 interrupt register
C9H	CMH0		Compare register 0 MSB
CAH	CMH1		Compare register 1 MSB
CBH	CMH2		Compare register 2 MSB
CCH	CTH0		Capture register 0 MSB
CDH	CTH1		Capture register 1 MSB
CEH	CTH2		Capture register 2 MSB
CFH	CTH3		Capture register 3 MSB
D0H *	PSW	*	Program Status Word
D8H	S1CON	*	Control register I^2C-bus
D9H	S1STA		Status register I^2C-bus
DAH	S1DAT		Data register I^2C-bus
DBH	S1ADR		Address register I^2C-bus
E0H *	ACC	*	Accumulator register
E8H	IEN1	*	Interrupt enable 1
EAH	TM2CON		Control register van timer T2
EBH	CTCON		Capture control register
ECH	TML2		Timer T2 LSB
EDH	TMH2		Timer T2 MSB
EEH	STE		Set enable register (timer T2)
EFH	RTE		Reset Toggle register (timer T2)
F0H *	B	*	Register used by MUL and DIV
F8H	IP1	*	Interrupt priority register 1
FCH	PWM0		Pulse Width modulation register 0
FDH	PWM1		Pulse Width modulation register 1
FEH	PWMP		Pulse width modulation prescaler
FFH	T3		Register timer T3

2.6 Functions of I/O-ports

All Input/Output ports (I/O ports) of the 80C552 have alternate functions. The primary function is called 'Bit Input/Output'. The alternative functions have to be programmed in a separate Special Function Register (SFR). Alternative functions are mutual exclusive.

Ports are connections to the outside world. A program can access a port as an SFR. By reading a port the electrical voltage levels at the external port pins are being transferred to a register inside the microcontroller. A voltage level between 2,5 and 5 Volts is defined as a logical '1' and a level between 0 and 1 Volt is a logical '0'. Pin 0 of a port relates to bit 0 of the SFR of that port, etc. Ports can be used to transfer data to and from a connected peripheral device.

The internal output circuit of, for instance, port P1 has a switched transistor and a pull-up resistor per bit. In an 8051 is the resistor value of this pull-up resistor is 9 Kohm; in an 80C51 this value is 90 Kohm. A logical '0' at the output has a low output impedance. A '1' however behaves like a high impedance source (9 respectively 90 Kohm). This is important if a peripheral device has to be connected to this port. When the port is used as an input port the external device controls the signal levels at the pins. This is only possible if the output latch does not contain a - low resistance - '0'. To use a pin as an input it has to be made high impedance first by writing a '1' into the output latch.

The 80C51 has four 8-bit parallel ports (P0, P1, P2 and P3). The 80C552 has an additional two ports, P4 and P5.

Port 0 (P0)
P0 is an 8 bit digital I/O-port, used by the microcontroller as a multiplexed address/data bus to access external ROM or RAM. In this configuration P0 cannot be used to perform programmed I/O-operation.

Port 1 (P1)
Also P1 is an 8 bit digital I/O-port. The alternative functions of the 8 pins exist in the 80C552 only. Look at the following table.

P1.0 - P1.3	External interrupt inputs of timer T2 (capture interrupts)
P1.4	External clock input for timer T2
P1.5	Reset signal for timer T2
P1.6	Clock-line of the I^2C-bus
P1.7	I^2C data-line

Ports can be used to perform many tasks. Anything with an 'on' and an 'off' state can be controlled. An important drawback is the output power/current limitation of CMOS outputs. In many cases an additional amplifier is needed. The following table shows the maximum current in an output pin of port P1 in two cases: short circuit of the output pin to V_{DD} (+5 Volt) and to V_{SS} (GND) driving the output high '1' or low '0'.

Bit	**P1->V_{DD}**	**P1->V_{SS}**
	Sink current	Source current
0	-22mA	0µA
1	-0,8µA	12µA

It is important to mention here that, under normal conditions, according to the specifications, the maximum 'sink current' is 10mA. This limits the output power capability even more.

Port 2 (P2)
P2 is an 8 bit digital I/O port. The alternative function is the output of the most significant address byte to the external memory. The processor uses P2 as an address bus buffer.
Apart from this addressing method the processor can use an 8 bit addressing method within a 256-byte memory page (instructions MOVX A,@Ri and MOVX @Ri,A). In this case P2 is used as a page-register. The number in P2 selects a memory page of 256 bytes. This addressing method is possible in external data RAM only.

Port 3 (P3)
P3 is, like ports P0, P1, and P2 an 8 bit digital I/O port. All pins have alternative functions, which are the same in both the 80C51 and the 80C552.

P3.0	Serial input line
P3.1	Serial output line
P3.2	External interrupt INT0
P3.3	External interrupt INT1
P3.4	Timer T0, external input
P3.5	Timer T1, external input
P3.6	Write pulse for external memory
P3.7	Read pulse for external memory

In a computer system with external memory and a serial interface for communication with a PC (like our 80C552 system) only pins P3.2, P3.3, P3.4, and P3.5 can be used in applications. With the exception of P3.5 these pins can be found in the 40-pin expansion bus of the 80C552 board.

Port 4 (P4) [80C552]
The 80C552 controller has an extra 8 bit digital I/O port. In combination with a number of signals of timer T2 and the compare registers it is possible to realize complex functions with P4. Depending of the content of the registers RTE and STE signals at the 8 pins of P4 are set or cleared when the content of the count register of timer T2 equals the content of the compare registers. You will find more about this in the section on Compare Registers.

P4.0	CMSR0	These six outputs can be set or cleared
P4.1	CMSR1	
P4.2	CMSR2	
P4.3	CMSR3	
P4.4	CMSR4	
P4.5	CMSR5	
P4.6	CMT0	These two outputs can be inverted
P4.7	CMT1	

Port 5 (P5) [80C552]
The 80C552 features a special digital input port: P5. The alternative function concerns the input of up to 8 analog signals internally connected to the build-in 10-bit Analog/Digital converter through an analog multiplexer.

2.7 Timers/Counters

The 80C51 has two timer/counters, T0 and T1. The functions of these timers are identical apart from a few details. The 80C552 has an additional timer/counter, T2 and a fourth timer named T3. These timer/counters have their specific functions for special purposes. All timer/counters are up-counters: the content of the count register is incremented by 1 at each active edge of the timers input signal.

To generate a time interval or to count a pre-defined number of pulses the timer is used to "count down" a value to zero. To get this effect the count register is to be loaded with 2^n - NUMBER, in which n is the number of bits in the count register and NUMBER is the value to be counted down. After loading the count register the timer has to be started. The timer overflow indicates the end of the desired time interval. This event is signaled in an overflow bit. It will generate an interrupt too if enabled.
A running timer does not stop after overflowing, it continues counting from 0. The count register can be (re)programmed only when the timer is not running.

2.7.1 Timers T0 and T1

There are two Special Function Registers connected to the timer functions: TMOD and TCON. Register TMOD (timer mode) configures both timers and register TCON (timer control) interacts with a running program.

TMOD (Timer/Counter Mode Control Register)

7	6	5	4	3	2	1	0
GATE	C/T	M1	M0	GATE	C/T	M1	M0

The four low significant bits of TMOD configure timer T0, the highest four bits have the same functionality but they configure timer T1. Both timers have four modes, mode 0 t/m 3. The mode has to written into bits M0 and M1 in register TMOD.

Bit C/T in register TMOD is used to select the counter-function (with C/T=1) or the timer-function (C/T=0).
The counter-function counts falling edges (high-low transitions) of the external timer-input signal at pin P3.4 (T0) or pin P3.5 (T1). The maximum frequency of that input signal is Fosc/24 (which is two times the machine cycle interval time). The duty-cycle has no restrictions, however the signal has to be high during at least one machine cycle and low during at least one machine cycle.
The timer function counts each 12th pulse of the oscillator signal. Thus, the interval time equals one machine cycle interval. In this situation pin T0 respectively T1 are free to use as a bit input or output pins.

The function GATE enables the external control of timers T0 and T1. With bit GATE set to '1' the timer is only counting if pin INT0 (P3.2) respectively INT1 (P3.3) is high and if the timer is in run mode (see register TCON). A timer will stop counting when a low signal is put at the corresponding control input pin. So INT0 and INT1 can be considered to be external start/stop inputs of the timers T0 and T1.
With GATE set to '0' the timer run mode is controlled by software only. In this situation pin INT0 respectively INT1 are free to use as bit input or output pins.

The various modes of timer T0 and T1 are discussed below.

Mode 0: M1=0, M0=0
Timer T0 and T1 behave independently as 8 bit timers with a fixed 5 bit prescaler in TL0 respectively TL1. So the frequency of the input signal (which is the oscillator signal) is divided by 32 (2^5). The resulting signal is fed to the 8 bit timer TH0 respectively TH1. Therefore the effective counting frequency with C/T=0 is Fosc/(12*32).
When the timer overflows (the count value changes from FFH to 00H) the timer overflow bit (TF0 respectively TF1 in register TCON) is set to '1'.

Mode 1: M1=0, M0=1
Timer T0 and T1 behave independently as a 16 bits timer. T0 consists of TL0 and TH0 whereas T1 is a combination of TL1 and TH1. The resulting counting frequency with C/T=0 is Fosc/12.
When the timer overflows (the count value changes from FFFFH to 0000H) the timer overflow bit (TF0 respectively TF1 in register TCON) is set to '1'.

Mode 2: M1=1, M0=0
Timer T0 and T1 behave independently as 8 bit timers in TL0 respectively TL1 without a prescaler. TH0 respectively TH1 contains a 'reload value'. Counting is done in TL0 respectively TL1. When the timer overflows (the count value changes from FFH to 00H) the timer overflow bit is set to '1' and the reload value is copied automatically to the count register. Counting is continued from this value. The effective count frequency with C/T=0 is Fosc/12.

Mode 3: M1=1, M0=1
Timer T0 consists of two different 8-bit counters, TL0 and TH0.
TL0 is configured by bit GATE and C/T of timer T0 in register TMOD. The external input in this case is T0 at pin P3.4. TL0 is controlled by TR0 and TF0 in register TCON.
TH0 can only run on 1/12th of the oscillator frequency. TH0 is started with the timer run bit TR1 of timer T1. Overflow of TH0 is registered in TF1 of timer T1.
Timer T1 is halted in mode 3. The count registers of timer T1 keep their content. Switching from mode 3 to one of the other modes can start the timer. Because the overflow bit TF1 is not available timer T1 cannot generate interrupts. With timer T0 in mode 3, timer T1 can only be used as a baud-rate generator.

TCON (Timer/Counter Control Register)

7	6	5	4	3	2	1	0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Timer T0	Timer T1	Function
bit 0	bit 2	Interrupt type control bits IT0 and IT1. These bits have to be set and reset by software. With IT0 respectively IT1 set to '1' the external interrupt input INT0 respectively INT1 is sensitive to a falling edge. With a '0' in one of these bits the interrupt is active during a low level at the corresponding interrupt input.
bit 1	bit 3	Interrupt status bits IE0 and IE1. These bits are set by the processor upon detection of an interrupt signal at the corresponding external interrupt input INT0 respectively INT1. When the interrupt input is edge sensitive the processor resets the status bit automatically during the interrupt cycle. When the interrupt is level triggered the status bit has to be reset by software.

bit 4	bit 6	Timer run bits TR0 for timer T0 and TR1 for timer T1. These bits have to be set and reset by software. With the run bit set to '1' a timer/counter is running (counting). While running the value in the count register is incremented by 1 at each input clock pulse. With the run bit set to '0' a timer/counter is halted.
bit 5	bit 7	Timer overflow bits TF0 for timer T0 and TF1 for timer T1. These bits are set to '1' by hardware when a timer overflows. They are reset automatically during the interrupt cycle or by software in case interrupts have been disabled. For more details, see the section about interrupts.

The value in a count register can only be changed by software if the timer is halted (TR0=0 respectively TR1=0).

After an overflow a timer continues counting.

Important: Timer T1 is often used as the baud-rate generator for the serial port. In that case one has to be careful with programming registers TMOD and TCON.

2.7.2 Timer T2 [80C552]

Timer T2 is a 16-bit timer/counter with 'compare' and 'capture' functions. A programmable prescaler that divides by 1, 2, 4, or 8 clocks the 16-bit timer/counter. The prescaler's input is clocked by 1/12th of the oscillator frequency, of with a rising edge at the clock input of timer T2 (P1.4).
The prescaler is reset when its division factor or its clock input source is changed or when the timer/counter's mode is changed. The count register is read directly, as there is no buffer.

Although timer T2 cannot be loaded it can be reset by the RST signal or by a rising edge at input pin RT2 (P1.5), provided it is active. In the Idle mode the timer/counter is halted and reset.

TM2CON (Timer T2 Control Register)

7	6	5	4	3	2	1	0
T2IS1	T2IS0	T2ER	T2B0	T2P1	T2P0	T2MS1	T2MS0

Bit	Abbreviation	Function
TM2CON.0	T2MS0	Timer T2 Mode select bit 0.
TM2CON.1	T2MS1	Timer T2 Mode select bit 1.
TM2CON.2	T2P0	Timer T2 prescaler bit 0.
TM2CON.3	T2P1	Timer T2 prescaler bit 1.
TM2CON.4	T2B0	Timer T2 byte overflow interrupt flag.
TM2CON.5	T2ER	Enable timer T2 reset signal at pin RT2.
TM2CON.6	T2IS0	Enable byte overflow interrupt.
TM2CON.7	T2IS1	Enable 16 bit overflow interrupt.

Timer T2 Prescaler

T2P1	T2P0	T2 clock input
0	0	source
0	1	source /2
1	0	source /4
1	1	source /8

Timer T2 Mode Selector

T2MS1	T2MS0	Selected mode
0	0	Timer T2 halted
0	1	Timer T2 clock = $F_{osc}/12$
1	0	Test mode; Do not use!
1	1	T2 clock = clock at external pin T2.

Compare Registers
Three compare registers (CM0, CM1, CM2) belong to timer T2. Each of them occupies two bytes in the SFR map. The high and the low byte of the compare registers are compared with the high and the low byte of timer T2. When both 16 bits values are equal an interrupt flag is set and if enabled, an interrupt is generated.

The registers STE and RTE are used to set or clear or invert certain bits of port P4.

- Bit 0 to 5 of port P4 are set when the values of timer T2 and compare register CM0 become equal under the condition that the corresponding bits 0 to 5 in register STE are set.
- Bit 0 to 5 of port P4 are reset when the values of timer T2 and compare register CM1 become equal under the condition that the corresponding bits 0 to 5 in register RTE are set.
- Bit 6 and 7 of port P4 are inverted when the values of timer T2 and compare register CM2 become equal under the condition that the corresponding bits 6 and 7 in register RTE are set. Bit 6 and 7 in register STE have no function.

Capture Registers
The function of the capture registers is the opposite of the compare registers. There are four capture registers, CT0, CT1, CT2, and CT3 associated to timer T2. An external capture signal at one of the capture input pins (P1.0-P1.3) copies the current value of timer T2 to the corresponding capture register. If necessary this event can be transferred to the processor as an interrupt. Per capture register there are two bits in the Capture Control Register (CTCON). A '0' disables a capture signal, a '1' enables it.

CTCON (Capture Control Register)

7	6	5	4	3	2	1	0
CTN3	CTP3	CTN2	CTP2	CTN1	CTP1	CTN0	CTP0

Bit	Abbreviation	Function
CTCON.0	CTP0	Capture 0, capture on positive edge.
CTCON.1	CTN0	Capture 0, capture on negative edge.
CTCON.2	CTP1	Capture 1, capture on positive edge.
CTCON.3	CTN1	Capture 1, capture on negative edge.
CTCON.4	CTP2	Capture 2, capture on positive edge.
CTCON.5	CTN2	Capture 2, capture on negative edge.
CTCON.6	CTP3	Capture 3, capture on positive edge.
CTCON.7	CTN3	Capture 3, capture on negative edge.

It is possible to capture on both a negative and a positive edge.

To conclude the discussion of timer T2 the interrupt flags in register TM2IR has to be mentioned. All flags indicate an active interrupt status with '1'. A processor interrupt will occur if the corresponding interrupt enable flag in register IEN1 is set and interrupts have been enabled by setting bit EA in register IEN0. In register IEN1 all timer T2 interrupt flags are present with the exception of the byte overflow interrupt flag, which is located in bit 4 of the timer T2 Control Register (TM2CON).

TM2IR (Timer T2 Interrupt Flag Register)

7	6	5	4	3	2	1	0
TM2OV	CMI2	CMI1	CMI0	CTI3	CTI2	CTI1	CTI0

Bit	Abbreviation	Function
0	CTI0	Capture 0 interrupt flag.
1	CTI1	Capture 1 interrupt flag.
2	CTI2	Capture 2 interrupt flag.
3	CTI3	Capture 3 interrupt flag.
4	CMI0	Compare register 0 Interrupt flag.
5	CMI1	Compare register 1 Interrupt flag.
6	CMI2	Compare register 2 interrupt flag.
7	TM2OV	Timer T2 16 bit overflow flag.

2.7.3 Timer T3 [80C552]

Timer T3 is not a common timer like timer T0, T1, or T2. Timer T3 is a 'watchdog timer' that can only be switched on and off by hardware with pin EW. Timer T3 is running when pin EW is connected to GND. By connecting pin EW to the 5 Volt power source T3 is switched off.

The timer value is incremented by 1 at every 12th oscillator period. The timer itself is build of an 11 bit prescaler and an 8-bit count register. The effective counting frequency F_{Timer} is:

$$F_{Timer} = F_{Osc} / (12 \times 2048)$$

The interval time (counting time until overflow) depends on the content of the count register. The maximum interval time is obtained with a starting value of 00H. With a starting value of FFH the minimum time is obtained. These interval times are respectively 419 millisecond and 1.6 millisecond, assumed that the oscillator frequency is 15 MHz.

To prevent the timer from overflowing a running program has to reload the timer regularly. If this is not done in time as a result of a hardware or a software error, the timer will overflow causing the processor to be reset after which the processor restarts at address 0. The reset time is 3 machine cycles long.

Before loading a new value in the watchdog timer, the WLE bit in register PCON has to be set to '1' first (Watchdog Load Enable). After loading the count register this bit is cleared automatically.

2.8 Idle Mode

In the Idle mode interrupts, serial ports and timers keep functioning while the CPU is halted.

In the 80C552 the following actions are halted in the Idle Mode as well:
- Timer T2 (stop + reset)
- PWM0, PWM1 (reset + output=high)
- ADC (stop when active)

To go into Idle Mode the IDL-bit (PCON.0) has to be set. While in Idle Mode the CPU is halted. The CPU-status (all relevant registers) keeps in tact. The ALE and PSEN signal are kept high in Idle Mode.

There are two ways to recover from Idle Mode:

- An interrupt cancels Idle Mode. The interrupt routine is handled and after return from the interrupt service routine the program is resumed with the instruction following the instruction that activated the Idle Mode. With such an instruction thee flags GF1 and GF0 (General Purpose Flag Bits) in register PCON can be set to '1'. The interrupt routine should check and reset these flags. Using these flags it is possible to find out that the interrupt has occurred during Idle Mode or not.
- The Idle Mode is cancelled by an external reset or by an internal reset caused by timer T3. As the oscillator is still active, the hardware reset signal has to be active during 2 machine cycles (24 oscillator periods).

PCON (Power Control Register)

7	6	5	4	3	2	1	0
SMOD	-	-	WLE	GF1	GF0	PD	IDL

2.9 A/D Converter [80C552]

The Analog/Digital converter (ADC) has 8 input signals (port P5). The resolution is 10 bit, which means that 2^{10} different signal levels can be converted. The generated code is unsigned binary. The conversion result can be calculated using this formula:

$$CODE = 1024 \times (V_{in} - AV_{ref-}) / (AV_{ref+} - AV_{ref-})$$

In this formula V_{in} denotes the measured signal level. CODE is the conversion result, AV_{ref+} is the positive reference voltage and AV_{ref-} is the negative reference voltage. In most situations pin AV_{ref-} is tied to GND and AVref+ is connected to +5 Volt in which case the formula can be simplified:

$$CODE = 1024 \times V_{in} / 5 \quad \text{or} \quad V_{in} = 5 * Code / 1024 \text{ Volt}$$

The conversion time is 50 machine cycles, which results in a conversion time of 40 microseconds in case the oscillator frequency is 15 MHz.

The microcontroller has two ADC-registers. In register ADCH bit 9 to 2 of the last measured result is stored. Register ADCON contains the 2 least significant bits of the conversion result together with 6 status and control bits. Writing a ‘1’ to bit ADCS starts the conversion. When bit ADEX is '1' the ADC can be started too with a rising edge at pin STADC. There is a timing condition concerned. Preceding the starting edge the signal has to be low during at least one machine cycle and has to be kept high at least one machine cycle following the starting edge. During the conversion the ADC is insensitive to writing a '1' in ADCS or a rising edge at STADC.

ADCON (AD Control Register)

7	6	5	4	3	2	1	0
ADC.1	ADC.0	ADEX	ADCI	ADCS	AADR2	AADR1	AADR0

Bit	Abbreviation	Function	
0		AADR0	LSB of a 3-bit channel selection code.
1		AADR1	Middle bit of a 3-bit channel selection code.
2		AADR2	MSB of 3 bits channel selection code.
3		ADCS	Conversion starts with a '1'. This bit is automatically reset at the end of the conversion.
4		ADCI	Signals end of conversion (set to '1'). If enabled this bit generates an interrupt. Has to be reset by software.
5	ADEX		A '1' enables the external start input.
6	ADC.0		Bit 0 of the conversion result.
7	ADC.1		Bit 1 of the conversion result.

2.10 Interrupts

Events that happen externally (including events in the on-chip circuits) often need the CPU to response immediately. The running program has to be interrupted and the CPU continues to execute a service routine that will handle this request of an interrupting device. Drawing attention from the CPU by a hardware signal is called an "interrupt". The 80C552 has a not to complex interrupt system characterized by 'multiple source', 'two priority-level' and 'nested interrupts'.

The CPU has to remember where the running program was interrupted. To do that, the current Program Counter (which points to the next instruction to execute) is saved on the stack after which the PC is loaded the starting address (called vector address) of an interrupt routine that corresponds to the interrupt source. The 80C51 family has fixed vector addresses in the lower program memory. In many systems an EPROM holding the monitor program occupies this part of the memory map. In such cases there has to be jump instruction on the vector address that connects to the interrupt service routine in the program memory. In a system with a monitor program in ROM and a program memory in RAM a jump table in RAM is used to organize this.

The 80C552 has 15 interrupt sources of which the first 5 are standard family features (printed bold in the table below). The column 'Jump Address' shows the destination addresses. Jump instructions to interrupt routines has to be placed at these locations. The instruction RETI pulls the return address from the stack and resets the priority level flip-flop.

Nr	Name Interrupt source	Prio	Vector address	Jump address
0	**INT0 external interrupt 0**	**1**	**0003H**	2000H
1	**TF0 timer 0 overflow**	**4**	**000BH**	2003H
2	**INT1 external interrupt 1**	**7**	**0013H**	2006H
3	**TF1 timer 1 overflow**	**10**	**001BH**	2009H
4	**RI/TI UART (serial interface)**	**13**	**0023H**	200CH
5	I2C-PORT	2	002BH	200FH
6	T2 CAPTURE 0	5	0033H	2012H
7	T2 CAPTURE 1	8	003BH	2015H
8	T2 CAPTURE 2	11	0043H	2018H
9	T2 CAPTURE 3	14	004BH	201BH
10	ADC	3	0053H	201EH
11	T2 COMPARE 0	6	005BH	2021H
12	T2 COMPARE 1	9	0063H	2024H
13	T2 COMPARE 2	12	006BH	2027H
14	TF2/EXF2	15	0073H	202AH

Table: The interrupt sources of the 80C552. The 8051 has only the first 5 interrupt sources.

Each interrupt source can be made active or passive individually by writing a '1' or a '0' to the corresponding bit in the IEN0 and IEN1 registers. The interrupt latency is about 3 to 8 machine cycles.

IEN0 (Interrupt Enable Register 0)

7	6	5	4	3	2	1	0
EA	EAD	ES1	ES0	ET1	EX1	ET0	EX0

A '1' enables a specific interrupt.

Bit	Abbreviation	Function
0	EX0	Enable external interrupt 0.
1	ET0	Enable timer T0 overflow interrupt.
2	EX1	Enable external interrupt 1.
3	ET1	Enable timer T1 overflow interrupt.
4	ES0	Enable UART interrupt.
5	ES1	Enable I^2C-bus interrupt.
6	EAD	Enable ADC interrupt.
7	EA	Main interrupt enable/disable.
		A '0' in this bit disables all interrupts.

IEN1 (Interrupt Enable Register 1)

7	6	5	4	3	2	1	0
ET2	ECM2E	ECM1	ECM0	ECT3	ECT2	ECT1	ECT0

A '1' enables a specific interrupt.

Bit	Abbreviation	Function
0	ECT0	Enable interrupt capture register 0.
1	ECT1	Enable interrupt capture register 1.
2	ECT2	Enable interrupt capture register 2.
3	ECT3	Enable interrupt capture register 3.
4	ECM0	Enable interrupt comparator 0.
5	ECM1	Enable interrupt comparator 1.
6	ECM2	Enable interrupt comparator 2.
7	ET2	Enable both overflow interrupts of timer T2.

2.11 Interrupt Priority Structure

There are two interrupt priority levels, high and low. The priority of each interrupt source can set to one of these levels. The interrupt priorities are set in the registers IP0 and IP1. A low level interrupt can be interrupted by a high level interrupt. If a low level interrupt and a high level interrupt occur simultaneously the high level interrupt has precedence.

When two (or more) interrupts with the same priority levels occur simultaneously a second priority scheme is used. The interrupt with the highest level is handled first. The highest priority has value 1 in the table below; the lowest priority has value 15. These priority levels do not correspond with the numbered order of interrupts (see table above). The priority level of the five 80C51 interrupts is stored in register IP0.

IP0 (Interrupt Priority Register 0)

7	6	5	4	3	2	1	0
-	PAD	PS1	PS0	PT1	PX1	PT0	PX0

A '1' sets the interrupt priority to high, a '0' to low.

Bit	Abbreviation	Priority	Function
0	PX0 [80C51]	1	External interrupt INT0.
1	PT0 [80C51]	4	Timer T0.
2	PX1 [80C51]	7	External interrupt INT1.
3	PT1 [80C51]	10	Timer T1.
4	PS0 [80C51]	13	Serial port.
5	PS1	2	I^2C-bus.
6	PAD	3	ADC.
7	---		Not used.

IP1 (Interrupt Priority Register 1)

7	6	5	4	3	2	1	0
PT2	PCM2	PCM1	PCM0	PCT3	PCT2	PCT1	PCT0

A '1' sets the interrupt priority to high and a '0' to low.

Bit		Abbreviation	Priority	Function
0	PCT0	5	Capture register 0.	
1	PCT1	8	Capture register 1.	
2	PCT2	11	Capture register 2.	
3	PCT3	14	Capture register 3.	
4	PCM0	6	Comparator 0.	
5	PCM1	9	Comparator 1.	
6	PCM2	12	Comparator 2.	
7	PT2	15	Both overflows of timer T2.	

2.12 Registers

There are 8 registers named R0 to R7 in the internal RAM. The register names appear in the instruction set. The CPU can address them by name. The registers are organized in 4 different memory areas. With 2 bits in the PSW (program status word) a program can select the desired bank of registers. The table below shows the memory locations of R0 to R7 in the various banks.

Bank 0	00H - 07H (the address of R0 is 00H)
Bank 1	08H - 0FH (the address of R0 is 08H)
Bank 2	10H - 17H (the address of R0 is 10H)
Bank 3	18H - 1FH (the address of R0 is 18H)

After a hardware reset the stack pointer is set to 07H. The first byte written to the stack is placed at address 08H. So the content of R0 in register bank 1 is changed. If necessary the stack pointer has to be loaded with a suitable base address before the stack can be used.

2.13 Bit Addressable Registers

Next to the fourth register bank, starting from address 20H, there are 16 bit addressable bytes. All bits within this memory area can be addressed without changing the other bits in the same byte. With for instance the instruction SETB or CLR a bit can be set or reset. Bit instructions use bit addresses not byte addresses. Bit 0 within the memory location 20H has bit address 0; bit 7 has bit address 7,and so on. Bit 7 of memory location 2FH has the highest bit address, 127. A bit address can be calculated in the following way:
bit address = (byte address - 20H) x 8 + bit number.

Apart from these bit addresses there are 16 bit addressable bytes in the Special Function Registers area. All SFR addresses with bit 0, 1, and 2 set to 0 are bit addressable, so every 8th SFR is bit addressable. The first bit addressable byte in the SFR area has address 80H. The next byte is 88H and so on. It is easy to calculate a bit address: simply add the byte address and the bit number within that byte. Example: bit address 9AH equals to byte address 98H (logical and with 11111000B) plus bit 2.

2.14 Introduction to the Instruction Set

An instruction has a 1, 2, or 3-byte length. The first byte is the operation code (opcode). The operand can be coded in the opcode, but in most cases one or two extra bytes are added to specify the operand. The way an operand is specified is called addressing mode. Not all addressing modes can be used in most instructions or each memory types. There are five different addressing modes:

Register (Rn):
The operand is present in one of the registers R0 to R7.

```
Example:    MOV   A,R0
```

Direct (dir):
The operand is present in the internal memory location at the address specified in the instruction. Direct addressing can be used in the internal RAM area from 0 to 7FH and in the Special Function Register area.

```
Example:    MOV   A,60H       ;address is RAM location
            MOV   A,90H       ;address is SFR
```

Register indirect (Ri of DPTR):
Register Ri holds the address of the operand. In the internal RAM area indirect addressing is possible only by using register pointers R0 and R1. All 256 bytes can be addressed using the register indirect addressing mode.

Register DPTR has to be used to address an operand in the external data memory. There is an alternative way to address an external RAM location: select a 256 byte memory page by writing its page number to P2 and use R0 or R1 to address a memory location within the selected page. The only instruction that can address external RAM using one of these two methods is MOVX.

```
Example:    MOV   A,@R0             ;R0 points to an internal RAM location
            MOVX A,@DPTR    ;DPTR points to an external RAM location
            MOV   P2,#20H
            MOVX @R1,A              ;R1 points to a location in page 20H in external RAM
```

Immediate (#):
The operand is a part of the instruction.

Example: MOV A,#55H ;55H is the operand

Base address plus index register indirect:
During execution the content of the index register (A) is added to the basis register (DPTR or PC). The resulting address points to the location of the operand. This addressing mode makes it possible to read constant values from the internal or external code memory (dependent on the EA pin).

Example: MOVC A,@A+DPTR
MOVC A,@A+PC

Register specific:
This is not a genuine addressing mode. The operand is restricted one particular register, for instance A, B, or DPTR.

Example: MUL AB

Relative addressing:
Relative jump instructions have a relative destination operand (rel) that is added to the Program Counter at execution time which points to the subsequent instruction. The resulting jump address is at a distance of maximum +127 or -128 bytes from the current PC. So 'rel' is an 8 bit 2-complement number.

Example: JC LABEL

The destination address is specified as an absolute address. The assembler calculates the relative jump operand value.

2.15 The Instruction Set

See appendix 1 for a fully detailed list of instructions.

3 Monitor Program

The EPROM holds a monitor program called VMON552. Version 3.0 is discussed in this section. Communication between the monitor and a terminal program takes place through a serial connection between a COM-port of a PC and the serial interface of the microcontroller board.

3.1 Switch On / Reset Procedure

Both after switching on the power and after resetting the microcontroller board with the reset button, the Enter-key of the PC keyboard (or the Carriage Return-key of terminal) has to be pressed. In response the monitor initializes a number of settings and shows the following message on the screen:

```
VMON552 - version 3.1
  Auto baud rate detection
  Stack pointer: 40H
  External monitor RAM area: 2000H - 207FH
  Internal monitor RAM area:   30H -   3FH
```

If the LCD unit is connected the following message will be shown:

```
* VMON552 V3.1 *
* LCD driver   *
```

The initialization procedure is performed as follows:

Cold Start (reset or service routine 0)

a SP = 40H, PSW = 00H

b Link default interrupt routines though a secondary jump table.

c Selection of standard conversion table for matrix keyboard.

d Initialize LCD-module.

e Search for an application program:
- If found the program is started.
- After finishing a program or if there is no program continue at f.

f Auto baud-rate detection.

g Initialization of serial interface with format 8N1.
- TMOD = 2XH (X = unchanged)
- TH1 = TH2 = measured baud-rate parameter.
- SMOD = 1 (PCON.7)
- SCON = 52H, serial mode 1, receiver and transmitter on.
- TR1 = 1 (TCON.6), start Timer 1.

h Send Start-message to terminal.

i If LCD-module present then:
- Send message to terminal,
- Show start-message on LCD.
Continue at k.

Warm Start (service routine 1)

j SP = 40H, PSW = 00H, register bank 0 selected.
Continue at k.

Interactive command processor

k - Show prompt ('$'),
- Read a command,
- Execute command,
- Repeat from k.

3.2 Monitor Commands

In response to most of the commands the monitor program asks to enter a number in hexadecimal format. The 'prompt' to do this is the '>'-sign. Numbers are to be entered without the 'H', '$' of 0x number base designators. Numbers have to be delimited by <Enter>. When a byte value has to be entered no more then two hexadecimal keys can be typed. When two-byte value is to be entered the number of characters is limited to four. Typing errors can be corrected by typing the Backspace-key. Type the <Esc>-key to abort or to finish a command.
In the various examples in this section both entered characters and monitor output is shown intermingled. Non-visible characters are shown between < >, like <Enter>.

P: Program memory display

In response to this command the content of a section of the external program memory (ROM) is displayed. In the 80C552 system this function can display the content of external data memory (RAM) as the PSEN pulse (ROM) is combined with the RD pulse (RAM).
After typing 'P' the '>' prompt is displayed and a hexadecimal address value has to be typed. A part of the memory content is displayed starting from the entered address. By typing P<enter> the consecutive memory area will be displayed.

Example:
Dump from address 500H:

```
$P>500<Enter>
0500: 00 01 02 03 04 05 06 FF FF FF FF FF FF FF FF FF
0510: 01 02 03 04 05 06 07 FF FF FF FF FF FF FF FF FF
0520: 02 03 04 05 06 07 08 FF FF FF FF FF FF FF FF FF
0530: FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00 00
$
```

D: external Data memory display

In response to this command the content of a section of the external data memory (RAM) is displayed. In the 80C552 microcontroller board program memory and data memory resides in the same physical memory (see command P). Apart from a dump in hexadecimal format the memory content is displayed in ASCII as well. Only alphanumeric characters and are displayed. The remaining ASCII values are shown as a 'period' sign. By typing D<enter> the consecutive memory area will be displayed.

Example:

Dump from address 3000H:

```
$D>500<Enter>
3000: 00 01 02 03 04 05 06 FF FF FF FF FF FF FF FF FF  ................
3010: 41 42 43 44 45 46 47 20 20 00 00 00 00 00 00 00  ABCDEFG  .......
3020: 02 03 04 05 06 07 08 FF FF FF FF FF FF FF FF FF  ................
3030: FF FF FF FF FF FF FF 31 32 33 34 35 36 37 38 39  .......123456789
$
```

I: Internal data memory display

In response to this command the content of a section of the internal data memory (RAM) is displayed. After typing 'I' the '>' prompt is displayed and a hexadecimal address value has to be typed. A part of the memory content is displayed starting from the entered address. By typing I<enter> the consecutive memory area will be displayed.

Example:

Dump from address A0H:

```
$I>20<Enter>
A0: 00 01 02 03 04 05 06 FF FF FF FF FF FF FF FF FF
B0: 01 02 03 04 05 06 07 FF FF FF FF FF FF FF FF FF
C0: 02 03 04 05 06 07 08 FF FF FF FF FF FF FF FF FF
D0: FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00 00
$
```

F: Fill external data memory

With this command the content of a section of the external data memory (RAM) can be changed. After typing 'F' the '>' prompt is displayed and a hexadecimal address value has to be typed. In response to the typed address the monitor displays the address and its content followed by the '>'-prompt. Now one can enter a new value or leave it as it is. With <Enter> the new (or current) value is stored in memory after which the content of the next address is shown. By typing the space bar the previous location is shown again. Type <Esc> to finish this command.

Example:

Enter byte values in external RAM from address 2300H:

```
$F>2300<Enter>          start at address 2300H
2300= 45 >A9<Enter>     change value 45H into A9H
2301= 6B >11<Enter>     change value 6BH into 11H
2302= FF ><Enter>       do not change value FFH
2303= 00 ><Space>       return to previous address
2302= FF >F0<Enter>     change value FFH into F0H
2303= 00 ><Esc>         ready
$
```

M: Modify internal data memory

With this command the content of a section of the internal data memory (RAM) can be changed. After typing 'M' the '>' prompt is displayed and a hexadecimal address value has to be typed. In response to the typed address the monitor displays the address and its content followed by the '>'-prompt. Now one can enter a new value or leave it as it is. With <Enter> the new (or current) value is stored in memory after which the content of the next address is shown. By typing the space bar the previous location is shown again. Type <Esc> to finish this command.

Example:

Enter byte values in internal RAM from address 70H:

```
$F>70<Enter>            start at address 70H
70= 45 >A9<Enter>       change 45H into A9H
71= 6B ><Enter>         do not change 6BH
72= FF >F0<Enter>       change FFH into F0H
73= 00 >44<Space>       change 00H into 44H
72= F0 >C0<Enter>       change F0H into C0H
73= 44 ><Esc>           stop
$
```

L: Load program

This command is used to load a program that is formatted in an INTEL HEX-file, into the external RAM of a micro-controller system. After pressing the 'L'-key the monitor is waiting for the transmission of a HEX-file. The transmission of the last record of the HEX-file has to be delimited by <Ctrl-Z>.

In the section on the program PRACSYS-51 you can find more information about how to send a HEX-file.

Example:

```
$L                                          download commando
Start transmission or type ^Z to quit.
<PgUp>                                      start transmission with <PgUp>
Send file: DEMO.HEX<Enter>                  enter filename
......                                      one period per received record
Ready.                                      transmission completed
$
```

X: Execute program

This command is used to execute a program or a subroutine. The start address has to be entered at the '>'-prompt. When no start address is entered the value of the Program Counter save location is taken as the start address. An entered start address is saved in the PC save location. The other register save locations do not play a role with respect to this command. Possible assigned breakpoints will not be made active.
After finishing a program or a subroutine (with the RET-instruction) the message "End of program." will be displayed.

Example:

Start program at address 3200H:

```
$X>3200<Enter>
End of program.
$
(re) start program at the current PC address (3200H):
$X><Enter>
End of program.
$
```

S: Subroutine call

This command is used to execute a complete subroutine. The value in the Program Counter save location is used as start address. This must be the address of an LCALL or an ACALL instruction. Nothing happens if the address points to another instruction. Preceding the subroutine call the register save location values are copied to the corresponding CPU-registers. After the execution of the subroutine the possibly changed CPU-register values are copied to the save locations. Assigned breakpoints are not active during execution of a subroutine.
While tracing a program with the T-command a subroutine can be executed entirely with the S-command.
After completion of the execution the values in the register save locations are displayed.

Example:

Execute an LCALL-instruction at the current PC address (3000H):

```
$S
R0 R1 R2 R3 R4 R5 R6 R7 A  B  DPH DPL SP PSW PC
11 00 00 00 00 00 00 04 34 00  10 00  40  00 3003
$
```

G: Go command

With this command a program or a subroutine can be executed. The start address has to be entered at the '>'-prompt. An entered start address will be stored in the Program Counter save

location. If no start address is typed in the value of the PC save location is taken as start address (in this way it is possible to resume executing a program after a breakpoint has been hit or after tracing an single instruction.
Unlike what happens in response to the X-command, the register save location values are copied to the corresponding CPU-registers prior to program execution. Assigned breakpoints are active during execution of the user program (see also the 'B' command).
After finishing a program or a subroutine (with the RET-instruction) the message "End of program." will be displayed followed by a register dump.

Example:

Start a program at address 3200H:

```
$G>3200<Enter>
End of program.
R0 R1 R2 R3 R4 R5 R6 R7 A  B  DPH DPL SP PSW PC
00 00 00 00 00 00 00 00 00 00  00 00  40  00 3200
$
```

Start a program at address 3200H with a breakpoint at address 3070H:

```
$G>3200<Enter>
Breakpoint.
R0 R1 R2 R3 R4 R5 R6 R7 A  B  DPH DPL SP PSW PC
11 22 00 00 00 00 00 02 34 00  10 00  42  00 3070
$
```

Resume the program at the current PC address (3070H):

```
$G><Enter>
End of program.
R0 R1 R2 R3 R4 R5 R6 R7 A  B  DPH DPL SP PSW PC
11 22 00 00 00 00 00 02 34 00  10 00  40  00 3070
$
```

T: Trace one instruction

This command executes one single instruction at the address stored in the Program Counter save location. The register save location values is copied to the corresponding CPU-registers prior to the execution of this one instruction. After the execution of the instruction the possibly changed CPU-register values are copied back to the save locations.
The registers (changed) save locations values are displayed. In this way a program stored in ROM or RAM can be executed step by step. By observing the register values and the content memory locations programming errors can be located or correct operation verified. Program execution may be resumed with one of the command 'T', 'S', 'G', or 'X'.

The trace command was implemented by using the external interrupt INT0 (P3.2, level active low). The trace command sets bit 0 (EX0) and 7 (EA) in the Interrupt Enable Register 0 to 1. After one trace EX0 is cleared (INT0 interrupt disabled) but EA remains 1 (interrupt system enabled). When tracing a program the external interrupt input INT0 is not available for application purposes and external hardware may not control the signal level of P3.2.

Example:

Execute the instruction at the current PC address (3000H):

```
$T
R0 R1 R2 R3 R4 R5 R6 R7 A  B  DPH DPL SP PSW PC
11 00 00 00 00 00 00 04 34 00  10 00  40  00 3202
$
```

R: display Registers

The value of all register save locations can be viewed with this command. The save locations hold the register values that were copied from the CPU-registers after each 'T' and 'S' command or by hitting a breakpoint. Prior to the execution of the commands 'G', 'S', and 'T' the save location values are loaded into the corresponding CPU-registers. The commands 'G' and 'X' loads the Program Counter with PC save location value if no start address was typed. Command 'X' does not load any of the other CPU-registers.

Example:
A register dump immediately after a hardware reset:

```
$R
R0 R1 R2 R3 R4 R5 R6 R7 A  B  DPH DPL SP PSW PC
00 00 00 00 00 00 00 00 00 00  00 00  40  00 0000
$
```

.: Modify register (dot command)

The content of a register save locations can be changed with the '.' command. Type '.' followed by the register name and an <Enter>. The monitor shows the register value followed by the '>' prompt. Now a new value may be entered. By just typing <Enter> the register value is not altered. Use <Backspace> to correct typing errors.

Example:

```
$.PC<Enter>= 0000 >3200<Enter>      change the PC from 0000 to 3200
$.A<Enter>= 56 ><Enter>             do not change register A
$
```

B: set, clear, list Breakpoints

With this command program breakpoints can be assigned to instruction locations, breakpoints can be cleared and displayed.
Breakpoints can be set up to a maximum of four, numbered from 1 to 4. A breakpoint is not assigned (it has been cleared) if it has a value of 0000. An assigned breakpoint has a value greater then 0. Active breakpoints are assigned breakpoints that have been actually placed in the program by the monitor command 'G'.
After switching on the computer all breakpoints are cleared, however after a reset or a jump to address 0000 breakpoints remain assigned but are removed from program memory.

Example:
Clear all breakpoints:

```
$BC
1=0000 2:0000 3:0000 4:0000
$
```

Assign breakpoints 1 and 2:

```
$B1=0000 >3008<Enter>
$B2=0000 >3050<Enter>
$
```

Clear breakpoint 1:

```
$B1=3008 ><Enter>
$
```

Show breakpoint addresses:

```
$B<Enter>
1=0000 2:3050 3:0000 4:0000
$
```

Breakpoints are being used to discover programming errors or to verify correct program execution. When a program hits an active breakpoint, execution is aborted, register values are saved and the command prompt is displayed. Now, register values may be observed or memory contents inspected or changed if necessary.

Program execution is resumed by one of the commands 'T', 'S', or 'G'. Only command 'G' activates the assigned breakpoints by storing breakpoint codes in the program at the assigned breakpoint addresses. A breakpoint address is the address of an instruction (the opcode) where program execution has to stop. Breakpoint codes consist of three bytes: an LCALL instruction followed by the address of the monitor's breakpoint handler.

Before the 'G'-command routine can activate a breakpoint it has to save the three program bytes that will by replaced by the breakpoint code bytes. The saved bytes will be stored in save locations in the 'External monitor RAM area'. After a breakpoint hit the breakpoint code bytes are removed

from the program and the original program code bytes are restored. Obviously the program has to be in RAM if breakpoints have to be used.

The above explanation makes it clear that breakpoint addresses has to differ by 3 at least to prevent unpredictable program behavior. More then one breakpoint at the same address is not allowed.

The 'G'-command does not place an assigned breakpoint at the start address to ensure the execution will resume from there. Knowing that if there is only one breakpoint, program execution resumes without an active breakpoint after hitting that breakpoint for the first time and restarting with the 'G'-command. If this is not desirable there are two solutions to this problem: assign a second breakpoint to another suitable address or trace the instruction at the breakpoint address prior to the 'G'-command. The second solution will not work with an instruction like SJMP $ or something like LOOP LJMP LOOP. The assigned breakpoint at such an instruction will be active only once.

Be careful in the following situation. A breakpoint at the instruction SJMP LOOP (2 bytes) changes also the opcode of the first instruction of subroutine SUBR. It is very likely that the program will behave unpredictable or even crash. By temporarily adding a NOP instruction following the SJMP this problem can be solved.

```
LOOP       LCALL       SUBR
           SJMP        LOOP        ;a breakpoint here will eat
SUBR       first instruction       ;  a byte of this instruction
           //
           RET
```

Appendix 1: Machine Codes

The following tables present an overview of the 8051 instruction set. The column INSTRUCTION shows the assembly-notation. In Column OPERATION shows what happens during execution of an instruction. The (possible) changes in the Processor Status Word are listed in column PSW. Column T shows the number of machine cycles and column L has the instruction length in bytes. Finally column CODE shows the opcode (the first byte) in binary form. The following symbols are used in the tables:

Symbol	Meaning
#	immediate addressing
data	8-bit value
data16	16 bits value
Rn	R0,R1,R2,R3,R4,R5,R6 or R7 in the current selected register bank
rrr	3-bit value 000,001,010,011,100,101,110 of 111
Ri	R0 or R1 in the current selected register bank
@Ri	indirect addressing of an internal memory location using pointer register R0 or R1 all memory locations are accessible using indirect addressing
direct	8-bit internal byte address memory locations 0-127 and SFR locations 128-255 are accessible using direct addressing
bit	8 bit address of a bit in the bit addressable internal data memory or in a bit addressable SFR in data memory: bit address range is 0-127 in SFR area: bit address range is 128-255
rel	2's complement 8 bit offset address relative to the address of the first byte succeeding the instruction rel = FEH means 2 bytes lower rel = 2 means 2 bytes higher
addr11	11-bit address aaa = A10,A9,A8 A7-A0 in second byte
addr16	16-bit address in second (high) byte and third (low) byte
A_n	bit n of register A, n = 0-7
A_{0-3}	bit 0 t/m 3 of register A
not	logical NOT (inversion or 1's complement) bit: not 0 = 1, not 1 = 0 byte: not 11110000 = 00001111
and	logical AND bit: 0 and X = 0, 1 and 1 = 1 byte: 11110000 and 00110011 = 00110000
or	logical OR bit: 1 or X = 1, 0 or 0 = 0 byte: 11110000 or 11001100 = 11111100

exor logical EXCLUSIVE OR
bit: 0 exor 1 = 1, 0 exor 0 = 0, 1 exor 1 = 0
carry from A6 exor carry from A7 gives "1" if unequal
borrow to A6 exor borrow to A7 gives "1" if unequal
byte: 11110000 exor 11001100 = 00111100

(direct) content of memory location with address 'direct'

(Ri) content of memory location with address in Ri

- assignment
  A•Rn means: the content of Rn is copied to A
  (SP) • (direct) means: the content of address 'direct' is copied to the memory location pointed to by the stack pointer
  C•0 means: C-flag is reset to 0

- exchange
  A•Rn means: the contents of Rn and A are exchanged

DATA TRANSFER					
INSTRUCTION	**OPERATION**	**PSW**	**T**	**L**	**CODE**
MOV A,Rn	A•Rn	-	1	1	11101rrr
MOV A,direct	A•(direct)	-	1	2	11100101
MOV A,@Ri	A•(Ri)	-	1	1	1110011i
MOV A,#data	A•data	-	1	2	01110100
MOV Rn,A	Rn•A	-	1	1	11111rrr
MOV Rn,direct	Rn•(direct)	-	2	2	10101rrr
MOV Rn,#data	Rn•data	-	1	2	01111rrr
MOV direct,A	(direct)•A	-	1	2	11110101
MOV direct,Rn	(direct)•Rn	-	2	2	10001rrr
MOV direct,direct	(direct)•(direct)	-	2	3	10000101
MOV direct,@Ri	(direct)•(Ri)	-	2	2	1000011i
MOV direct,#data	(direct)•data	-	2	3	01110101
MOV @Ri,A	(Ri)•A	-	1	1	1111011i
MOV @Ri,direct	(Ri)•(direct)	-	2	2	1010011i
MOV @Ri,#data	(Ri)•data	-	1	2	0111011i
MOV DPTR,#data16	DPTR•data16	-	2	3	10010000
MOVC A,@A+DPTR	A•(A+DPTR)	-	2	1	10010011
MOVC A,@A+PC	A•(A+PC)	-	2	1	10000011
MOVX A,@Ri	A•(Ri)	-	2	1	1110001i
MOVX A,@DPTR	A•(DPTR)	-	2	1	11100000
MOVX @Ri,A	(Ri)•A	-	2	1	1111001i
MOVX @DPTR,A	(DPTR)•A	-	2	1	11110000
PUSH direct	SP•SP+1 (SP)•(direct)	-	2	2	11000000
POP direct	(direct)•(SP) SP•SP-1	-	2	2	11010000
XCH A,Rn	A•Rn	-	1	1	11001rrr
XCH A,direct	A•(direct)	-	1	2	11000101
XCH A,@Ri	A•(Ri)	-	1	1	1100011i
XCHD A,@Ri	A_{0-3}•(Ri_{0-3})	-	1	1	1101011i

ARITHMETIC OPERATIONS					
INSTRUCTION	**OPERATION**	**PSW**	**T**	**L**	**CODE**
ADD A,Rn	A•A+Rn	C•carry from A7 AC•carry from A3 OV•carry from A6 exor carry from A7	1	1	00101rrr
ADD A,direct	A•A+(direct)	C AC OV	1	2	00100101
ADD A,@Ri	A•A+(Ri)	C AC OV	1	1	0010011i
ADD A,#data	A•A+data	C AC OV	1	2	00100100
ADDC A,Rn	A•A+Rn+C	C AC OV	1	1	00111rrr
ADDC A,direct	A•A+(direct)+C	C AC OV	1	2	00110101
ADDC A,@Ri	A•A+(Ri)+C	C AC OV	1	1	0011011i
ADDC A,#data	A•A+data+C	C AC OV	1	2	00110100
SUBB A,Rn	A•A-Rn-C	C•borrow for A7 AC•borrow for A3 OV•borrow for A6 exor borrow for A7	1	1	10011rrr
SUBB A,direct	A•A-(direct)-C	C AC OV	1	2	10010101
SUBB A,@Ri	A•A-(Ri)-C	C AC OV	1	1	1001011i
SUBB A,#data	A•A-data-C	C AC OV	1	2	10010100
INC A	A•A+1	-	1	1	00000100
INC Rn	Rn•Rn+1	-	1	1	00001rrr
INC direct	(direct)•(direct)+1	-	1	2	00000101
INC @Ri	(Ri)•(Ri)+1	-	1	1	0000011i
INC DPTR	DPTR•DPTR+1	-	2	1	10100011
DEC A	A•A-1	-	1	1	00010100
DEC Rn	Rn•Rn-1	-	1	1	00011rrr
DEC direct	(direct)•(direct)-1	-	1	2	00010101
DEC @Ri	(Ri)•(Ri)-1	-	1	1	0001011i
MUL AB	unsigned multiply A•low byte of A*B B•high byte of A*B	C•0 OV•1 if A*B>255 OV•0 if A*B•255	4	1	10100100
DIV AB	unsigned divide A•integer part of A/B B•remainder of A/B	C•0 OV•1 if divide by 0 OV•0 if not	4	1	10000100
DA A	A•decimal adjust A following ADD or ADDC in packed BCD format	C•1 if result>99 C•0 if result•99 OV•0	1	1	11010100

BOOLEAN VARIABLE MANIPULATION					
INSTRUCTION	**OPERATION**	**PSW**	**T**	**L**	**CODE**
CLR C	C•0	C•0	1	1	11000011
CLR bit	(bit)•0	-	1	2	11000010
SETB C	C•1	C•1	1	1	11010011
SETB bit	(bit)•1	-	1	2	11010010
CPL C	C•not C	C•not C	1	1	10110011
CPL bit	(bit)•not(bit)	-	1	2	10110010
ANL C,bit	C•C and (bit)	C	2	2	10000010
ANL C,/bit	C•C and not(bit)	C	2	2	10110000
ORL C,bit	C•C or (bit)	C	2	2	01110010
ORL C,/bit	C•C or not(bit)	C	2	2	10100000
MOV C,bit	C•(bit)	C	1	2	10100010
MOV bit,C	(bit)•C	-	2	2	10010010
JC rel	PC•PC+2 if C=1 then PC•PC+rel	-	2	2	01000000
JNC rel	PC•PC+2 if C=0 then PC•PC+rel	-	2	2	01010000
JB bit,rel	PC•PC+3 if (bit)=1 then PC•PC+rel	-	2	3	00100000
JNB bit,rel	PC•PC+3 if (bit)=0 then PC•PC+rel	-	2	3	00110000
JBC bit,rel	PC•PC+3 if (bit)=1 then PC•PC+rel and (bit)•0	-	2	3	00010000

PROGRAM BRANCHING					
INSTRUCTION	**OPERATION**	**PSW**	**T**	**L**	**CODE**
ACALL addr11	PC•PC+2 SP•SP+1 (SP)•PC_{7-0} SP•SP+1 (SP)•PC_{15-8} PC_{10-0}•addr11	-	2	2	aaa10001
LCALL addr16	PC•PC+3 SP•SP+1 (SP)•PC_{7-0} SP•SP+1 (SP)•PC_{15-8} PC_{15-0}•addr16	-	2	3	00010010
RET	PC_{15-8}•(SP) SP•SP-1 PC_{7-0}•(SP) SP•SP-1	-	2	1	00100010
RETI	PC_{15-8}•(SP) SP•SP-1 PC_{7-0}•(SP) SP•SP-1	-	2	1	00110010
AJMP addr11	PC•PC+2 PC_{10-0}•addr11	-	2	2	aaa00001
LJMP addr16	PC_{15-0}•addr16	-	2	3	00000010
SJMP rel	PC•PC+2 PC•PC+rel	-	2	2	10000000
JMP @A+DPTR	PC•A+DPTR	-	2	1	01110011
JZ rel	PC•PC+2 if A=0 then PC•PC+rel	-	2	2	01100000
JNZ rel	PC•PC+2 if A<>0 then PC•PC+rel	-	2	2	01110000
CJNE A,direct,rel	PC•PC+3 if A<>(direct) then PC•PC+rel if A<(direct) then C•1 else C•0	C	2	3	10110101
CJNE A,#data,rel	PC•PC+3 if A<>data then PC•PC+rel if A<data then C•1 else C•0	C	2	3	10110100
CJNE Rn,#data,rel	PC•PC+3 if Rn<>data then PC•PC+rel if Rn<data then C•1 else C•0	C	2	3	10111rrr

CJNE @Ri,#data,rel	PC•PC+3
	if (Ri)<>data then PC•PC+rel
	if (Ri)<data then C•1 else C•0
		C	2	3	1011011i

DJNZ Rn,rel	PC•PC+2 Rn•Rn-1 if Rn<>0 then PC•PC+rel	-	2	2	11011rrr
DJNZ direct,rel	PC•PC+3 (direct)•(direct)-1 if (direct)<>0 then PC•PC+rel	-	2	3	11010101

LOGICAL OPERATIONS FOR BYTE VARIABLES					
INSTRUCTION	**OPERATION**	**PSW**	**T**	**L**	**CODE**
ANL A,Rn	A•A and Rn	-	1	1	01011rrr
ANL A,direct	A•A and (direct)	-	1	2	01010101
ANL A,@Ri	A•A and (Ri)	-	1	1	0101011i
ANL A,#data	A•A and data	-	1	2	01010100
ANL direct,A	(direct)•(direct) and A	-	1	2	01010010
ANL direct,#data	(direct)•(direct) and data	-	2	3	01010011
ORL A,Rn	A•A or Rn	-	1	1	01001rrr
ORL A,direct	A•A or (direct)	-	1	2	01000101
ORL A,@Ri	A•A or (Ri)	-	1	1	0100011i
ORL A,#data	A•A or data	-	1	2	01000100
ORL direct,A	(direct)•(direct) or A	-	1	2	01000010
ORL direct,#data	(direct)•(direct) or data	-	2	3	01000011
XRL A,Rn	A•A exor Rn	-	1	1	01101rrr
XRL A,direct	A•A exor (direct)	-	1	2	01100101
XRL A,@Ri	A•A exor (Ri)	-	1	1	0110011i
XRL A,#data	A•A exor data	-	1	2	01100100
XRL direct,A	(direct)•(direct) exor A	-	1	2	01100010
XRL direct,#data	(direct)•(direct) exor data	-	2	3	01100011
CLR A	A•0	-	1	1	11100100
CPL A	A_n•not A_n n=0-7	-	1	1	11110100
RL A	A_{n+1}•A_n n=0-6 A_0•A_7	-	1	1	00100011
RLC A	A_{n+1}•A_n n=0-6 A_0•C C•A_7	-	1	1	00110011
RR A	A_n•A_{n+1} n=0-6 A_7•A_0	-	1	1	00000011
RRC A	A_n•A_{n+1} n=0-6 A_7•C C•A_0	-	1	1	00010011
SWAP A	A_{3-0}•A_{7-4}	-	1	1	11000100
NOP	No operation	-	1	1	00000000

Appendix 2: ASCII character codes table

HEX	DEC	CHAR	CNTL	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR
00	0	NUL		20	32	SPACE	40	64	@	60	96	`
01	1	SOH	^A	21	33	!	41	65	A	61	97	a
02	2	STX	^B	22	34	"	42	66	B	62	98	b
03	3	ETX	^C	23	35	#	43	67	C	63	99	c
04	4	EOT	^D	24	36	$	44	68	D	64	100	d
05	5	ENQ	^E	25	37	%	45	69	E	65	101	e
06	6	ACK	^F	26	38	&	46	70	F	66	102	f
07	7	BEL	^G	27	39	'	47	71	G	67	103	g
08	8	BS	^H	28	40	(	48	72	H	68	104	h
09	9	HT	^I	29	41	)	49	73	I	69	105	I
0A	10	LF	^J	2A	42	*	4A	74	J	6A	106	j
0B	11	VT	^K	2B	43	+	4B	75	K	6B	107	k
0C	12	FF	^L	2C	44	,	4C	76	L	6C	108	l
0D	13	CR	^M	2D	45	-	4D	77	M	6D	109	m
0E	14	SO	^N	2E	46	.	4E	78	N	6E	110	n
0F	15	SI	^O	2F	47	/	4F	79	O	6F	111	o
10	16	DLE	^P	30	48	0	50	80	P	70	112	p
11	17	DC1	^Q	31	49	1	51	81	Q	71	113	q
12	18	DC2	^R	32	50	2	52	82	R	72	114	r
13	19	DC3	^S	33	51	3	53	83	S	73	115	s
14	20	DC4	^T	34	52	4	54	84	T	74	116	t
15	21	NAK	^U	35	53	5	55	85	U	75	117	u
16	22	SYN	^V	36	54	6	56	86	V	76	118	v
17	23	ETB	^W	37	55	7	57	87	W	77	119	w
18	24	CAN	^X	38	56	8	58	88	X	78	120	x
19	25	EM	^Y	39	57	9	59	89	Y	79	121	y
1A	26	SUB	^Z	3A	58	:	5A	90	Z	7A	122	z
1B	27	ESC		3B	59	;	5B	91	[	7B	123	{
1C	28	FS		3C	60	<	5C	92	\	7C	124	|
1D	29	GS		3D	61	=	5D	93	]	7D	125	}
1E	30	RS		3E	62	>	5E	94	^	7E	126	~
1F	31	US		3F	63	?	5F	95	_	7F	127	DEL

ACK=Acknowledge
BEL=Bell
BS =Backspace
CAN=Cancel
CR =Carriage Return
DC1=Direct Control 1
DC2=Direct Control 2
DC3=Direct Control 3
DC4=Direct Control 4
DLE=Data Link Escape
EM =End of Medium
ENQ=Enquiry
EOT=End Of Transmission
ESC=Escape
ETB=End Transmission Block
ETX=End Text
FF =Form Feed
FS =Form Separator
GS =Group Separator
HT =Horizontal Tab
LF =Line Feed
NAK=Negative Acknowledge
NUL=Null
RS =Record Separator
SI =Shift In
SO =Shift Out
SOH=Start Of Heading
STX=Start Text
SUB=Substitute
SYN=Synchronous Idle
US =Unit Separator
VT =Vertical Tab

Appendix

A-2

Interrupts

The 8XC552 has fifteen interrupt sources, each of which can be assigned one of two priority levels, as shown in Figure 27. The five interrupt sources common to the 80C51 are the external interrupts ($\overline{INT0}$ and $\overline{INT1}$), the timer 0 and timer 1 interrupts (IT0 and IT1), and the serial I/O interrupt (RI or TI). In the 8XC552, the standard serial interrupt is called SIO0. Since the subsystems which create these interrupts are identical on both parts, their functionality is likewise identical. The only differences are the locations of the enable and priority register configurations and the priority structure. This is detailed below along with the specifics of the interrupts unique to the 8XC552.

The eight Timer T2 interrupts are generated by flags CTI0-CT13, CMI0-CMI2, and by the logical OR of flags T2OV and T2BO. Flags CTI0 to CT13 are set by input signals CT0I to CT3i. Flags CMI0 to CMI2 are set when a match occurs between Timer T2 and the compare registers CM0, CM1, and CM2. When an 8-bit or 16-bit overflow occurs, flags T2BO and T2OV are set, respectively. These nine flags are not cleared by hardware and must be reset by software to avoid recurring interrupts.

The ADC interrupt is generated by the ADCI flag in the ADC control register (ADCON). This flag is set when an ADC conversion result is ready to be read. ADCI is not cleared by hardware and must be reset by software to avoid recurring interrupts.

The SIO1 (I^2C) interrupt is generated by the SI flag in the SIO1 control register (S1CON). This flag is set when S1STA is loaded with a valid status code.

The ADCI flag may be reset by software. It cannot be set by software. All other flags that generate interrupts may be set or cleared by software, and the effect is the same as setting or resetting the flags by hardware. Thus, interrupts may be generated by software and pending interrupts can be canceled by software.

Interrupt Enable Registers: Each interrupt source can be individually enabled or disabled by setting or clearing a bit in the interrupt enable special function registers IEN0 and IEN1. All interrupt sources can also be globally enabled or disabled by setting or clearing bit EA in IEN0. The interrupt enable registers are described in Figures 28 and 29.

Interrupt Priority Structure: Each interrupt source can be assigned one of two priority levels. Interrupt priority levels are defined by the interrupt priority special function registers IP0 and IP1. IP0 and IP1 are described in Figures 30 and 31.

Interrupt priority levels are as follows:
"0"—low priority
"1"—high priority

A low priority interrupt may be interrupted by a high priority interrupt. A high priority interrupt cannot be interrupted by any other interrupt source. If two requests of different priority occur simultaneously, the high priority level request is serviced. If requests of the same priority are received simultaneously, an internal polling sequence determines which request is serviced. Thus, within each priority level, there is a second priority structure determined by the polling sequence. This second priority structure is shown in Table 8.

The above Priority Within Level structure is only used when there are simultaneous requests of the same priority level.

Interrupt Handling: The interrupt sources are sampled at S5P2 of every machine cycle. The samples are polled during the following machine cycle. If one of the flags was in a set condition at S5P2 of the previous machine cycle, the polling cycle will find it and the interrupt system will generate an LCALL to the appropriate service routine, provided this hardware-generated LCALL is not blocked by any of the following conditions:

1. An interrupt of higher or equal priority level is already in progress.
2. The current machine cycle is not the final cycle in the execution of the instruction in progress. (No interrupt request will be serviced until the instruction in progress is completed.)
3. The instruction in progress is RETI or any access to the interrupt priority or interrupt enable registers. (No interrupt will be serviced after RETI or after a read or write to IP0, IP1, IE0, or IE1 until at least one other instruction has been subsequently executed.)

The polling cycle is repeated with every machine cycle, and the values polled are the values present at S5P2 of the previous machine cycle. Note that if an interrupt flag is active but is not being responded to because of one of the above conditions, and if the flag is inactive when the blocking condition is removed, then the blocked interrupt will not be serviced. Thus, the fact that the interrupt flag was once active but not serviced is not remembered. Every polling cycle is new.

The processor acknowledges an interrupt request by executing a hardware-generated LCALL to the appropriate service routine. In some cases it also clears the flag which generated the interrupt, and in others it does not. It clears the Timer 0, Timer 1, and external interrupt flags. An external interrupt flag (IEO or IE1) is cleared only if it was transition-activated. All other interrupt flags are not cleared by hardware and must be cleared by the software. The LCALL pushes the contents of the program counter on to the stack (but it does not save the PSW) and reloads the PC with an address that depends on the source of the interrupt being vectored to as shown in Table 9.

Execution proceeds from the vector address until the RETI instruction is encountered. The RETI instruction clears the "priority level active" flip-flop that was set when this interrupt was acknowledged. It then pops the top two bytes from the stack and reloads the program counter. Execution of the interrupted program continues from where it was interrupted.

Analog-to-Digital Conversion: Figure 35 shows the elements of a successive approximation (SA) ADC. The ADC contains a DAC which converts the contents of a successive approximation register to a voltage (VDAC) which is compared to the analog input voltage (Vin). The output of the comparator is fed to the successive approximation control logic which controls the successive approximation register. A conversion is initiated by setting ADCS in the ADCON register. ADCS can be set by software only or by either hardware or software.

The software only start mode is selected when control bit ADCON.5 (ADEX) = 0. A conversion is then started by setting control bit ADCON.3 (ADCS). The hardware or software start mode is selected when ADCON.5 = 1, and a conversion may be started by setting ADCON.3 as above or by applying a rising edge to external pin STADC. When a conversion is started by applying a rising edge, a low level must be applied to STADC for at least one machine cycle followed by a high level for at least one machine cycle.

The low-to-high transition of STADC is recognized at the end of a machine cycle, and the conversion commences at the beginning of the next cycle. When a conversion is initiated by software, the conversion starts at the beginning of the machine cycle which follows the instruction that sets ADCS. ADCS is actually implemented with two flip-flops: a command flip-flop which is affected by set operations, and a status flag which is accessed during read operations.

The next two machine cycles are used to initiate the converter. At the end of the first cycle, the ADCS status flag is set and a value of "1" will be returned if the ADCS flag is read while the conversion is in progress. Sampling of the analog input commences at the end of the second cycle.

During the next eight machine cycles, the voltage at the previously selected pin of port 5 is sampled, and this input voltage should be stable in order to obtain a useful sample. In any event, the input voltage slew rate must be less than 10V/ms in order to prevent an undefined result.

The successive approximation control logic first sets the most significant bit and clears all other bits in the successive approximation register (10 0000 0000B). The output of the DAC (50% full scale) is compared to the input voltage Vin. If the input voltage is greater than VDAC, then the bit remains set; otherwise it is cleared.

The successive approximation control logic now sets the next most significant bit (11 0000 0000B or 01 0000 0000B, depending on the previous result), and VDAC is compared to Vin again. If the input voltage is greater than VDAC, then the bit being tested remains set; otherwise the bit being tested is cleared. This process is repeated until all ten bits have been tested, at which stage the result of the conversion is held in the successive approximation register. Figure 36 shows a conversion flow chart. The bit pointer identifies the bit under test. The conversion takes four machine cycles per bit.

The end of the 10-bit conversion is flagged by control bit ADCON.4 (ADCI). The upper 8 bits of the result are held in special function register ADCH, and the two remaining bits are held in ADCON.7 (ADC.1) and ADCON.6 (ADC.0). The user may ignore the two least significant bits in ADCON and use the ADC as an 8-bit converter (8 upper bits in ADCH). In any event, the total actual conversion time is 50 machine cycles for the 8XC552 or 24 machine cycles for the 8XC562. ADCI will be set and the ADCS status flag will be reset 50 (or 24) cycles after the command flip-flop (ADCS) is set.

Control bits ADCON.0, ADCON.1, and ADCON.2 are used to control an analog multiplexer which selects one of eight analog channels (see Figure 37). An ADC conversion in progress is unaffected by an external or software ADC start. The result of a completed conversion remains unaffected provided ADCI = logic 1; a new ADC conversion already in progress is aborted when the idle or power-down mode is entered. The result of a completed conversion (ADCI = logic 1) remains unaffected when entering the idle mode.

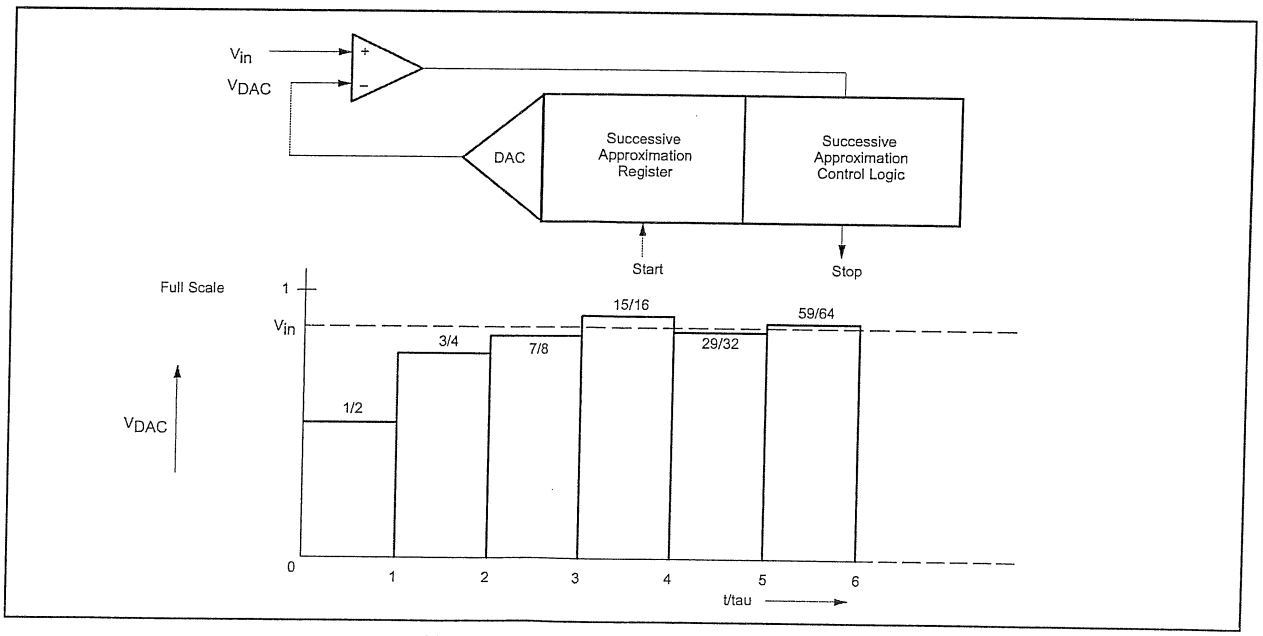


Figure 35. Successive Approximation ADC

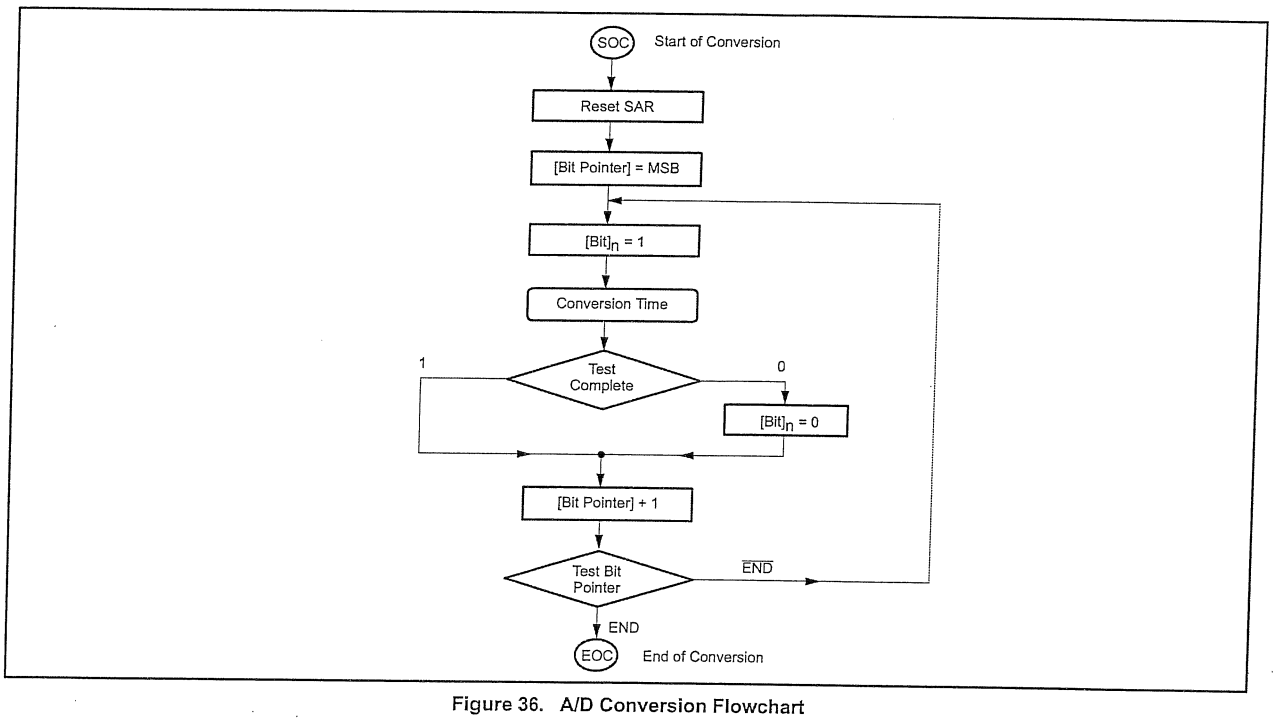


Figure 36. A/D Conversion Flowchart

ADCON (C5H)

7	6	5	4	3	2	1	0
ADC.1	ADC.0	ADEX	ADCI	ADCS	AADR2	AADR1	AADR0
(MSB)							(LSB)

Bit	Symbol	Function
ADCON.7	ADC.1	Bit 1 of ADC result
ADCON.6	ADC.0	Bit 0 of ADC result
ADCON.5	ADEX	Enable external start of conversion by STADC 0 = Conversion can be started by software only (by setting ADCS) 1 = Conversion can be started by software or externally (by a rising edge on STADC)
ADCON.4	ADCI	ADC interrupt flag: this flag is set when an A/D conversion result is ready to be read. An interrupt is invoked if it is enabled. The flag may be cleared by the interrupt service routine. While this flag is set, the ADC cannot start a new conversion. ADCI cannot be set by software.
ADCON.3	ADCS	ADC start and status: setting this bit starts an A/D conversion. It may be set by software or by the external signal STADC. The ADC logic ensures that this signal is HIGH while the ADC is busy. On completion of the conversion, ADCS is reset immediately after the interrupt flag has been set. ADCS cannot be reset by software. A new conversion may not be started while either ADCS or ADCI is high.

ADCI	ADCS	ADC Status
0	0	ADC not busy; a conversion can be started
0	1	ADC busy; start of a new conversion is blocked
1	0	Conversion completed; start of a new conversion requires ADCI=0
1	1	Conversion completed; start of a new conversion requires ADCI=0

If ADCI is cleared by software while ADCS is set at the same time, a new A/D conversion with the same channel number may be started. But it is recommended to reset ADCI *before* ADCS is set.

ADCON.2 ADCON.1 ADCON.0	AADR2 AADR1 AADR0	Analogue input select: this binary coded address selects one of the eight analogue port bits of P5 to be input to the converter. It can only be changed when ADCI and ADCS are both LOW.

AADR2	AADR1	AADR0	Selected Analog Channel
0	0	0	ADC0 (P5.0)
0	0	1	ADC1 (P5.1)
0	1	0	ADC2 (P5.2)
0	1	1	ADC3 (P5.3)
1	0	0	ADC4 (P5.4)
1	0	1	ADC5 (P5.5)
1	1	0	ADC6 (P5.6)
1	1	1	ADC7 (P5.7)

Figure 37. ADC Control Register (ADCON)

B

Appendix B: Keil 8051 development tools

Keil 8051 Development Tools

This document contains a summary of the tutorials for the Keil 8051 development environment.

Document Conventions

This document uses the following conventions:

Examples	Description
README.TXT	Bold capital text is used for the names of executable programs, data files, source files, environment variables, and commands you enter at the MS-DOS Command prompt. This text usually represents commands that you must type in literally. For example: **CLS DIR BL51.EXE** Note that you are not required to enter these commands using all capital letters.
`Courier`	Text in this typeface is used to represent information that displays on screen or prints at the printer. This typeface is also used within the text when discussing or describing command line items.
Variables	Text in italics represents information that you must provide. For example, project file in a syntax string means that you must supply the actual project file name. Occasionally, italics are also used to emphasize words in the text.
Elements that Repeat…	Ellipsis (…) are used to indicate an item that may be repeated.
Omitted code . . .	Vertical ellipses are used in source code listings to indicate that a fragment of the program is omitted. For example: **void main (void) {** **.** **.** **.** **while (1);**
[*Optional Items*]	Double brackets indicate optional arguments in command-line and option fields. For example: **C51 TEST.C PRINT [(filename)]**
{ *opt1* | *opt2* }	Text contained within braces, separated by a vertical bar represents a group of items from which one must be chosen. The braces enclose all of the choices and the vertical bars separate the choices. One item in the list must be selected.
Keys	Text in this typeface represents actual keys on the keyboard. For example, “Press **Enter** to continue.”

Content

1. Introduction

1.1 8051 Microcontroller Family

The 8051 has been available since the early 1980's. With a wide variety of outstanding features and peripherals, the 8051 CPU core is destined to see service well into the next century. More than 200 different 8051 derivatives are available today from a variety of chip vendors. More than half of all embedded projects with a CPU use members of the 8051 microcontroller family. As an embedded processor, the 8051 has no equal.
A typical 8051 family member contains the 8051 CPU core, data memory, code memory, and some versatile peripheral functions. A flexible memory interface lets you expand the capabilities of the 8051 using standard peripherals and memory devices.

1.2 8051 Development Tools

This manual provides information for using the following 8051 development tools:

- C51 Optimizing C Compiler,
- A51 Macro Assembler,
- BL51 Linker/Locator,
- OH51 Object-Hex Converter,
- AC51 Hexfile Start Address Converter.

2 A51 Macro Assembler

The A51 assembler is a macro assembler for the 8051 microcontroller family. It translates symbolic assembly language mnemonics into relocatable object code where the utmost speed, small code size, and hardware control are critical. The A51 assembler supports symbolic access to all features of the 8051 architecture and is configurable for the numerous 8051 derivatives.

2.1 Functional Overview

The A51 assembler translates an assembler source file into a relocatable object module containing the corresponding machine code. In addition to the object file, the A51 assembler generates a list file which may optionally include symbol table and cross reference information.

The source file is translated during two passes. While reading the file for the first time the assembler stores encountered labels and their corresponding addresses in a 'symbol table' and translates instructions into machine code. During the second pass references to labels are resolved and output files are generated.
There are five different memory types available:

Memory Type	Description
CODE	Program memory (64 Kbytes); accessed by opcode MOVC @A+DPTR.
DATA	Directly addressable internal data memory; fastest access to variables (128 bytes).
IDATA	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
BIT	Bit-addressable internal data memory; allows mixed bit and byte access (16 bytes).
XDATA	External data memory (64 Kbytes); accessed by opcode MOVX @DPTR.

The assembler keeps track of its most important task - where to put what in program memory - by means of a pointer, called the 'location counter'. After each translation of an instruction the location counter marks the location (memory address) of the next byte to be stored in program memory. Note that the assembler's location counter and the processor's program counter have completely different functions.
Apart from storing machine code in program memory the assembler can be instructed to allocate memory of any type or to fill program memory with data. If the computer system has separate program and external data memory areas, data placed in program memory can be read only, not written. However, if the memory areas are overlapping there is no physical difference in the actual storage location. Consequently data stored in program memory can be accessed (read and written) as if it had been stored in data memory.

2.2 Assembly line format

An assembly source-line generally consists of three parts:

```
Label:      Instruction          Comment
```

Example:

```
Lab_1:   MOV    A,#12   ;load accumulator with number
```

If the first character in a line is one of the letters A to Z, or a to z, the first symbol of the line is taken as a label. Labels may be up to 32 characters long, and may contain the underscore symbol **'_'** and the question mark symbol **'?'**. If the first character is a space or a tab character, the assembler decides that there is no label. Labels must be delimited by a colon symbol **':'**. Note that a delimiting colon is not a part of the label and is not counted either.

The assembler is not case sensitive with respect to both labels and instructions, which means that upper case and lower case characters are recognized as identical characters.

When a semicolon symbol **';'** appears anywhere in the line, the text following it is interpreted as comment.

2.3 Arithmetic expressions

Numerical constants and character constants are arithmetic expressions. Also labels are arithmetic expressions.

When expr1 and expr2 are arithmetic expressions, then the following operators may be used to form more complex expressions:

Operator	Function	Expression
-	unary minus (negative number)	-expr1
+	Sum	expr1 + expr2
-	Difference	expr1 – expr2
*	Product	expr1 * expr2
/	Integer division (quotient part of divide result)	expr1 / expr2
mod	Modulo division (remainder part of divide result)	expr1 mod expr2

The operator precedence order is: * , / , and mod followed by + and - .
The assembler recognizes parentheses **'('** and **')'** in expressions. Parentheses are used to change the precedence order.

The integer range is 0 to 65535. Overflow of intermediate calculation results causes erroneous expression values.

The dollar symbol **'$'** is a special arithmetic expression, its value is the value of the location counter.

Examples:

```
3+7*2     gives 17
(3+7)*2   gives 20
17/3      gives  5
17 mod 3  gives  2
MOV  R0,#(2300H + (115*3)) / 256
MOV  DPTR,#LABEL+6
```

2.4 Notation of numerical and alphabetic constants

Constants are fixed numerical or alphabetic values.

Constant type	Format	Examples
Decimal numbers	normal notation (no suffix)	0 00001 65536
Hexadecimal numbers	marked by suffix H, must start with a number 0 to 9	0CH 1A3H 0A563H
Binary numbers	marked by suffix B	11B 011B 00000011B
Character constants	a character enclosed by single quotes	'A' '0' '#'
String constants	a string of characters enclosed by single quotes	'XYZ' '1234H' 'this is text'

2.5 Pseudo Instructions

The following pseudo-instructions are assembler directives. They control the assembler but do not generate executable machine code.

```
          ORG       expr
          DB        expr,expr,expr ...
          DW        expr,expr,expr ...
          DS        expr
          DBIT      expr
Label     EQU       expr
          EXTRN     Label
          PUBLIC    Label
Label     SEGMENT   memory_type
          RSEG      label
          END
```

Short description:

- The expression `'expr'` is evaluated to a number.
- **ORG** initializes (or sets) the location counter with the value of `'expr'`.
- **SEGMENT** defines a memory segment.
- **RSEG** switches to a memory segment.
- **DB** inserts bytes in a program and advances the location counter accordingly.
- **DW** inserts double bytes in a program and advances the location counter accordingly.
- **DS** only advances the location counter with the value of `'expr'`. No data is stored.
- **DBIT** allocates a number of bits equal to the value of `'expr'`.
- **EQU** assigns the value of `'expr'` to a symbol.
- **EXTRN** imports symbols from other modules.
- **PUBLIC** exports symbols to other modules.
- **END** marks the end of a source file.

2.5.1 ORG (Origin)

The instruction

```
          ORG       expr
```

assigns the value of the arithmetic expression to the location counter. This instruction is used to make the program start or continue at a specific location in the memory. In the next example `ORG` is used to define the start address of a program at location 100H.

Example:

```
          ORG       100H
START:    LJMP      SOMEWHERE
```

2.5.2 SEGMENT

The instruction

```
LABEL     SEGMENT   memory_type
```

defines a memory segment. The `memory_type` specifies in which part of memory the generated code for the segment must be located (CODE, DATA, IDATA, BIT or XDATA).

In the next example `SEGMENT` is used to define a memory segment named `data_seg` in the directly addressable internal data memory.

Example:

```
data_seg  SEGMENT   DATA
```

2.5.3 RSEG

The instruction

```
          RSEG      LABEL
```

forces the assembler to switch to the specified memory segment. Subsequent (pseudo-) instructions will refer to this segment.

Example:

```
          RSEG      code_seg
```

2.5.4 DB (Define Byte)

The instruction

```
[label]    DB        expr,expr,....
```

saves the values of the arithmetic expressions as bytes in the program memory (CODE), and increments the location counter as required. String constants are also allowed. The next example shows how bytes and string values may be combined.

Example:

```
COMMAND:   DB        13,10,'COMMAND',0   ;CR,LF,String,Delimiter
```

This line places the (hex) byte values 0D, 0A, 43, 4F, 4D, 4D, 41, 4E, 44, 00 in program memory in a consecutive order. The zero-byte marks the end of the string. An optional label represents the address of the first byte of the string.

2.5.5 DW (Define Word)

The instruction

```
[label]    DW        expr,expr,....
```

saves the values of the arithmetic expressions as two byte values in the program memory (CODE), and increments the location counter as required. The byte order is high byte at the current location counter followed by the low byte. String constants holding one or two characters are also allowed. The following example shows the use of this directive. A label is optional.

Example:

```
           DW        234H,0ABCDH,'AB','C'
```

resulting in the word values 0234, ABCD, 4142, 0043 (hex). The consecutive byte values in program memory are 12, 34, 0A, BC, 41, 42, 00, 43 (hex).

2.5.6 DS (Define Storage)

The instruction

```
[label]    DS        expr
```

is used to allocate memory (CODE, DATA, IDATA or XDATA). The number of allocated bytes is equal to the value of the arithmetic expression. If the program is to be placed in external RAM and program memory and data memory are physically identical the defined storage area may be used to store variables. In the next example DS is used to define a data storage area. The label is optional.
Example:

```
           RSEG      ExternalDataSegment  ;defined elsewhere
NUMBER:    DS        2     ;reserve 2 bytes for variable NUMBER
RESULT:    DS        4     ;reserve 4 bytes for variable RESULT
KEY:       DS        1     ;reserve 1 byte for variable KEY
           RSEG      InternalDirectDataSegment  ;defined elsewhere
DATA1:     DS        1     ;reserve 1 byte for variable DATA1
           RSEG      InternalIndirectDataSegment  ;defined elsewhere
IDATA1:    DS        1     ;reserve 1 byte for variable IDATA1
```

To read or write a variable in external data memory use the MOVX instruction:

```
           MOV       DPTR,#RESULT ;initialize Data Pointer first
           MOVX      A,@DPTR      ;read first byte of RESULT
           MOVX      @DPTR,A      ;write first byte of RESULT
```

Directive DS may also be used to manage the internal RAM memory (DATA or IDATA). To read or write a variable in internal RAM use the MOV instruction with direct or indirect addressing:

```
        MOV     A,DATA1     ;read variable DATA1 in direct RAM
        MOV     R0,#IDATA1  ;initialize Memory Pointer first
        MOV     @R0,A       ;write variable DATA1 in indirect RAM
```

Note that the same method is used in all three cases. However, variables have to be accessed differently.

2.5.7 DBIT (Define BIT Storage)

The instruction

```
        DBIT    expr
```

Is used to allocate bit addressable memory (BIT). The number of allocated bits is equal to the value of the arithmetic expression. In the next example DBIT is used to define a bit variable. A label is optional.

Example:

```
FLAG:   DBIT    1     ;reserve 1 bit for variable FLAG
```

2.5.8 EQU (Equate)

The instruction

```
NAME    EQU     expr
```

assigns the value of the arithmetic expression to identifier NAME. In the next example is shown how to use it to define constants.

Example:

```
MIN     EQU     100
MAX     EQU     1000+MIN-1  ;evaluates a simple expression
```

An alternative method to organize variables space in internal RAM uses the EQU directive in stead of the DS directive. In the next example EQU is used to define the addresses in internal RAM of several variables.

Example:

```
POINTER EQU     50H   ;reserve 2 bytes for variable POINTER
RESULT  EQU     52H   ;reserve 4 bytes for variable RESULT
KEY     EQU     56H   ;reserve 1 byte for variable KEY
```

The same method can be used to define space for variables in external data memory. Again, the MOVX instruction has to be used to access variables in external RAM.

2.5.9 EXTRN

The instruction

```
EXTRN     type      (label,label,....)
```

imports symbols from another modules. It allows symbols to be referenced in the current module though the symbols are defined in another module. The type of the symbol can be any memory type (CODE, DATA, IDATA, BIT or XDATA) or NUMBER (if the symbol is defined using EQU). There is no label. EXTRN may be placed in the labelfield.

Example:

```
EXTRN     DATA      (COUNTER1,COUNTER2)
                                ;variables COUNTER1 and COUNTER2 are
                                ;defined in another module
EXTRN     CODE      (MULTIPLY)  ;function MULTIPLY is defined in
                                ;another module
```

2.5.10 PUBLIC

The instruction

```
PUBLIC    label,label,....
```

exports labels to other modules. It allows labels to be referenced by other modules though the labels are defined in the current module. In the next example is shown how to use it to export labels to other modules. There is no label. EXTRN may be placed in the labelfield.

Example:

```
PUBLIC    BUFFER                ;label BUFFER can be used in other
                                ;modules
```

2.5.11 END

The instruction

```
          END
```

marks the end of the source file. The assembler discards all lines following the END directive.

2.6 Bit addressing

The bit addressable locations may be specified with their actual bit addresses.

Example:

```
          CLR       21H         ;clear bit 1 of location 20H
          CPL       0E7H        ;complement bit 7 of the accumulator
          JB        95H,label   ;jump if port 1 bit 5 is logical '1'
```

An alternative way uses the assemblers bit specifier '.' following the byte address in which the bit is located.

Example:

```
ACC       EQU       0E0H        ;address of accumulator
P1        EQU       090H        ;address of port P1
SW5       EQU       P1.5        ;bit address of port P1 bit 5
          CLR       20H.1       ;clear bit 1 of location 20H
          CPL       ACC.7       ;complement bit 7 of the accumulator
          JB        P1.5,label  ;jump if port P1 bit 5 is logical '1'
          JB        SW5,label   ;jump if port P1 bit 5 is logical '1'
```

The last line in this example shows the use of a fully symbolic bit address to specify a bit operand. This method has clearly many advantages over the blunt way of bit addressing used in the previous example. Be careful not to use instructions for bit addressing when accessing byte locations. The results will be unpredictable.

Example:

```
ACC       EQU       0E0H  ;address of accumulator
          CLR       ACC   ;does not clear the whole accumulator
          CLR       A     ;this clears the accumulator
```

2.7 Configuration

The A51 assembler supports all members of the 8051 family. The special function register (SFR) set of the 8051 is predefined. However, the **NOMOD51** control lets you override these definitions with processor-specific include files.
The A51 assembler is shipped with include files for the 8051, 8051F*x*, 8051GB, 8052, 80152, 80451, 80452, 80515, 80C517, 80C515A, 80C517A, 8x552, 8xC592, 8xCL781, 8xCL410 and 80C320 microcontrollers. You can easily create include files for other 8051 family members.

2.8 Listing File Example

The following example shows a listing file generated by the A51 assembler during assembly. The listing file contains source code, machine code generated, directive information, and a symbol table.

The column "LOC" (Location) indicates the value of the location (address) counter (4-digit hexadecimal).
The column "OBJ" indicates the generated object code in hexadecimal notation.
The column "LINE" contains the line number of the source text, in decimal notation.
The column "SOURCE" shows the assembly-language source code.

```
A51 MACRO ASSEMBLER  Test Program     07/01/95 08:00:00 PAGE   1

DOS MACRO ASSEMBLER A51 V5.02
OBJECT MODULE PLACED IN SAMPLE.OBJ
ASSEMBLER INVOKED BY: C:\C51\BIN\A51.EXE SAMPLE.A51 XREF

LOC  OBJ    LINE  SOURCE
               1  $TITLE ('Test Program')
               2  NAME SAMPLE
               3
               4  EXTRN   CODE (PUT CRLF, PUTSTRING, InitSerial)
               5  PUBLIC  TXTBIT
               6
               7  PROG    SEGMENT     CODE
               8  CONST   SEGMENT     CODE
               9  BITVAR  SEGMENT     BIT
              10
----          11          CSEG  AT    0
              12
0000 020000 F 13  Reset:  JMP   Start
              14
----          15          RSEG  PROG
              16  ; *****
0000 120000 F 17  Start:  CALL  InitSerial ;Init Serial Interface
              18
              19  ; This is the main program. It is an endless
              20  ; loop which displays a text on the console.
0003 C200   F 21          CLR   TXTBIT ; read from CODE
0005 900000 F 22  Repeat: MOV   DPTR,#TXT
0008 120000 F 23          CALL  PUTSTRING
000B 120000 F 24          CALL  PUT_CRLF
000E 80F5     25          SJMP  Repeat
              26  ;
----          27          RSEG  CONST
0000 54455354 28  TXT:    DB    'TEST PROGRAM',00H
0004 2050524F
0008 4752414D
000C 00
              29
              30
              31
----          32          RSEG  BITVAR ; TXTBIT=0 read from CODE
0000          33  TXTBIT: DBIT  1      ; TXTBIT=1 read from XDATA
              34
              35          END

XREF SYMBOL TABLE LISTING
---- ------ ----- -------

N A M E            T Y P E   V A L U E   ATTRIBUTES / REFERENCES

BITVAR . . . . . B SEG     0001H       REL=UNIT   9# 32
CONST. . . . . . C SEG     000DH       REL=UNIT   8# 27
INITSERIAL . . . C ADDR    -----       EXT   4# 17
PROG . . . . . . C SEG     0010H       REL=UNIT   7# 15
PUTSTRING. . . . C ADDR    -----       EXT   4# 23
PUT_CRLF . . . . C ADDR    -----       EXT   4# 24
REPEAT . . . . . C ADDR    0005H   R   SEG=PROG   22# 25
RESET. . . . . . C ADDR    0000H   A      13#
SAMPLE . . . . . N NUMB    -----          2
START. . . . . . C ADDR    0000H   R   SEG=PROG   13 17#
TXT. . . . . . . C ADDR    0000H   R   SEG=CONST   22 28#
TXTBIT . . . . . B ADDR    0000H.0 R   SEG=BITVAR   5 5 21 33#

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)
```

The A51 assembler produces a listing file with page numbers as well as the time and date of the assembly. Remarks about the assembler invocation and the object file output are displayed in this listing.

Typical programs start with EXTERN, PUBLIC, and SEGMENT directives.

The listing file includes a line number for each source line.

If a source line generates code, the HEX values are displayed at the beginning of the line.

Error messages and Warning messages are included in the listing file. The position of each error is clearly marked.

The **XREF** assembler option produces a cross reference list. The cross reference report shows all symbols and the line numbers in which they are used. The line number where the symbol is defined is marked with a pound symbol ('#').

The register banks used, and the total number of warnings and errors is stated at the end of the listing file.

3 C51 Optimizing C Cross Compiler

The C programming language is a general-purpose programming language that provides code efficiency, elements of structured programming, and a rich set of operators. C is not a *big* language and is not designed for any one particular area of application. Its generality, combined with its absence of restrictions, make C a convenient and effective programming solution for a wide variety of software tasks. Many applications can be solved more easily and efficiently with C than with other more specialized languages.
The C51 optimizing cross compiler for the MS-DOS operating system is a complete implementation of the ANSI (American National Standards Institute) standard for the C language. The C51 compiler generates code for the 8051 microprocessor but is not a universal C compiler adapted for the 8051 target. It is a ground-up implementation dedicated to generating extremely fast and compact code for the 8051 microprocessor.
For most 8051 applications, the C51 compiler gives software developers the flexibility of programming in C while matching the code efficiency and speed of assembly language. Using a high-level language like C has many advantages over assembly language programming. For example:

- Knowledge of the processor instruction set is not required. A rudimentary knowledge of the 8051's memory architecture is desirable but not necessary.
- Register allocation and addressing mode details are managed by the compiler.
- The ability to combine variable selection with specific operations improves program readability.
- Keywords and operational functions that more nearly resemble the human thought process can be used.
- Program development and debugging times are dramatically reduced when compared to assembly language programming.
- The library files that are supplied provide many standard routines (such as formatted output, data conversions, and floating-point arithmetic) that may be incorporated into your application.
- Existing routine can be reused in new programs by utilizing the modular programming techniques available with C.
- The C language is very portable and very popular. C compilers are available for almost all target systems. Existing software investments can be quickly and easily converted from or adapted to other processors or environments.

3.1 C51 Language Extensions

The C51 compiler is an ANSI compliant C compiler and includes all aspects of the C programming language that are specified by the ANSI standard. A number of extensions to the C programming language are provided to support the facilities of the 8051 microprocessor. The C51 compiler includes extensions for:

- Data Types,
- Memory Types,
- Memory Models,
- Pointers,
- Interrupt Functions,
- Interfacing to A51 source files.

The following sections briefly describe these extensions.

3.2 Data Types

The C51 compiler supports the data types listed in the following table. In addition to these scalar types, variables can be combined into structures, unions, and arrays. Except as noted, you may use pointers to access these data types.

Data Type	Bits	Bytes	Value Range
bit †	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	±1.175494E-38 to ±3.402823E+38
sbit †	1		0 to 1
sfr †	8	1	0 to 255
sfr16 †	16	2	0 to 65535

† The **bit**, **sbit**, **sfr**, and **sfr16** data types are specific to the 8051 hardware and the C51 compiler. They are not a part of ANSI C and cannot be accessed through pointers.

The **sbit**, **sfr**, and **sfr16** data types are included to allow access to the special function registers that are available on the 8051. For example, the declaration: `sfr P0 = 0x80;` declares the variable `P0` and assigns it the special function register address of `0x80`. This is the address of PORT 0 on the 8051.
The C51 compiler automatically converts between data types when the result implies a different data type. For example, a bit variable used in an integer assignment is converted to an integer. You can, of course, coerce a conversion by using a type cast. In addition to data type conversions, sign extensions are automatically carried out for signed variables.

3.3 Memory Types

The C51 compiler supports the architecture of the 8051 and its derivatives and provides access to all memory areas of the 8051. Each variable may be explicitly assigned to specific memory space.

Memory Type	Description
code	Program memory (64 Kbytes); accessed by opcode MOVC @A+DPTR.
data	Directly addressable internal data memory; fastest access to variables (128 bytes).
idata	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
bdata	Bit-addressable internal data memory; allows mixed bit and byte access (16 bytes).
xdata	External data memory (64 Kbytes); accessed by opcode MOVX @DPTR.
pdata	Paged (256 bytes) external data memory; accessed by opcode MOVX @Rn.

Accessing the internal data memory is considerably faster than accessing the external data memory. For this reason, you should place frequently used variables in internal data memory and less frequently used variables in external data memory.
By including a memory type specifier in the variable declaration, you can specify where variables are stored.
As with the **signed** and **unsigned** attributes, you may include memory type specifiers in the variable declaration. For example:

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

If the memory type specifier is omitted in a variable declaration, the default or implicit memory type is automatically selected. Function arguments and automatic variables, which cannot be located in registers, are also stored in the default memory area.
The default memory type is determined by the **SMALL**, **COMPACT** and **LARGE** compiler control directives. These directives specify the memory model to use for the compilation.

3.4 Memory Models

The memory model determines the default memory type used for function arguments, automatic variables, and variables declared with no explicit memory type. You specify the memory model on the command line using the **SMALL**, **COMPACT**, and **LARGE** control directives. By explicitly declaring a variable with a memory type specifier, you may override the default memory type.

SMALL In this model, all variables default to the internal data memory of the 8051. This is the same as if they were declared explicitly using the **data** memory type specifier. In this memory model, variable access is very efficient. However, all data objects, as well as the stack must fit into the internal RAM. Stack size is critical because the stack space used depends upon the nesting depth of the various functions. Typically, if the BL51 linker/locator is configured to overlay variables in the internal data memory, the small model is the best model to use.

COMPACT Using compact model, all variables default to one page of external data memory. This is the same as if they were explicitly declared using the **pdata** memory type specifier. This memory model can accommodate a maximum of 256 bytes of variables. The limitation is due to the addressing scheme used, which is indirect through registers R0 and R1. This memory model is not as efficient as the small model, therefore, variable access is not as fast. However, the compact model is faster than the large model. The high byte of the address is usually set up via port 2. The compiler does not set this port for you.

LARGE In large model, all variables default to external data memory. This is the same as if they were explicitly declared using the **xdata** memory type specifier. The data pointer (**DPTR**) is used for addressing. Memory access through this data pointer is inefficient, especially for variables with a length of two or more bytes. This type of data access generates more code than the small or compact models.

NOTE
*You should always use the **SMALL** memory model. It generates the fastest, tightest, and most efficient code. You can always explicitly specify the memory area for variables. Move up in model size only if you are unable to make your application fit or operate using **SMALL** model.*

3.5 Pointers

The C51 compiler supports pointer declarations using the asterisk character ('*'). You may use pointers to perform all operations available in standard C. However, because of the unique architecture of the 8051 and its derivatives, the C51 compiler supports two different types of pointers: memory specific pointers and generic pointers.

3.5.1 Generic Pointers

Generic pointers are declared in the same way as standard C pointers. For example:

```
char *s;     /* string ptr */
int *numptr; /* int ptr    */
long *state; /* long ptr   */
```

Generic pointers are always stored using three bytes. The first byte is for the memory type, the second is for the high-order byte of the offset, and the third is for the low-order byte of the offset.
Generic pointers may be used to access any variable regardless of its location in 8051 memory space. Many of the library routines use these pointer types for this reason. By using these generic untyped pointers, a function can access data regardless of the memory in which it is stored.

3.5.2 Memory Specific Pointers

Memory specific pointers always include a memory type specification in the pointer declaration and always refer to a specific memory area. For example:

```
char data *str;    /* ptr to string in data  */
int xdata *numtab; /* ptr to int(s) in xdata */
long code *powtab; /* ptr to long(s) in code */
```

Because the memory type is specified at compile-time, the memory type byte required by untyped pointers is not needed by typed pointers. Typed pointers can be stored using only one byte (**idata**, **data**, **bdata**, and **pdata** pointers) or two bytes (**code** and **xdata** pointers).

3.5.3 Comparison: Memory Specific & Generic Pointers

You can significantly accelerate an 8051 C program by using 'memory specific' pointers. The following sample program shows the differences in code & data size and execution time for various pointer declarations.

Description	Idata Pointer	Xdata Pointer	Generic Pointer

Sample Program	`char idata *ip;`
	`char val;`
	`val = *ip;`
		`char xdata *xp;`
		`char val;`
		`val = *xp;`
			`char *p;`
			`char val;`
			`val = *p;`

8051 Program Code Generated	`MOV R0,ip`
	`MOV val,@R0`
		`MOV DPL,xp +1`
		`MOV DPH,xp`
		`MOV A,@DPTR`
		`MOV val,A`
			`MOV R1,p + 2`
			`MOV R2,p + 1`
			`MOV R3,p`
			`CALL CLDPTR`

Pointer Size	`1 byte data`	`2 bytes data`	`3 bytes data`
Code Size	`4 bytes code`	`9 bytes code`	`11 bytes code + Library`
Execution Time	`4 cycles`	`7 cycles`	`13 cycles`

3.6 Interrupt Functions

The C51 compiler provides you with a method of calling a C function when an interrupt occurs. This support allows you to create interrupt service routines in C. You need only be concerned with the interrupt number and register bank selection. The compiler automatically generates the interrupt vector and entry and exit code for the interrupt routine. The **interrupt** function attribute, when included in a declaration, specifies that the associated function is an interrupt function. Additionally, you can specify the register bank used for that interrupt with the **using** function attribute.

```
unsigned int interruptcnt;
unsigned char second;
void timer0 (void) interrupt 1 using 2 {
  if (++interruptcnt == 4000) {                  /* count to 4000     */
    second++;                                    /* second counter    */
    interruptcnt = 0;                            /* clear int counter */
  }
}
```

3.7 Parameter Passing

The C51 compiler passes up to three function arguments in CPU registers. This significantly improves system performance since arguments do not have to be written to and read from

memory. Argument passing can be controlled with the **REGPARMS** and **NOREGPARMS** control directives.
The following table lists the registers used for different arguments and data types.

Argument Number	char, 1-byte pointer	int, 2-byte pointer	long, float	Generic pointer
1	R7	R6 & R7	R4 - R7	R1 - R3
2	R5	R4 & R5		
3	R3	R2 & R3		

If no registers are available for argument passing or too many arguments are involved, fixed memory locations are used for those extra arguments.

3.8 Function Return Values

CPU registers are always used for function return values. The following table lists the return types and the registers used for each.

Return Type	Register	Description
bit	Carry Flag	
char, **unsigned char, 1-byte pointer**	R7	
int, **unsigned int, 2-byte pointer**	R6 & R7	MSB in R6, LSB in R7
long, **unsigned long**	R4 - R7	MSB in R4, LSB in R7
generic pointer	R1 - R3	Memory type in R3, MSB R2, LSB R1

3.9 Register Optimizing

Depending on program context, the C51 compiler allocates up to 7 CPU registers for register variables. Any registers modified during function execution are noted by the C51 compiler within each module. The linker/locator generates a global, register file which contains information of all registers altered by external functions. Consequently, the C51 compiler *knows* the register used by each function in an application and can optimize the CPU register allocation of each C function.

3.10 Interfacing to Assembly

You can easily access assembly routines from C and vice versa. Function parameters are passed via CPU registers or, if the **NOREGPARMS** control is used, via fixed memory locations. Values returned from functions are always passed in CPU registers.
You can use the **SRC** directive to direct the C51 compiler to generate a file ready to assemble with the A51 assembler instead of an object file. For example, the following C source file:

```
unsigned int asmfunc1 (unsigned int arg) {
  return (1 + arg);
}
```

generates the following assembly output file when compiled using the **SRC** directive.

```
?PR?_asmfunc1?ASM1      SEGMENT CODE
PUBLIC                  _asmfunc1
                RSEG  ?PR?_asmfunc1?ASM1
                USING 0
_asmfunc1:
;---- Variable 'arg?00' assigned to Register 'R6/R7' ----
                MOV   A,R7      ; load LSB of the int
                ADD   A,#01H    ; add 1
                MOV   R7,A      ; put it back into R7
                CLR   A
                ADDC  A,R6      ; add carry & R6
                MOV   R6,A
?C0001:
                RET             ; return result in R6/R7
```

NOTE:
The label of the entry point of the assembly function is equal to the C function name *if the function has no arguments*. If the function uses arguments, the label of the entry point of the assembly function is equal to the C function name preceded by an underscore.

C function name	Label of assembly function
`int calculate( a, b )`	`_calculate`
`void submit()`	`Submit`

3.11 Code Optimizations

The C51 compiler is an aggressive optimizing compiler. This means that the compiler takes certain steps to ensure that the code generated and output to the object file is the most efficient (smaller and/or faster) code possible. The compiler analyzes the generated code to produce the most efficient instruction equences. This ensures that your C program runs as quickly and effectively as possible in the least amount of code space.
The C51 compiler provides six different levels of optimizing. Each increasing level includes the optimizations of levels below it. The following is a list of all optimizations currently performed by the C51 compiler.

3.11.1 General Optimizations

- **Constant Folding**: Several constant values occurring in an expression or address calculation are combined as a single constant.
- **Jump Optimizing**: Jumps are inverted or extended to the final target address when the program efficiency is thereby increased.
- **Dead Code Elimination**: Code which cannot be reached (dead code) is removed from the program.
- **Register Variables**: Automatic variables and function arguments are located in registers whenever possible. No data memory space is reserved for these variables.
- **Parameter Passing Via Registers**: A maximum of three function arguments can be passed in registers.
- **Global Common Subexpression Elimination**: Identical subexpressions or address calculations that occur multiple times in a function are recognized and calculated only once whenever possible.

3.11.2 8051-Specific Optimizations

- **Peephole Optimization**: Complex operations are replaced by simplified operations when memory space or execution time can be saved as a result.

- **Access Optimizing**: Constants and variables are computed and included directly in operations.
- **Data Overlaying**: Data and bit segments of functions are identified as OVERLAYABLE and are overlaid with other data and bit segments by the BL51 linker/locator.
- **Case/Switch Optimizing**: Depending upon their number, sequence, and location, **switch** and **case** statements can be further optimized by using a jump table or string of jumps.

3.11.3 Options for Code Generation

- **OPTIMIZE(SIZE)**: Common C operations are replaced by subprograms. Program code size is reduced at the expense of program speed.
- **OPTIMIZE(SPEED)**: Common C operations are expanded in-line. Program speed is increased at the expense of code size.
- **NOAREGS**: The C51 compiler no longer uses absolute register access. Program code is independent of the register bank.
- **NOREGPARMS**: Parameter passing is always performed in local data segments rather then dedicated registers

3.11.4 Global Register Optimization

The C51 compiler provides support for application wide register optimization which is also known as application register coloring. The following sample program compares the code generated by C51 version 5.0 using application register coloring to the code generated by C51 version 3.4 without application register coloring. With the application wide register optimization, the C compiler *knows* the registers used by external functions. Registers that are not altered in external functions are used for register variables. The generated code needs less data and code space and executes faster. In the following example *input* and *output* are external functions, which require only a few registers.

With Global Register Optimization	Without Global Register Optimization

`main () {`
`  unsigned char i;`
`  unsigned char a;`
`  while (1) {`
`  i = input ();`
	`/* get number of values */`

`?C0001:`
`  LCALL input`
`;- 'I' assigned to 'R6' -`
`  MOV R6,AR7`
	`?C0001:`
	`  LCALL input`
	`  MOV DPTR,#I`
	`  MOV A,R7`
	`  MOV @DPTR,A`

`  do {` `  a = input ();`	`/* get input value */`

`?C0005:`
`  LCALL input`
`;- 'a' assigned to 'R7' -`
`  MOV R5,AR7`
	`?C0005:`
	`  LCALL input`
	`  MOV DPTR,#a`
	`  MOV A,R7`
	`  MOVX @DPTR,A`

`  output (a);`	`/* output value */`
`  LCALL _output`	`  LCALL _output`
`  } while (--i);`	`/* decrement values */`

`  DJNZ R6,?C0005`	`  MOV DPTR,#I`
	`  MOVX A,@DPTR`
	`  DEC A`
	`  MOVX @DPTR,A`
	`  JNZ ?C0005`

`  }`	
`  SJMP ?C0001`	`  SJMP ?C0001`
`  }`	
`  RET`	`  RET`
Code Size: 18 Bytes	**Code Size: 30 Bytes**

3.12 Library Routines

The C51 compiler includes four different ANSI compile-time libraries which are optimized for various functional requirements.

Library File Description
C51S.LIB Small model library without floating-point arithmetic
C51C.LIB Compact model library without floating-point arithmetic
C51L.LIB Large model library without floating-point arithmetic
80C751.LIB Library for use with the Philips 8xC751 and derivatives.

3.13 Intrinsic Library Routines

The libraries included with the compiler include a number of routines that are implemented as intrinsic functions. Non-intrinsic functions generate **ACALL** or **LCALL** instructions to perform the library routine. Intrinsic functions generate in-line code (which is faster and more efficient) to perform the library routine.

Intrinsic	Function Description
crol	Rotate character left.
cror	Rotate character right.
irol	Rotate integer left.
iror	Rotate integer right.
lrol	Rotate long integer left.
lror	Rotate long integer right.
nop	No operation (8051 NOP instruction).
testbit	Test and clear bit (8051 JBC instruction).

3.14 Listing File Example

The C51 compiler produces a listing file that contains source code, directive information, an assembly listing, and a symbol table.

```
C51 COMPILER V5.02, SAMPLE            07/01/95 08:00:00 PAGE 1
DOS C51 COMPILER V5.02, COMPILATION OF MODULE SAMPLE
OBJECT MODULE PLACED IN SAMPLE.OBJ
COMPILER INVOKED BY: C:\C51\BIN\C51.EXE SAMPLE.C CODE

Stmt level source
   1       #include <reg51.h>   /* SFR definitions for 8051 */
   2       #include <stdio.h>   /* standard i/o definitions */
   3       #include <ctype.h>   /* defs for char conversion */
   4
   5       #define EOT 0x1A     /* Control+Z signals EOT    */
   6
   7       void main (void) {
   8 1       unsigned char c;
   9 1
  10 1       /* setup serial port hdw (2400 Baud @12 MHz) */
  11 1       SCON = 0x52;       /* SCON */
  12 1       TMOD = 0x20;       /* TMOD */
  13 1       TCON = 0x69;       /* TCON */
  14 1       TH1 = 0xF3;        /* TH1  */
  15 1
  16 1       while ((c = getchar ()) != EOF) {
  17 2         putchar (toupper (c));
  18 2       }
  19 1       P0 = 0;   /* clear Output Port to signal ready */
  20 1     }

ASSEMBLY LISTING OF GENERATED OBJECT CODE

            ; FUNCTION main (BEGIN)
                                             ; SOURCE LINE # 7
                                             ; SOURCE LINE # 11
0000 759852       MOV     SCON,#052H
                                             ; SOURCE LINE # 12
0003 758920       MOV     TMOD,#020H
                                             ; SOURCE LINE # 13
0006 758869       MOV     TCON,#069H
                                             ; SOURCE LINE # 14
0009 758DF3       MOV     TH1,#0F3H
000C        ?C0001:
                                             ; SOURCE LINE # 16
000C 120000 E     LCALL   getchar
000F 8F00   R     MOV     c,R7
0011 EF           MOV     A,R7
0012 F4           CPL     A
0013 6008         JZ      ?C0002
                                             ; SOURCE LINE # 17
0015 120000 E     LCALL _ toupper
0018 120000 E     LCALL _ putchar
                                             ; SOURCE LINE # 18
001B 80EF         SJMP    ?C0001
001D        ?C0002:
                                             ; SOURCE LINE # 19
001D E4           CLR     A
001E F580         MOV     P0,A
                                             ; SOURCE LINE # 20
0020 22           RET
            ; FUNCTION main (END)


MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     33    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----       1
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.


C51 COMPILATION COMPLETE. 0 WARNING(S), 0 ERROR(S)
```

The C51 compiler produces a listing file with page numbers as well as time and date of the compilation. Remarks about the compiler invocation and object file output are displayed in this listing.

The listing includes a line number for each statement and a nesting level for each block enclosed within curly braces ('{' and '}').

Error messages and warning messages are included in the listing file.

The **CODE** compiler option includes an assembly code listing in the listing file. Source line numbers are embedded within the generated code.

A memory overview provides information about the 8051 memory areas that are used.

The total number of errors and warnings is stated at the end of the listing file.

4 BL51 Linker/Locator

The BL51 linker/locator combines one or more object modules into a single executable 8051 program. The linker also resolves external and public references, and assigns absolute addresses to relocatable programs segments.
The BL51 linker/locator processes object modules created by the C51 compiler and the A51 assembler. The linker automatically selects the appropriate run-time library and links only the library modules that are required.
Normally, you invoke the BL51 linker/locator from the command line specifying the names of the object modules to combine. The default controls for the BL51 linker/locator have been carefully chosen to accommodate most applications without the need to specify additional directives. However, it is easy for you to specify custom settings for your application.

4.1 Data Address Management

The BL51 linker/locator manages the limited internal memory of the 8051 by overlaying variables for functions that are mutually exclusive. This greatly reduces the overall memory requirement of most 8051 applications. The BL51 linker/locator analyzes the references between functions to carry out memory overlaying. You may use the **OVERLAY** directive to manually control functions references the linker uses to determine exclusive memory areas. The **NOOVERLAY** directive lets you completely disable memory overlaying. These directives are useful when using indirectly called functions or when disabling overlaying for debugging.

4.2 Listing File Example

The following example shows a map file created by the BL51 linker/locator:

```
BL51 BANKED LINKER/LOCATER V3.52     07/01/95  08:00:00  PAGE 1

MS-DOS BL51 BANKED LINKER/LOCATER V3.52, INVOKED BY:
C:\C51\BIN\BL51.EXE SAMPLE.OBJ

MEMORY MODEL: SMALL

INPUT MODULES INCLUDED:
  SAMPLE.OBJ (SAMPLE)
  C:\C51\LIB\C51S.LIB (?C_STARTUP)
  C:\C51\LIB\C51S.LIB (PUTCHAR)
  C:\C51\LIB\C51S.LIB (GETCHAR)
  C:\C51\LIB\C51S.LIB (TOUPPER)
  C:\C51\LIB\C51S.LIB ( GETKEY)


LINK MAP OF MODULE: SAMPLE (SAMPLE)

     TYPE    BASE      LENGTH    RELOCATION SEGMENT NAME
     -----------------------------------------------------

     * * * * * * * D A T A M E M O R Y * * * * * * *
     REG     0000H     0008H     ABSOLUTE   "REG BANK 0"
     DATA    0008H     0001H     UNIT       ?DT?GETCHAR
     DATA    0009H     0001H     UNIT       _DATA_GROUP_
             000AH     0016H               *** GAP ***
     BIT     0020H.0   0000H.1   UNIT       ?BI?GETCHAR
             0020H.1   0000H.7               *** GAP ***
     IDATA   0021H     0001H     UNIT       ?STACK

     * * * * * * * C O D E M E M O R Y * * * * * * *
     CODE    0000H     0003H     ABSOLUTE
     CODE    0003H     0021H     UNIT       ?PR?MAIN?SAMPLE
     CODE    0024H     000CH     UNIT       ?C_C51STARTUP
     CODE    0030H     0027H     UNIT       ?PR?PUTCHAR?PUTCHAR
     CODE    0057H     0011H     UNIT       ?PR?GETCHAR?GETCHAR
     CODE    0068H     0018H     UNIT       ?PR?_TOUPPER?TOUPPER
     CODE    0080H     000AH     UNIT       ?PR?_GETKEY?_GETKEY


OVERLAY MAP OF MODULE:  SAMPLE (SAMPLE)

SEGMENT                          DATA_GROUP
  +--> CALLED SEGMENT          START    LENGTH
------------------------------------------------
?C_C51STARTUP                  -----    -----
  +--> ?PR?MAIN?SAMPLE

?PR?MAIN?SAMPLE                0009H    0001H
  +--> ?PR?GETCHAR?GETCHAR
  +--> ?PR? TOUPPER?TOUPPER
  +--> ?PR?PUTCHAR?PUTCHAR

?PR?GETCHAR?GETCHAR            -----    -----
  +--> ?PR?_GETKEY?_GETKEY
  +--> ?PR?PUTCHAR?PUTCHAR


LINK/LOCATE RUN COMPLETE. 0 WARNING(S), 0 ERROR(S)
```

The BL51 linker/locator produces a map file with the time and date of the link/locate run.

The invocation line and the selected memory module are listed.

The link-map contains a table of the memory usage of the physical 8051 memory area.

The overlay-map displays the structure of the program and the location of the bit and data segments of each function.

Error messages and warnings are listed at the end of the map file. These messages indicate possible problems during the link/locate run.

Appendix

C

Appendix C: The Altera UP1 Board

University Program Design Laboratory Package

October 2001, ver. 2.0 User Guide

Introduction

The University Program (UP) Design Laboratory Package was designed to meet the needs of universities teaching digital logic design with state-of-the-art development tools and programmable logic devices (PLDs). The package provides all of the necessary tools for creating and implementing digital logic designs, including the following features:

- MAX+PLUS® II University development software
- UP Education Board
  - EPF10K20 device for the UP1 board or an EPF10K70 device for the UP2 board in a 240-pin power quad flat pack (RQFP) package
  - EPM7128S device for the UP1 and UP2 boards in an 84-pin plastic J-lead chip carrier (PLCC) package
- ByteBlasterMV™ parallel port download cable

MAX+PLUS II University Software

The MAX+PLUS II University software contains many of the features of the commercial version of the MAX+PLUS II software including a completely integrated design flow and an intuitive graphical user interface. This software supports schematic capture and text-based hardware description language (HDL) design entry, including Verilog HDL, VHDL, and the Altera® Hardware Description Language (AHDL™). It also provides design programming, compilation, and verification support for all devices supported by the MAX+PLUS II BASELINE software including the EPM7128S, EPF10K20, and EPF10K70 devices. The MAX+PLUS II University software can be freely distributed to students for installation on their personal computers and provides instant access to online help.

For information on how to install the MAX+PLUS II software on your computer, see "Software Installation" on page 18.

UP1 & UP2 Education Boards

The UP1 and UP2 Education Boards are stand-alone experiment boards based on a FLEX® 10K device and include a MAX® 7000 device. When used with the MAX+PLUS II University software, the boards provide a superior platform for learning digital logic design using industry-standard development tools and PLDs.

The boards are designed to meet the needs of instructors and students in a laboratory environment. The UP1 and UP2 Education Boards support both look-up table (LUT)-based and product term-based architectures. The EPF10K70 and EPF10K20 devices can be configured in-system with either the ByteBlasterMV download cable or an EPC1 configuration device. Additional download cables can be purchased separately. The EPM7128S device can be programmed in-system with the ByteBlasterMV download cable.

EPF10K70 Device

The EPF10K70 device is based on SRAM technology. It is available in a 240-pin RQFP package and has 3,744 logic elements (LEs) and nine embedded array blocks (EABs). Each LE consists of a four-input LUT, a programmable flipflop, and dedicated signal paths for carry-and-cascade functions. Each EAB provides 2,048 bits of memory which can be used to create RAM, ROM, or first-in first-out (FIFO) functions. EABs can also implement logic functions, such as multipliers, microcontrollers, state machines, and digital signal processing (DSP) functions. With 70,000 typical gates, the EPF10K70 device is ideal for intermediate to advanced digital design courses, including computer architecture, communications, and DSP applications.

EPF10K20 Device

The EPF10K20 device is based on reconfigurable SRAM elements. The EPF10K20 device is available in a 240-pin RQFP package and has 1,152 LEs and six EABs. Each LE consists of a four-input LUT, a programmable flipflop, and dedicated signal paths for carry-and-cascade functions. Each EAB provides 2,048 bits of memory which can be used to create RAM, ROM, or FIFO functions. The EABs can implement logic functions, such as multipliers, microcontrollers, state machines, and DSP functions. With 20,000 typical gates, the EPF10K20 device is ideal for introductory digital design courses.

For more information on FLEX 10K devices, see the *FLEX 10K Embedded Programmable Logic Family Data Sheet.*

EPM7128S Device

The EPM7128S device, a member of the high-density, high-performance MAX 7000S family, is based on erasable programmable read-only memory (EEPROM) elements. The EPM7128S device features a socket-mounted 84-pin plastic j-lead chip carrier (PLCC) package and has 128 macrocells. Each macrocell has a programmable-AND/fixed-OR array as well as a configurable register with independently-programmable clock, clock enable, clear, and preset functions. With a capacity of 2,500 gates and a simple architecture, the EPM7128S device is ideal for introductory designs as well as larger combinatorial and sequential logic functions.

For more information on MAX 7000 devices, go to the *MAX 7000 Programmable Logic Device Family Data Sheet.*

ByteBlasterMV Parallel Port Download Cable

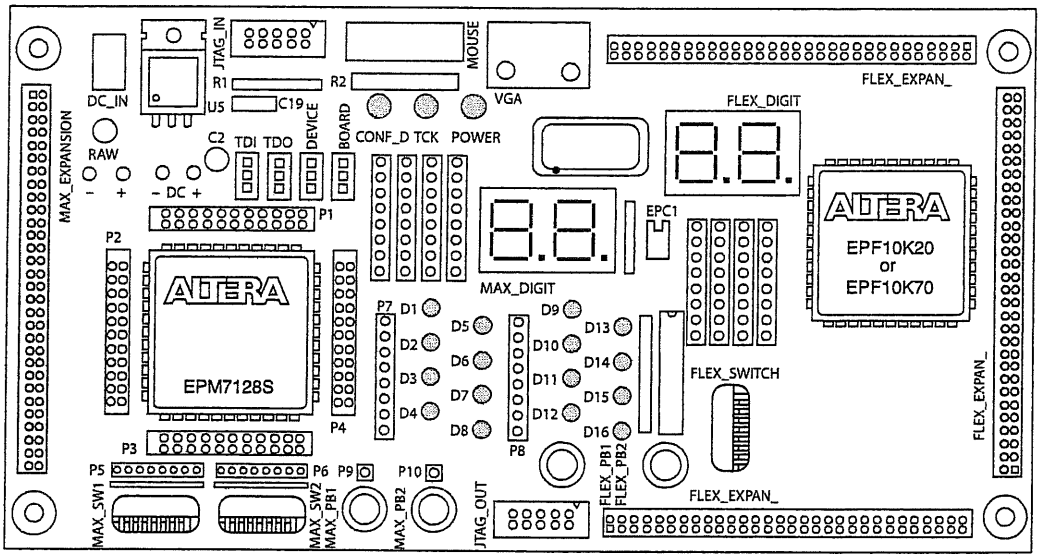
Designs can be easily and quickly downloaded into the UP1 and UP2 Education Boards using the ByteBlasterMV download cable, which is a hardware interface to a standard parallel port. This cable sends programming or configuration data between the MAX+PLUS II University software and the UP Education Boards. Because design changes are downloaded directly to the devices on the board, prototyping is easy and multiple design iterations can be accomplished in quick succession.

For more information on the ByteBlasterMV download cable, see the *ByteBlasterMV Parallel Port Download Cable Data Sheet.*

UP Education Board Description

The UP1 and UP2 Education Boards contain the features described in this section. Figure 1 shows a block diagram of the UP Education Board.

Figure 1. UP Education Board Block Diagram



DC_IN & RAW Power Input

The DC_IN power input accepts a 2.5-mm × 5.55-mm female connector. The acceptable DC input is 7 to 9 V at a minimum of 350 mA. The RAW power input consists of two holes for connecting an unregulated power source. The hole marked with a plus sign (+) is the positive input; the hole marked with a minus sign (–) is board-common.

On-Board Voltage Regulator

The on-board voltage regulator, an LM340T, regulates the DC positive input at 5 V. The DC input consists of two holes for connecting a 5-V DC regulated power source. The hole marked with a plus sign (+) is the positive input; the hole marked with a minus sign (–) is board common. A green light-emitting diode (LED) labeled POWER is illuminated when current is flowing from the 5-V DC-regulated power source.

Oscillator

The UP Education Boards contain a 25.175-MHz crystal oscillator. The output of the oscillator drives a global clock input on the EPM7128S device (pin 83) and a global clock input on the FLEX 10K device (pin 91).

JTAG_IN Header

The 10-pin female plug on the ByteBlasterMV download cable connects with the JTAG_IN 10-pin male header on the UP Education Boards. The boards provide power and ground to the ByteBlasterMV download cable. Data is shifted into the devices via the TDI pin and shifted out of the devices via the TDO pin. Table 1 identifies the JTAG_IN pin names when the ByteBlasterMV is operating in Joint Test Action Group (JTAG) mode.

Table 1. JTAG_IN 10-Pin Header Pin-Outs

Pin	JTAG Signal
1	TCK
2	GND
3	TDO
4	VCC
5	TMS
6	No Connect
7	No Connect
8	No Connect
9	TDI
10	GND

Jumpers

The UP Education Boards have four three-pin jumpers (TDI, TDO, DEVICE, and BOARD) that set the JTAG configuration. The JTAG chain can be set for a variety of configurations (i.e., to program only the EPM7128S device, to configure only the FLEX 10K device, to configure and program both devices, or to connect multiple UP Education Boards together). Figure 2 shows the positions of the three connectors (C1, C2, and C3) on each of the four jumpers.

Figure 2. Position of C1, C2 & C3 Connectors

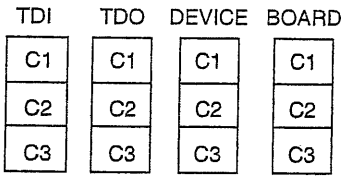


Table 2 defines the settings for each configuration.

Table 2. JTAG Jumper Settings

Desired Action	TDI	TDO	DEVICE	BOARD
Program EPM7128S device only	C1 & C2	C1 & C2	C1 & C2	C1 & C2
Configure FLEX 10K device only	C2 & C3	C2 & C3	C1 & C2	C1 & C2
Program/configure both devices *(1)*	C2 & C3	C1 & C2	C2 & C3	C1 & C2
Connect multiple boards together *(2)*	C2 & C3	OPEN	C2 & C3	C2 & C3

Notes to Table 2:

(1) The first device in the JTAG chain is the FLEX 10K device, and the second device is the EPM7128S device.

(2) The first device in the JTAG chain is the FLEX 10K device, and the second device is the EPM7128S device. The last board in the chain must be set for a single board configuration (i.e., for programming only the EPM7128S device, configuring only the FLEX 10K device, or configuring/programming both devices). The last board cannot be set for connecting multiple boards together.

During configuration, the green CONF_D LED will turn off and the green TCK LED will modulate to indicate that data is transferring. After the device has successfully configured, the CONF_D LED will illuminate.

For information on how to program or configure EPF10K20, EPF10K70, or EPM7128S devices, see "Programming or Configuring Devices" on page 19.

FLEX 10K Device

The UP1 and UP2 Education Boards provide the following resources for the FLEX 10K device. The pins from the FLEX 10K device are pre-assigned to switches and LEDs on the board.

- JTAG chain connection for the ByteBlasterMV cable
- Socket for an EPC1 configuration device
- Two momentary push button switches
- One octal DIP switch
- Dual-digit seven-segment display
- On-board oscillator (25.175 MHz)
- VGA port
- Mouse port
- Three expansion ports, each with 42 I/O pins and seven global pins

FLEX_PB1 & FLEX_PB2 Push Buttons

FLEX_PB1 and FLEX_PB2 are two push buttons that provide active-low signals to two general-purpose I/O pins on the FLEX 10K device. FLEX_PB1 connects to pin 28, and FLEX_PB2 connects to pin 29. Each push button is pulled-up through a 10-KΩ resistor.

FLEX_SW1 Switches

FLEX_SW1 contains eight switches that provide logic-level signals to eight general-purpose I/O pins on the FLEX 10K device. An input pin is set to logic 1 when the switch is open and set to logic 0 when the switch is closed. Table 6 lists the pin assignment for each switch.

Table 6. FLEX_SW1 Pin Assignments

Switch	FLEX 10K Pin
FLEX_SWITCH-1	41
FLEX_SWITCH-2	40
FLEX_SWITCH-3	39
FLEX_SWITCH-4	38
FLEX_SWITCH-5	36
FLEX_SWITCH-6	35
FLEX_SWITCH-7	34
FLEX_SWITCH-8	33

FLEX_DIGIT Display

FLEX_DIGIT is a dual-digit, seven-segment display connected directly to the FLEX 10K device. Each LED segment on the display can be illuminated by driving the connected FLEX 10K device I/O pin with a logic 0. See Figure 4 on page 9 for the name of each segment. Table 7 lists the pin assignment for each segment.

Table 7. FLEX_DIGIT Segment I/O Connections

Display Segment	Pin for Digit 1	Pin for Digit 2
a	6	17
b	7	18
c	8	19
d	9	20
e	11	21
f	12	23
g	13	24
Decimal point	14	25

VGA Interface

The VGA interface allows the FLEX 10K device to control an external video monitor. This interface is composed of a simple diode-resistor network and a 15-pin D-sub connector (labeled VGA), where the monitor can plug into the boards. The diode-resistor network and D-sub connector are designed to generate voltages that conform to the VGA standard.

Information about the color, row, and column indexing of the screen is sent from the FLEX 10K device to the monitor via five signals. Three VGA signals are red, green, and blue, while the other two signals are horizontal and vertical synchronization. Manipulating these signals allows images to be written to the monitor's screen.

☞ See "VGA Driver Operation" on page 26 for details on how the VGA interface operates.

Table 8 lists the D-sub connector and the FLEX 10K device connections.

6. Click **Detect JTAG Chain Info** to have the ByteBlasterMV cable check the device count, JTAG ID code, and total instruction length of the JTAG chain. A message just above the **Detect JTAG Chain Info** button reports the information detected by the ByteBlasterMV cable. This message must be manually verified to match the information in the **Device Names & Programming File Names** box.

7. Click **Save JCF**. In the **Save JCF** dialog box, type the name of the file in the **File Name** box and then select the desired directory in the **Directories** box to save the current settings to a JTAG Chain File (.jcf) for future use. Click **OK**.

8. Click **OK** to save changes.

9. Click **Program** in the MAX+PLUS II Programmer.

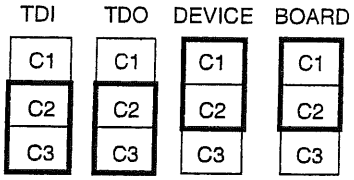
EPF10K70 & EPF10K20 Configuration

This section describes the procedures for configuring only the EPF10K70 or EPF10K20 devices (i.e., how to set the on-board jumpers, connect the ByteBlasterMV download cable, and set options in the MAX+PLUS II software).

Setting the On-Board Jumpers for EPF10K70 or EPF10K20 Configuration

To configure only the EPF10K70 or EPF10K20 device in a JTAG chain, set the jumpers TDI, TDO, DEVICE, and BOARD as shown in Figure 8.

Figure 8. Jumper Settings for Configuring Only the FLEX 10K Device



Connecting the ByteBlasterMV Download Cable for EPF10K70 or EPF10K20 Configuration

Attach the ByteBlasterMV cable directly to the PC's parallel port and to the JTAG_IN connector on the UP Education Board. For more information on setting up the ByteBlasterMV cable, see the *ByteBlasterMV Parallel Port Download Cable Data Sheet.*

Setting the JTAG Options in the MAX+PLUS II Software for EPF10K70 or EPF10K20 Configuration

The following steps describe how to use the MAX+PLUS II software to configure the EPF10K70 or EPF10K20 device in a JTAG chain. For more information on how to configure a device, see the MAX+PLUS II Help.

1. Turn on the **Multi-Device JTAG Chain** command (JTAG menu) in the MAX+PLUS II Programmer to configure the EPF10K70 or EPF10K20 devices. Follow this step even if you are only programming one device.

2. Choose **Multi-Device JTAG Chain Setup** (JTAG menu).

3. Select **EPF10K70 or EPF10K20** in the **Device Name** list in the **Multi-Device JTAG Chain Setup** dialog box.

4. Type the name of the programming file for the EPF10K70 or EPF10K20 device in the **Programming File Name** box. You can also use the **Select Programming File** button to browse your computer's directory structure to locate the appropriate programming file.

5. Click **Add** to add the device and associated programming file to the **Device Names & Programming File Names** box. The number to the left of the device name shows the order of the device in the JTAG chain. The device's associated programming file is displayed on the same line as the device name. If no programming file is associated with a device, "<none>" is displayed next to the device name.

6. Click **Detect JTAG Chain Info** to have the ByteBlasterMV cable check the device count, JTAG ID code, and total instruction length of the JTAG chain. A message just above the **Detect JTAG Chain Info** button reports the information detected by the ByteBlasterMV cable. You must manually verify that this message matches the information in the **Device Names & Programming File Names** box.

7. Click **Save JCF** to save the current settings to a JCF for future use. Type the name of the file in the **File Name** box and then select the desired directory in the **Directories** box in the **Save JCF** dialog box. Click **OK**.

8. Click **OK** to save your changes.

9. Click **Configure** in the MAX+PLUS II Programmer.

Appendix D: FFT for programmers

FFTs for programmers
algorithms and source code

preliminary draft version

Jörg Arndt
arndt@jjj.de

This document[1] was LaTeX'd at September 25, 2001

[1]This document is online at http://www.jjj.de/fxt/. It will stay available online for free.

Contents

List of important Symbols

$\Re x$	real part of x
$\Im x$	imaginary part of x
x^*	complex conjugate of x
a	a sequence, e.g. $\{a_0, a_1, ..., a_{n-1}\}$, the index always starts with zero.
$\hat{a}$	transformed (e.g. Fourier transformed) sequence
$\stackrel{m}{=}$	emphasize that the sequences to the left and right are all of length m
$\mathcal{F}[a]$ $(= c)$	(discrete) Fourier transform (FT) of a, $c_k = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x z^{x\,k}$ where $z = e^{\pm 2\pi i/n}$
$\mathcal{F}^{-1}[a]$	inverse (discrete) Fourier transform (IFT) of a, $\mathcal{F}^{-1}[a]_k = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x z^{-x\,k}$
$\mathcal{S}^k a$	a sequence c with elements $c_x := a_x e^{\pm k 2\pi i x/n}$
$\mathcal{H}[a]$	discrete Hartley transform (HT) of a
$\overline{a}$	sequence reversed around element with index $n/2$
a_S	the symmetric part of a sequence: $a_S := a + \overline{a}$
a_A	the antisymmetric part of a sequence: $a_A := a - \overline{a}$
$\mathcal{Z}[a]$	discrete z-transform (ZT) of a
$\mathcal{W}_v[a]$	discrete weighted transform of a, weight (sequence) v
$\mathcal{W}_v^{-1}[a]$	inverse discrete weighted transform of a, weight v
$a \circledast b$	cyclic (or circular) convolution of sequence a with sequence b
$a \circledast_{ac} b$	acyclic (or linear) convolution of sequence a with sequence b
$a \circledast_{-} b$	negacyclic (or skew circular) convolution of sequence a with sequence b
$a \circledast_{(v)} b$	weighted convolution of sequence a with sequence b, weight v
$n \backslash N$	n divides N

Chapter 1

The Fourier transform

1.1 The discrete Fourier transform

The *discrete Fourier transform* (DFT or simply FT) of a complex sequence a of length n is defined as

$$c = \mathcal{F}[a] \tag{1.1}$$

$$c_k := \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} a_x z^{\pm x\,k} \quad \text{where} \quad z = e^{\pm 2\pi i/n} \tag{1.2}$$

z is an n-th root of unity: $z^n = 1$.

Backtransform (or *inverse discrete Fourier transform* IDFT or simply IFT) is then

$$a = \mathcal{F}^{-1}[c] \tag{1.3}$$

$$a_x = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} c_k z^{-x\,k} \tag{1.4}$$

That this is really true is not straightforward. Consider element y of the IFT of the FT of a:

$$\mathcal{F}^{-1}[\mathcal{F}[a]]_y = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} (a_x z^{x\,k}) z^{-y\,k} \tag{1.5}$$

$$= \frac{1}{n} \sum_x a_x \sum_k (z^{x-y})^k \tag{1.6}$$

As $\sum_k (z^{x-y})^k = n$ for $x = y$ and zero else (because z is an n-th root of unity). Therefore the whole expression is equal to

$$\frac{1}{n} n \sum_x a_x \delta_{x,y} = a_y \tag{1.7}$$

where

$$\delta_{x,y} = \begin{cases} 1 & (x = y) \\ 0 & (x \neq y) \end{cases} \tag{1.8}$$

In this book the FT with the plus in the exponent is called forward transform, the one with the minus is called the backward transform, the choice is arbitrary[1].

[1] Electrical engineers prefer the minus for the forward transform, mathematicians the plus.

The FT is a linear transform, i.e. for $\alpha, \beta \in \mathbb{C}$

$$\mathcal{F}[\alpha a + \beta b] = \alpha \mathcal{F}[a] + \beta \mathcal{F}[b] \tag{1.9}$$

For the FT Parseval's equation holds, let $c = \mathcal{F}[a]$, then

$$\sum_{x=0}^{n-1} a_x^2 = \sum_{k=0}^{n-1} c_k^2 \tag{1.10}$$

The normalisation factor $\frac{1}{\sqrt{n}}$ in front of the FT sums is sometimes replaced by a single $\frac{1}{n}$ in front of the inverse FT sum which is often convenient in computation. Then, of course, Parseval's equation has to be modified accordingly.

A straightforward implementation of the discrete Fourier transform, i.e. the computation of n sums each of length n requires $\sim n^2$ operations.

Code 1.1 (Fourier transform by definition) *Compute the Fourier transform of the complex sequence* `a[]`*, the result is returned in* `c[]`

```
procedure ft(a[],c[],n,is)
{
    for k:=0 to n-1
    {
        s := 0
        for x:=0 to n-1
        {
            s := s + a[x]*exp(is*2.0*I*PI*x*k/n)
        }
        c[k] := s
    }
}
```

[FXT: `slow_ft` in file `slow/slowft.cc`]

A *fast Fourier transform* (FFT) algorithm is an algorithm that improves the operation count to proportional $n \sum_{k=1}^{m}(p_k - 1)$, where $n = p_1 p_2 \cdots p_m$ is a factorization of n. In case of a power $n = p^m$ the value computes to $n\,(p-1)\log_p(n)$. In the special case $p = 2$ even $n/2 \log_2(n)$ multiplications are enough. There are several different FFT algorithms with many variants.

1.2 Symmetries of the Fourier transform

The FT has several symmetry properties, a bit of notation turns out to be useful becoming written down. Let $\overline{a}$ be the sequence a (length n) reversed around element with index $n/2$:

$$\overline{a}_0 := a_0 \tag{1.11}$$

$$\overline{a}_{n/2} := a_{n/2} \quad \text{if } n \text{ even} \tag{1.12}$$

$$\overline{a}_k := a_{n-k} \tag{1.13}$$

Let a_S, a_A be the symmetric, antisymmetric part of the sequence a, respectively:

$$a_S := a + \overline{a} \tag{1.14}$$

$$a_A := a - \overline{a} \tag{1.15}$$

(The elements with indices 0 and $n/2$ of a_A are zero). Now let $a \in \mathbb{R}$ (meaning that each element of a is $\in \mathbb{R}$), then

$$\mathcal{F}[a_S] \in \mathbb{R} \tag{1.16}$$

$$\mathcal{F}[a_S] = \overline{\mathcal{F}[a_S]} \tag{1.17}$$

$$\mathcal{F}[a_A] \in i\,\mathbb{R} \tag{1.18}$$

$$\mathcal{F}[a_A] = -\overline{\mathcal{F}[a_A]} \tag{1.19}$$

i.e. the FT of a real symmetric sequence is real and symmetric and the FT of a real antisymmetric sequence is purely imaginary and antisymmetric. Thereby the FT of a general real sequence is the complex conjugate of its reversed:

$$\mathcal{F}[a] = \overline{\mathcal{F}[a]}^* \quad for \quad a \in \mathbb{R} \tag{1.20}$$

Similarly, for a purely imaginary sequence $b \in i\mathbb{R}$:

$$\mathcal{F}[b_S] \in i\,\mathbb{R} \tag{1.21}$$

$$\mathcal{F}[b_S] = \overline{\mathcal{F}[b_S]} \tag{1.22}$$

$$\mathcal{F}[b_A] \in \mathbb{R} \tag{1.23}$$

$$\mathcal{F}[b_A] = -\overline{\mathcal{F}[b_A]} \tag{1.24}$$

The FT of a complex symmetric/antisymmetric sequence is symmetric/antisymmetric, respectively.

1.3 Radix 2 FFT algorithms

1.3.1 A little bit of notation

Always assume a is a length-n sequence (n even) in what follows:

Let $a^{(even)}$, $a^{(odd)}$ denote the (length-$n/2$) subsequences of those elements of a that have even or odd indices, respectively.

Let $a^{(left)}$ denote the subsequence of those elements of a that have indices $0...n/2-1$.

Similarly, $a^{(right)}$ for indices $n/2...n-1$.

Let $\mathcal{S}^k a$ denote the sequence with elements $a_x\, e^{\pm k\, 2\pi i x/n}$ where n is the length of the sequence a and the sign is that of the transform. The symbol $\mathcal{S}$ shall suggest a shift operator. In the next two sections only $\mathcal{S}^{1/2}$ will appear. $\mathcal{S}^0$ is the identity operator.

1.3.2 Decimation in time (DIT) FFT

The following observation is the key to the decimation in time (DIT) FFT[2] algorithm:
For n even the k-th element of the Fourier transform is

$$\sum_{x=0}^{n-1} a_x\, z^{x\,k} = \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{(2\,x+1)\,k} \tag{1.25}$$

$$= \sum_{x=0}^{n/2-1} a_{2\,x}\, z^{2\,x\,k} + z^k \sum_{x=0}^{n/2-1} a_{2\,x+1}\, z^{2\,x\,k} \tag{1.26}$$

where $z = e^{\pm i\,2\pi/n}$ and $k \in \{0, 1, ..., n-1\}$.

The last identity tells us how to compute the k-th element of the length-n Fourier transform from the length-$n/2$ Fourier transforms of the even and odd indexed subsequences.

To actually rewrite the length-n FT in terms of length-$n/2$ FTs one has to distinguish the cases $0 \le k < n/2$ and $n/2 \le k < n$, therefore we rewrite $k \in \{0, 1, 2, ..., n-1\}$ as $k = j + \delta\,\frac{n}{2}$ where $j \in \{0, 1, ..., n/2-1\}$, $\delta \in \{0, 1\}$.

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(j+\delta\,\frac{n}{2})} = \sum_{x=0}^{n/2-1} a_x^{(even)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} + z^{j+\delta\,\frac{n}{2}} \sum_{x=0}^{n/2-1} a_x^{(odd)}\, z^{2\,x\,(j+\delta\,\frac{n}{2})} \tag{1.27}$$

[2] also called Cooley-Tukey FFT.

$$= \begin{cases} \sum_{x=0}^{n/2-1} a_x^{(even)} z^{2xj} + z^j \sum_{x=0}^{n/2-1} a_x^{(odd)} z^{2xj} & \text{for } \delta = 0 \\ \sum_{x=0}^{n/2-1} a_x^{(even)} z^{2xj} - z^j \sum_{x=0}^{n/2-1} a_x^{(odd)} z^{2xj} & \text{for } \delta = 1 \end{cases} \quad (1.28)$$

Noting that z^2 is just the root of unity that appears in a length-$n/2$ FT one can rewrite the last two equations as the

Idea 1.1 (FFT radix 2 DIT step) *Radix 2 decimation in time step for the FFT:*

$$\mathcal{F}[a]^{(left)} \stackrel{n/2}{=} \mathcal{F}\left[a^{(even)}\right] + \mathcal{S}^{1/2}\mathcal{F}\left[a^{(odd)}\right] \quad (1.29)$$

$$\mathcal{F}[a]^{(right)} \stackrel{n/2}{=} \mathcal{F}\left[a^{(even)}\right] - \mathcal{S}^{1/2}\mathcal{F}\left[a^{(odd)}\right] \quad (1.30)$$

(Here it is silently assumed that '+' or '−' between two sequences denotes elementwise addition or subtraction.)

The length-n transform has been replaced by two transforms of length $n/2$. If n is a power of 2 this scheme can be applied recursively until length-one transforms (identity operation) are reached.

Thereby the operation count is improved to proportional $n/2\ \log_2(n)$.

Code 1.2 (recursive radix 2 DIT FFT) *Pseudo code for a recursive procedure of the (radix 2) DIT FFT algorithm,* `is` *must be +1 (forward transform) or -1 (backward transform):*

```
procedure rec_fft_dit2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace

    if n == 1 then  // end of recursion
    {
        x[0] := a[0]
        return
    }

    nh := n/2

    for k:=0 to nh-1  // copy to workspace
    {
        s[k] := a[2*k]    // even indexed elements
        t[k] := a[2*k+1]  // odd  indexed elements
    }

    // recursion: call two half-length FFTs:
    rec_fft_dit2(s[],nh,b[],is)
    rec_fft_dit2(t[],nh,c[],is)

    fourier_shift(c[],nh,is*1/2)

    for k:=0 to nh-1  // copy back from workspace
    {
        x[k]    := b[k] + c[k];
        x[k+nh] := b[k] - c[k];
    }
}
```

The data length `n` must be a power of 2. The result is in `x[]`. Note that normalisation (i.e. multiplication of each element of `x[]` by $1/\sqrt{n}$) is not included here.

[FXT: `recursive_dit2_fft` in file `learn/recfftdit2.cc`] The procedure uses the subroutine

Code 1.3 (Fourier shift) *For each element in c[0..n-1] replace c[k] by c[k] times* $e^{v\,2\pi\,i\,k/n}$. *Used with* $v = \pm 1/2$ *for the Fourier transform.*


```
procedure fourier_shift(c[], n, v)
{
    for k:=0 to n-1
    {
        c[k] := c[k] * exp(v*2.0*PI*I*k/n)
    }
}
```

cf. [FXT: `fourier_shift` in file `fft/fouriershift.cc`]

The recursive FFT-procedure involves a lot of function calls, this can be avoided by rewriting it in a non-recursive way. One can even do all operations *in place*, no workspace array is needed at all. The price is the necessity of an additional data reordering: The procedure `revbin_permute(a[],n)` rearranges the array `a[]` in a way that each element a_x is swapped with $a_{\tilde{x}}$, where $\tilde{x}$ is obtained from x by reversing its binary digits. This is discussed in section 1.7.

Code 1.4 (radix 2 DIT FFT, naive) *Pseudo code for a non-recursive procedure of the (radix 2) DIT algorithm,* `is` *must be -1 or +1: (naive version, needs to be improved)*

```
procedure fft_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn  // length of a[] is a power of 2
    revbin_permute(a[],n)
    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2
        for r:=0 to n-m step m  // n/m iterations
        {
            for j:=0 to mh-1  // m/2 iterations
            {
                e := exp(is*2*PI*I*j/m)  // log_2(n)*n/m*m/2 = log_2(n)*n/2 computations
                u := a[r+j]
                v := a[r+j+mh] * e
                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

[FXT: `dit2_fft_localized` in file `learn/fftdit2.cc`]

This version of a non-recursive FFT procedure already avoids the calling overhead and it works in place. It works as given, but is a bit wasteful. The (expensive!) computation `e := exp(is*2*PI*I*j/m)` is done `log_2(n) n/2` times. To reduce the number of trigonometric computations, one can swap the two inner loops, leading to the first 'real world' FFT procedure presented here:

Code 1.5 (radix 2 DIT FFT) *Pseudo code for a non-recursive procedure of the (radix 2) DIT algorithm,* `is` *must be -1 or +1:*

```
procedure fft_dit2(a[], ldn, is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn

    revbin_permute(a[],n)

    for ldm:=1 to ldn  // log_2(n) iterations
    {
        m  := 2**ldm
        mh := m/2

        for j:=0 to mh-1  // m/2 iterations
        {
            e := exp(is*2*PI*I*j/m)  // 1 + 2 + ... + n/8 + n/4 + n/2 = n-1 computations
```

```
            for r:=0 to n-m step m
            {
                u := a[r+j]
                v := a[r+j+mh] * e

                a[r+j]    := u + v
                a[r+j+mh] := u - v
            }
        }
    }
}
```

[FXT: `dit2_fft` in file `learn/fftdit2.cc`]

Swapping the two inner loops reduces the number of trigonometric (`exp()`) computations to n but leads to a feature that many FFT implementations share: Memory access is highly nonlocal. For each recursion stage (value of `ldm`) the array is traversed `mh` times with `n/m` accesses in strides of `mh`. As `mh` is a power of 2 this can (on computers that use memory cache) have a very negative performance impact for large values of n. On a computer where the CPU clock (366MHz, AMD K6/2) is 5.5 times faster than the memory clock (66MHz, EDO-RAM) I found that indeed for small n the naive FFT is slower by a factor of about 0.66, but for large n the same ratio is in favour of the 'naive' procedure!

It is a good idea to extract the `ldm==1` stage of the outermost loop, this avoids complex multiplications with the trivial factors $1 + 0\,i$: Replace

```
for ldm:=1 to ldn
{
```

by

```
for r:=0 to n-1 step 2
{
    {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
}
for ldm:=2 to ldn
{
```

1.3.3 Decimation in frequency (DIF) FFT

The simple splitting of the Fourier sum into a left and right half (for n even) leads to the decimation in frequency (DIF) FFT[3]:

$$\sum_{x=0}^{n-1} a_x\, z^{x\,k} = \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=n/2}^{n} a_x\, z^{x\,k} \tag{1.31}$$

$$= \sum_{x=0}^{n/2-1} a_x\, z^{x\,k} + \sum_{x=0}^{n/2-1} a_{x+n/2}\, z^{(x+n/2)\,k} \tag{1.32}$$

$$= \sum_{x=0}^{n/2-1} \left(a_x^{(left)} + z^{k\,n/2}\, a_x^{(right)}\right) z^{x\,k} \tag{1.33}$$

(where $z = e^{\pm i\,2\,\pi/n}$ and $k \in \{0, 1, ..., n-1\}$)

Here one has to distinguish the cases k even or odd, therefore we rewrite $k \in \{0, 1, 2, ..., n-1\}$ as $k = 2\,j + \delta$ where $j \in \{0, 2, ..., \frac{n}{2} - 1\}$, $\delta \in \{0, 1\}$.

$$\sum_{x=0}^{n-1} a_x\, z^{x\,(2\,j+\delta)} = \sum_{x=0}^{n/2-1} \left(a_x^{(left)} + z^{(2\,j+\delta)\,n/2}\, a_x^{(right)}\right) z^{x\,(2\,j+\delta)} \tag{1.34}$$

[3] also called Sande-Tukey FFT, cf. [28].

$$= \begin{cases} \sum_{x=0}^{n/2-1} \left(a_x^{(left)} + a_x^{(right)}\right) z^{2\,x\,j} & \text{for} \quad \delta = 0 \\ \sum_{x=0}^{n/2-1} z^x \left(a_x^{(left)} - a_x^{(right)}\right) z^{2\,x\,j} & \text{for} \quad \delta = 1 \end{cases} \tag{1.35}$$

$z^{(2\,j+\delta)\,n/2} = e^{\pm\pi\,i\,\delta}$ is equal to plus/minus 1 for $\delta = 0/1$ (k even/odd), respectively.

The last two equations are, more compactly written, the

Idea 1.2 (radix 2 DIF step) *Radix 2 decimation in frequency step for the FFT:*

$$\mathcal{F}\,[a]^{(even)} \stackrel{n/2}{=} \mathcal{F}\left[a^{(left)} + a^{(right)}\right] \tag{1.36}$$

$$\mathcal{F}\,[a]^{(odd)} \stackrel{n/2}{=} \mathcal{F}\left[S^{1/2}\left(a^{(left)} - a^{(right)}\right)\right] \tag{1.37}$$

Code 1.6 (recursive radix 2 DIF FFT) *Pseudo code for a recursive procedure of the (radix 2) decimation in frequency FFT algorithm,* `is` *must be +1 (forward transform) or -1 (backward transform):*

```
procedure rec_fft_dif2(a[], n, x[], is)
// complex a[0..n-1] input
// complex x[0..n-1] result
{
    complex b[0..n/2-1], c[0..n/2-1]   // workspace
    complex s[0..n/2-1], t[0..n/2-1]   // workspace
    if n == 1 then
    {
        x[0] := a[0]
        return
    }
    nh := n/2
    for k:=0 to nh-1
    {
        s[k] := a[k]     // 'left' elements
        t[k] := a[k+nh]  // 'right' elements
    }
    for k:=0 to nh-1
    {
        {s[k], t[k]} := {(s[k]+t[k]), (s[k]-t[k])}
    }
    fourier_shift(t[],nh,is*0.5)
    rec_fft_dif2(s[],nh,b[],is)
    rec_fft_dif2(t[],nh,c[],is)
    j := 0
    for k:=0 to nh-1
    {
        x[j]   := b[k]
        x[j+1] := c[k]
        j := j+2
    }
}
```

The data length n must be a power of 2. The result is in `x[]`.

[FXT: `recursive_dif2_fft` in file `learn/recfftdif2.cc`]

The non-recursive procedure looks like this:

Code 1.7 (radix 2 DIF FFT) *Pseudo code for a non-recursive procedure of the (radix 2) DIF algorithm,* `is` *must be -1 or +1:*

```
procedure fft_dif2(a[],ldn,is)
```

```
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn
    for ldm:=ldn to 1 step -1
    {
        m  := 2**ldm
        mh := m/2
        for j:=0 to mh-1
        {
            e := exp(is*2*PI*I*j/m)
            for r:=0 to n-1 step m
            {
                u := a[r+j]
                v := a[r+j+mh]
                a[r+j]    := (u + v)
                a[r+j+mh] := (u - v) * e
            }
        }
    }
    revbin_permute(a[],n)
}
```

cf. [FXT: `dif2_fft` in file `learn/fftdif2.cc`]

In DIF FFTs the `revbin_permute()`-procedure is called after the main loop, in the DIT code it was called before the main loop. As in the procedure 1.5 the inner loops where swapped to save unnecessary trigonometric computations.

Extracting the `ldm==1` stage of the outermost loop is again a good idea:
Replace the line

```
for  ldm:=ldn to 1 step -1
```

by

```
for  ldm:=ldn to 2 step -1
```

and insert

```
for r:=0 to n-1 step 2
{
    {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
}
```

before the call of `revbin_permute(a[],n)`.

1.4 Saving trigonometric computations

The trigonometric (sin()- and cos()-) computations are an expensive part of any FFT. There are two apparent ways for saving the involved cpu-cycles, the use of lookup-tables and recursive methods for trig-computations.

Using lookup tables

The idea is to save all necessary sin/cos-values in an array and later looking up the values needed. This is a good idea if one wants to compute many FFTs of the same (small) length. For FFTs of large sequences one gets large lookup tables that likely introduce a high cache-miss rate. Thereby one is likely experiencing little or no speed gain, even a slowdown isn't unlikely. However, for a length-n FFT one doesn't need to store all the (n complex or $2n$ real) sin/cos-values $\exp(2\pi i k/n)$, $k = 0, 1, 2, 3, ..., n-1$. Already a table $\cos(2\pi i k/n)$, $k = 0, 1, 2, 3, ..., n/4-1$ (of $n/4$ reals) contains all different trig-values that

occur in the computation. The size of the trig-table is thereby cut by a factor of 8. For the lookups one can use the symmetry relations

$$\cos(\pi + x) = -\cos(x) \quad (1.38)$$
$$\sin(\pi + x) = -\sin(x) \quad (1.39)$$

(reducing the interval from $0...2\pi$ to $0...\pi$),

$$\cos(\pi/2 + x) = -\sin(x) \quad (1.40)$$
$$\sin(\pi/2 + x) = +\cos(x) \quad (1.41)$$

(reducing the interval to $0...\pi/2$) and

$$\sin(x) = \cos(\pi/2 - x) \quad (1.42)$$

(only cos()-table needed).

Recursive trig-computation

In the computation of FFTs one typically needs the values

$$\{\exp(i\,\omega\,0) = 1,\quad \exp(i\,\omega\,\delta),\quad \exp(i\,\omega\,2\,\delta),\quad \exp(i\,\omega\,3\,\delta),\quad ...\}$$

in sequence. The naive idea for a recursive computation of these values is to precompute $d = \exp(i\,\omega\,\delta)$ and then compute the next following value using the identity $\exp(i\,\omega\,k\,\delta)) = d \cdot \exp(i\,\omega\,(k-1)\,\delta)$. This method, however, is of no practical value because the numerical error grows (exponentially) in the process.

Here is a stable version of a trigonometric recursion for the computation of the sequence: Precompute

$$c = \cos\omega, \quad (1.43)$$
$$s = \sin\omega, \quad (1.44)$$
$$\alpha = 2\,(\sin\frac{\delta}{2})^2 \quad (1.45)$$
$$\beta = \sin\delta \quad (1.46)$$

Then compute the next power from the previous as:

$$c_{next} = c - (\alpha\,c + \beta\,s); \quad (1.47)$$
$$s_{next} = s - (\alpha\,s - \beta\,c); \quad (1.48)$$

Do not expect to get all the precision you would get with the repeated call of the sin and cos functions, but even for very long FFTs less than 3 bits of precision are lost. When (in C) working with `doubles` it might be a good idea to use the type `long double` with the trig recursion: the sin and cos will than always be accurate within the `double`-precision.

Using higher radix algorithms

It may be less apparent, that the use of higher radix FFT algorithms also saves trig-computations. The radix-4 FFT algorithms presented in the next sections replace all multiplications with complex factors $(0, \pm i)$ by the obvious simpler operations. Radix-8 algorithms also simplify the special cases where $\sin(\phi)$ or $\cos(\phi)$ are $\pm\sqrt{1/2}$. Apart from the trig-savings higher radix also brings a performance gain by their more unrolled structure.

1.5 Higher radix DIT and DIF algorithms

1.5.1 More notation

Again some useful notation, again let a be a length-n sequence.

Let $a^{(r\%m)}$ denote the subsequence of those elements of a that have subscripts $x \equiv r \ (mod\ m)$; e.g. $a^{(0\%2)}$ is $a^{(even)}$, $a^{(3\%4)} = \{a_3, a_7, a_{11}, a_{15}, ...\}$. The length of $a^{(r\%m)}$ is[4] n/m.

Let $a^{(r/m)}$ denote the subsequence of those elements of a that have indices $\frac{r\,n}{m} ... \frac{(r+1)\,n}{m} - 1$; e.g. $a^{(1/2)}$ is $a^{(right)}$, $a^{(2/3)}$ is the last third of a. The length of $a^{(r/m)}$ is also n/m.

1.5.2 Decimation in time

First reformulate the radix 2 DIT step (formulas 1.29 and 1.30) in the new notation:

$$\mathcal{F}[a]^{(0/2)} \stackrel{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\left[a^{(0\%2)}\right]_{n/2} + \mathcal{S}^{1/2}\mathcal{F}\left[a^{(1\%2)}\right]_{n/2} \tag{1.49}$$

$$\mathcal{F}[a]^{(1/2)} \stackrel{n/2}{=} \mathcal{S}^{0/2}\mathcal{F}\left[a^{(0\%2)}\right]_{n/2} - \mathcal{S}^{1/2}\mathcal{F}\left[a^{(1\%2)}\right]_{n/2} \tag{1.50}$$

(Note that $\mathcal{S}^0$ is the identity operator).

The radix 4 step, whose derivation is analogue as for the radix 2 step, it just involves more writing and doesn't give additional insights, is

Idea 1.3 (radix 4 DIT step) *Radix 4 decimation in time step for the FFT:*

$$\mathcal{F}[a]^{(0/4)} \stackrel{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + \mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] + \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + \mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.51}$$

$$\mathcal{F}[a]^{(1/4)} \stackrel{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + i\sigma\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] - i\sigma\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.52}$$

$$\mathcal{F}[a]^{(2/4)} \stackrel{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] - \mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] + \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] - \mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.53}$$

$$\mathcal{F}[a]^{(3/4)} \stackrel{n/4}{=} +\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] - i\sigma\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] - \mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + i\sigma\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right] \tag{1.54}$$

where $\sigma = \pm 1$ is the sign in the exponent. In contrast to the radix 2 step, that happens to be identical for forward and backward transform (with both decimation frequency/time) the sign of the transform appears here.

Or, more compact:

$$\begin{aligned}\mathcal{F}[a]^{(j/4)} \stackrel{n/4}{=} &+e^{\sigma\,2\,i\,\pi\,0\,j/4}\cdot\mathcal{S}^{0/4}\mathcal{F}\left[a^{(0\%4)}\right] + e^{\sigma\,2\,i\,\pi\,1\,j/4}\cdot\mathcal{S}^{1/4}\mathcal{F}\left[a^{(1\%4)}\right] \\ &+e^{\sigma\,2\,i\,\pi\,2\,j/4}\cdot\mathcal{S}^{2/4}\mathcal{F}\left[a^{(2\%4)}\right] + e^{\sigma\,2\,i\,\pi\,3\,j/4}\cdot\mathcal{S}^{3/4}\mathcal{F}\left[a^{(3\%4)}\right]\end{aligned} \tag{1.55}$$

where $j = 0, 1, 2, 3$ and n is a multiple of 4.

Still more compact:

$$\mathcal{F}[a]^{(j/4)} \stackrel{n/4}{=} \sum_{k=0}^{3} e^{\sigma 2\,i\,\pi\,k\,j/4}\cdot\mathcal{S}^{\sigma k/4}\mathcal{F}\left[a^{(k\%4)}\right] \tag{1.56}$$

where the summation symbol denotes *elementwise* summation of the sequences. (The dot indicates multiplication of every element of the rhs. sequence by the lhs. exponential.)

The general radix r DIT step, applicable when n is a multiple of r, is:

[4] Throughout this book will m divide n, so the statement is correct.

Idea 1.4 (FFT general DIT step) *General decimation in time step for the FFT:*

$$\mathcal{F}[a]^{(j/r)} \stackrel{n/r}{=} \sum_{k=0}^{r-1} e^{\sigma\,2\,i\,\pi\,k\,j/r}\cdot\mathcal{S}^{\sigma\,k/r}\mathcal{F}\left[a^{(k\%r)}\right] \qquad j = 0, 1, 2, ..., r-1 \tag{1.57}$$

1.5.3 Decimation in frequency

The radix 2 DIF step (formulas 1.36 and 1.37) was

$$\mathcal{F}[a]_n^{(0\%2)} \stackrel{n/2}{=} \mathcal{F}\left[\mathcal{S}^{0/2}\left(a^{(0/2)} + a^{(1/2)}\right)\right] \tag{1.58}$$

$$\mathcal{F}[a]_n^{(1\%2)} \stackrel{n/2}{=} \mathcal{F}\left[\mathcal{S}^{1/2}\left(a^{(0/2)} - a^{(1/2)}\right)\right] \tag{1.59}$$

The radix 4 DIF step, applicable for n divisible by 4, is

Idea 1.5 (radix 4 DIF step) *Radix 4 decimation in frequency step for the FFT:*

$$\mathcal{F}[a]^{(0\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{0/4}\left(a^{(0/4)} + a^{(1/4)} + a^{(2/4)} + a^{(3/4)}\right)\right] \tag{1.60}$$

$$\mathcal{F}[a]^{(1\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{1/4}\left(a^{(0/4)} + i\sigma a^{(1/4)} - a^{(2/4)} - i\sigma a^{(3/4)}\right)\right] \tag{1.61}$$

$$\mathcal{F}[a]^{(2\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{2/4}\left(a^{(0/4)} - a^{(1/4)} + a^{(2/4)} - a^{(3/4)}\right)\right] \tag{1.62}$$

$$\mathcal{F}[a]^{(3\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{3/4}\left(a^{(0/4)} - i\sigma a^{(1/4)} - a^{(2/4)} + i\sigma a^{(3/4)}\right)\right] \tag{1.63}$$

Or, more compact:

$$\mathcal{F}[a]^{(j\%4)} \stackrel{n/4}{=} \mathcal{F}\left[\mathcal{S}^{\sigma\,j/4}\sum_{k=0}^{3} e^{\sigma\,2\,i\,\pi\,k\,j/4}\cdot a^{(k/4)}\right] \tag{1.64}$$

where $j = 0, 1, 2, 3$ and the sign of the exponent and in the shift operator is the same as in the transform.

The general radix r DIF step is

Idea 1.6 (FFT general DIF step) *General decimation in frequency step for the FFT:*

$$\mathcal{F}[a]^{(j\%r)} \stackrel{n/r}{=} \mathcal{F}\left[\mathcal{S}^{\sigma\,j/r}\sum_{k=0}^{r-1} e^{\sigma\,2\,i\,\pi\,k\,j/r}\cdot a^{(k/r)}\right] \qquad j = 0, 1, 2, ..., r-1 \tag{1.65}$$

1.5.4 Implementation of radix $r = p^x$ DIF/DIT FFTs

If $r = p \neq 2$ (p prime) then the `revbin_permute()` function has to be replaced by its radix-p version: `radix_permute()`. The reordering now swaps elements x with $\tilde{x}$ where $\tilde{x}$ is obtained from x by reversing its radix-p expansion.

Code 1.8 (radix p^x DIT FFT) *Pseudo code for a radix* `r:=`p^x *decimation in time FFT:*

```
procedure fftdit_r(a[], n, is)
// complex a[0..n-1] input, result
// p (hardcoded)
// r == power of p (hardcoded)
// n == power of p (not necessarily a power of r)
{
```

```
    radix_permute(a[], n, p)
    lx := log(r) / log(p)  // r == p ** lx
    ln := log(n) / log(p)
    ldm := (log(n)/log(p)) % lx
    if ( ldm != 0 )  // n is not a power of p
    {
        xx := p**lx
        for z:=0 to n-1 step xx
        {
            fft_dit_xx(a[z..z+xx-1], is)  // inlined length-xx dit fft
        }
    }
    for ldm:=ldm+lx to ln step lx
    {
        m  := p**ldm
        mr := m/r
        for j := 0 to mr-1
        {
            e := exp(is*2*PI*I*j/m)
            for k:=0 to n-1 step m
            {
                // all code in this block should be
                // inlined, unrolled and fused:
                // temporary  u[0..r-1]
                for z:=0 to r-1
                {
                    u[z] := a[k+j+mr*z]
                }
                radix_permute(u[], r, p)
                for z:=1 to r-1  // e**0 = 1
                {
                    u[z] := u[z] * e**z
                }
                r_point_fft(u[], is)
                for z:=0 to r-1
                {
                    a[k+j+mr*z] := u[z]
                }
            }
        }
    }
}
```

Of course the loops that use the variable `z` have to be unrolled, the (length-p^x) scratch space `u[]` has to be replaced by explicit variables (e.g. `u0, u1, ...`) and the `r_point_fft(u[],is)` shall be an inlined p^x-point FFT.

If $r = p^x$ than there is a pitfall one must now: if one uses the `radix_permute()` procedure instead of a radix-p^x revbin_permute procedure (e.g. radix-2 revbin_permute for a radix-4 FFT), then some additional reordering is necessary in the innermost loop: in the above pseudo code this is indicated by the `radix_permute(u[],p)` just before the `p_point_fft(u[],is)` line. One wouldn't really use a call to a procedure, but change indices in the loops where the `a[z]` are read/written for the DIT/DIF respectively. In the code below the respective lines have the comment `// (!)`.

It is wise to extract the stage of the main loop where the exp()-function always has the value 1, which is the case when `ldm==1` in the outermost loop[5]. In order not to restrict the possible array sizes to powers of p^x but only to powers of p one will supply adapted versions of the `ldm==1` -loop: e.g. for a radix-4 DIF FFT append a radix 2 step after the main loop if the array size is not a power of 4.

Code 1.9 (radix 4 DIT FFT) *C++ code for a radix 4 DIF FFT on the array* `f[]`*, the data length* `n` *must be a power of 2,* `is` *must be +1 or -1:*

```
static const ulong RX = 4;  // == r
```

[5] cf. section 5.3.


```
static const ulong LX = 2;  // == log(r)/log(p) == log_2(r)
void
dit41_fft(Complex *f, ulong ldn, int is)
// decimation in time radix 4 fft
// ldn == log_2(n)
{
    double s2pi = ( is>0 ? 2.0*M_PI : -2.0*M_PI );
    const ulong n = (1<<ldn);
    revbin_permute(f, n);
    ulong ldm = (ldn&1);  // == (log(n)/log(p)) % LX
    if ( ldm!=0 )  // n is not a power of 4, need a radix 2 step
    {
        for (ulong r=0; r<n; r+=2)
        {
            Complex a0 = f[r];
            Complex a1 = f[r+1];
            f[r]   = a0 + a1;
            f[r+1] = a0 - a1;
        }
    }
    ldm += LX;
    for ( ; ldm<=ldn ; ldm+=LX)
    {
        ulong m = (1<<ldm);
        ulong m4 = (m>>LX);
        double ph0 = s2pi/m;
        for (ulong j=0; j<m4; j++)
        {
            double phi = j*ph0;
            double c, s, c2, s2, c3, s3;
            sincos(phi, &s, &c);
            sincos(2.0*phi, &s2, &c2);
            sincos(3.0*phi, &s3, &c3);
            Complex e  = Complex(c,s);
            Complex e2 = Complex(c2,s2);
            Complex e3 = Complex(c3,s3);
            for (ulong r=0, i0=j+r;  r<n;  r+=m, i0+=m)
            {
                ulong i1 = i0 + m4;
                ulong i2 = i1 + m4;
                ulong i3 = i2 + m4;
                Complex a0 = f[i0];
                Complex a1 = f[i2]; // (!)
                Complex a2 = f[i1]; // (!)
                Complex a3 = f[i3];
                a1 *= e;
                a2 *= e2;
                a3 *= e3;
                Complex t0 = (a0+a2) + (a1+a3);
                Complex t2 = (a0+a2) - (a1+a3);
                Complex t1 = (a0-a2) + Complex(0,is) * (a1-a3);
                Complex t3 = (a0-a2) - Complex(0,is) * (a1-a3);
                f[i0] = t0;
                f[i1] = t1;
                f[i2] = t2;
                f[i3] = t3;
            }
        }
    }
}
```

Code 1.10 (radix 4 DIF FFT) *Pseudo code for a radix 4 DIF FFT on the array* `a[]`*, the data length* `n` *must be a power of 2,* `is` *must be +1 or -1:*

```
procedure fftdif4(a[],ldn,is)
// complex a[0..2**ldn-1] input, result
{
    n := 2**ldn
    for ldm := ldn to 2 step -2
    {
        m  := 2**ldm
        mr := m/4
        for j := 0 to mr-1
        {
            e  := exp(is*2*PI*I*j/m)
            e2 := e * e
            e3 := e2 * e
            for r := 0 to n-1 step m
            {
                u0 := a[r+j]
                u1 := a[r+j+mr]
                u2 := a[r+j+mr*2]
                u3 := a[r+j+mr*3]
                x := u0 + u2
                y := u1 + u3
                t0 := x + y  // == (u0+u2) + (u1+u3)
                t1 := x - y  // == (u0+u2) - (u1+u3)
                x := u0 - u2
                y := (u1 - u3)*I*is
                t2 := x + y  // == (u0-u2) + (u1-u3)*I*is
                t3 := x - y  // == (u0-u2) - (u1-u3)*I*is
                t1 := t1 * e
                t2 := t2 * e2
                t3 := t3 * e3
                a[r+j]      := t0
                a[r+j+mr]   := t2  // (!)
                a[r+j+mr*2] := t1  // (!)
                a[r+j+mr*3] := t3
            }
        }
    }
    if is_odd(ldn) then  // n not a power of 4
    {
        for r:=0 to n-1 step 2
        {
            {a[r], a[r+1]} := {a[r]+a[r+1], a[r]-a[r+1]}
        }
    }
    revbin_permute(a[],n)
}
```

Note the 'swapped' order in which `t1, t2` are copied back in the innermost loop, this is what `radix_permute(u[], r, p)` was supposed to do.

The multiplication by the imaginary unit (in the statement `y := (u1 - u3)*I*is`) should of course be implemented without any multiplication statement: one could unroll it as

```
(dr,di) := u1 - u2   // dr,di = real,imag part of difference
      if is>0 then  y := (-di,dr)  // use (a,b)*(0,+1) == (-b,a)
      else          y := (di,-dr)  // use (a,b)*(0,-1) == (b,-a)
```

In section 1.6 it is shown how the `if`-statement can be eliminated.

If `n` is not a power of 4, then `ldm` is odd during the procedure and at the last pass of the main loop one has `ldm=1`.

To improve the performance one will instead of the (extracted) radix 2 loop supply extracted radix 8 and radix 4 loops. Then, depending on whether `n` is a power of 4 or not one will use the radix 4 or the radix 8 loop, respectively. The start of the main loop then has to be
`for ldm := ldn to 3 step -X`
and at the last pass of the main loop one has `ldm=3` or `ldm=2`.

[FXT: `dit41_fft` in file `learn/fftdit41.cc`] [FXT: `dif41_fft` in file `learn/fftdif41.cc`] [FXT: `dit4_fft` in file `fft/fftdit4.cc`] [FXT: `dif4_fft` in file `fft/fftdif4.cc`]

Code 1.11 (radix permute) *C++ code for the radix permutation of the array* `f[]`,

```
extern ulong nt[];  // nt[] = 9, 90, 900  for r=10, x=3
extern ulong kt[];  // kt[] = 1, 10, 100  for r=10, x=3
template <typename Type>
void
radix_permute(Type *f, ulong n, ulong r)
//
// swap elements with index pairs i, j were the
// radix-r representation of i and j are mutually
// digit-reversed (e.g. 436 <--> 634)
//
// This is a radix-r generalization of revbin_permute()
// revbin_permute(f, n) =^= radix_permute(f, n, 2)
//
// must be:
//  n == p**x for some x>=1
//  r >= 2
//
{
    ulong x = 0;
    nt[0] = r-1;
    kt[0] = 1;
    while ( 1 )
    {
        ulong z = kt[x] * r;
        if ( z>n )  break;
        ++x;
        kt[x] = z;
        nt[x] = nt[x-1] * r;
    }
    // here: n == p**x
    for (ulong i=0, j=0;  i < n-1;  i++)
    {
        if ( i<j )  swap(f[i], f[j]);
        ulong t = x - 1;
        ulong k = nt[t];  // =^=  k = (r-1) * n / r;
        while ( k<=j )
        {
            j -= k;
            k = nt[--t];  // =^=  k /= r;
        }
        j += kt[t]; // =^=  j += (k/(r-1));
    }
}
```

[FXT: `radix_permute` in file `permute/radixpermute.h`]

1.6 Inverse FFT for free

Suppose you programmed some FFT algorithm just for one value of `is`, the sign in the exponent. There is a nice trick that gives the inverse transform for free, if your implementation uses seperate arrays for real and imaginary part of the complex sequences to be transformed. If your procedure is something like

```
procedure my_fft(ar[], ai[], ldn)  // only for is==+1 !
// real ar[0..2**ldn-1] input, result, real part
// real ai[0..2**ldn-1] input, result, imaginary part
{
    // incredibly complicated code
    // that you can't see how to modify
    // for is==-1
}
```

Then you *don't* need to modify this procedure at all in order to get the inverse transform. If you want the inverse transform somewhere then just, instead of

```
my_fft(ar[], ai[], ldn)  // forward fft
```

type

```
my_fft(ai[], ar[], ldn)  // backward fft
```

Note the swapped real- and imaginary parts ! The same trick works if your procedure coded for fixed is$=-1$.

1.7 The revbin permute operation

The procedure `revbin_permute(a[],n)` used in the DIF and DIT FFT algorithms rearranges the array `a[]` in a way that each element a_x is swapped with $a_{\tilde{x}}$, where $\tilde{x}$ is obtained from x by reversing its binary digits. For example if $n = 256$ and $x = 43_{10} = 00101011_2$ then $\tilde{x} = 11010100_2 = 212_{10}$. Note that $\tilde{x}$ depends both on x and on n.

A naive version

Code 1.12 (revbin_permute, naive)

```
procedure revbin_permute(a[],n)
// a[0..n-1] input,result
{
    for x:=0 to n-1
    {
        r := revbin(x,n)
        if r>x then  swap(a[x],a[r])
    }
}
```

The function `revbin(x,n)` shall return the reversed bits of `x`.

Code 1.13 (revbin)

```
function revbin(x,n)
{
    j := 0
    ldn := log2(n)  // is an integer
    while ldn>0
    {
        j := j << 1
        j := j + (x & 1)
        x := x >> 1
        ldn := ldn - 1
    }
    return j
}
```

The condition `r>x` before the `swap()` statement makes sure that the swapping isn't undone when the loop variable `x` has the value of the present `r`. This version of the `revbin_permute`-routine is pretty unefficient (even if `revbin()` is inlined and `ldn` is only computed once). Each execution of `revbin()` costs proportional `ldn` operations, giving a total of proportional $\frac{n}{2}\log_2(n)$ operations (neglecting the swaps for the moment). One can do better.

A fast version

The key idea is to compute the value $\tilde{x}$ from the value $\widetilde{x-1}$. As x is one added to $x-1$, $\tilde{x}$ is one 'reversed' added to $\widetilde{x-1}$ if one finds a routine for that 'reversed add' update much of the computation can be saved.

Code 1.14 (revbin update) *Update* `r`*, that must be the same as the the result of* `revbin(x-1,n)` *to what would be the result of* `revbin(x,n)`

```
function revbin_update(r,n)
{
    do
    {
        n := n >> 1
        r := r^n  // bitwise exor
    } while ((r&n) == 0)
    return r
}
```

In C this can be cryptified to an efficient piece of code:

```
inline unsigned revbin_update(unsigned r, unsigned n)
{
    for (unsigned m=n>>1; (!((r^=m)&m)); m>>=1);
    return r;
}
```

Now we are ready for

Code 1.15 (revbin_permute, fast) *Put data in revbin order*

```
procedure revbin_permute(a[],n)
// a[0..n-1] input,result
{
    if n<=2  return
    r := 0  // the reversed 0
    for x:=1 to n-1
    {
        r := revbin_update(r,n)  // inline me
        if r>x then  swap(a[x],a[r])
    }
}
```

This routine is several times faster than the naive version. `revbin_update()` does for half of the calls just one iteration because in half of the updates just the leftmost bit changes[6], in half of the remaining updates it does two iterations, in half of the still remaining updates it three and so on. The total number operations done by `revbin_update()` is therefore proportional to $n\,(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots+\frac{\log_2(n)}{n})$ which is $n\sum_{j=1}^{\log_2(n)}\frac{j}{2^j}$ for n large this sum converges against $2n$. Thereby the asymptotics of `revbin_permute()` is improved from proportional $n\,\log(n)$ to proportional n.

How many swaps ?

How many `swap()`-statements will be executed in total for different n ? About $n-\sqrt{n}$, as there are only few numbers with symmetric bit patterns: for even $log_2(n) =: 2\,b$ the left half of the bit pattern must be the reversed of the right half. There are $2^b = \sqrt{2^{2b}}$ such numbers. For odd $log_2(n) =: 2\,b+1$ there are twice as much symmetric patterns, the bit in the middle does not matter and can be 0 or 1.

[6] corresponding to the change in only the rightmost bit if one is added to an even number

n	2 # swaps	# symm. pairs
2	0	2
4	2	2
8	4	4
16	12	4
32	24	8
64	56	8
2^{10}	992	32
2^{20}	$0.999 \cdot 2^{20}$	2^{10}
∞	$n - \sqrt{n}$	$\sqrt{n}$

Summarizing: almost all 'revbin-pairs' will be swapped by `revbin_permute()`.

A still faster version

x	x_2	$\tilde{x}_2$	$\tilde{x}$	Δ	$\tilde{x} > x$?
0	00000	00000	0	-31	
1	00001	10000	16	16	y
2	00010	01000	8	-8	y
3	00011	11000	24	16	y
4	00100	00100	4	-20	
5	00101	10100	20	16	y
6	00110	01100	12	-8	y
7	00111	11100	28	16	y
8	01000	00010	2	-26	
9	01001	10010	18	16	y
10	01010	01010	10	-8	
11	01011	11010	26	16	y
12	01100	00110	6	-20	
13	01101	10110	22	16	y
14	01110	01110	14	-8	
15	01111	11110	30	16	y
16	10000	00001	1	-29	
17	10001	10001	17	16	
18	10010	01001	9	-8	
19	10011	11001	25	16	y
20	10100	00101	5	-20	
21	10101	10101	21	16	
22	10110	01101	13	-8	
23	10111	11101	29	16	y
24	11000	00011	3	-26	
25	11001	10011	19	16	
26	11010	01011	11	-8	
27	11011	11011	27	16	
28	11100	00111	7	-20	
29	11101	10111	23	16	
30	11110	01111	15	-8	
31	11111	11111	31	16	

where the subscript 2 indicates printing in base 2, $\Delta := \tilde{x} - \widetilde{x-1}$ and an 'y' in the last column marks index pairs where `revbin_permute()` will swap elements.

Observation one: $\Delta = \frac{n}{2}$ for all odd x.

Observation two: if for even $x < \frac{n}{2}$ there is a swap (for the pair x, $\tilde{x}$) then there is also a swap for the pair $n-1-x$, $n-1-\tilde{x}$. As $x < \frac{n}{2}$ and $\tilde{x} < \frac{n}{2}$ one has $n-1-x > \frac{n}{2}$ and $n-1-\tilde{x} > \frac{n}{2}$, i.e. the swaps are independent.

There should be no difficulties to cast these observations into

Code 1.16 (revbin_permute, faster) *Put data in revbin order*

```
procedure revbin_permute(a[],n)
{
    if n<=2 return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        swap(a[x],a[r])
        x := x + 1

        // x even:
        r := revbin_update(r,n)  // inline me
        if r>x then
        {
            swap(a[x],a[r])
            swap(a[n-1-x],a[n-1-r])
        }
        x := x + 1
    }
}
```

The `revbin_update()` would be in C, inlined and the first stage of the loop extracted

```
r^=nh;  for (unsigned m=(nh>>1); !((r^=m)&m); m>>=1)  {}
```

The code above is an ideal candidate to derive an optimised version for zero padded data:

Code 1.17 (revbin_permute for zero padded data) *Put zero padded data in revbin order*

```
procedure revbin_permute0(a[],n)
{
    if n<=2 return
    nh := n/2
    r := 0  // the reversed 0
    x := 1
    while x<nh
    {
        // x odd:
        r := r + nh
        a[r] := a[x]
        a[x] := 0
        x := x + 1

        // x even:
        r := revbin_update(r,n)  // inline me
        if r>x then  swap(a[x],a[r])
        x := x + 1
    }
}
```

One could carry the scheme that lead to the 'faster' revbin_permute procedures further, e.g. using 3 hardcoded constants Δ_1, Δ_2, Δ_3 depending on whether $x \bmod 4 = 1, 2, 3$ only calling `revbin_update()` for $x \bmod 4 = 0$. However, the code quickly gets quite complicated and there seems to be no measurable gain in speed, even for very large sequences.

If, for complex data, one works with seperate arrays for real and imaginary part[7] one might be tempted to do away with half of the bookkeeping as follows: write a special procedure `revbin_permute(a[],b[],n)` that shall replace the two successive calls `revbin_permute(a[],n)` and `revbin_permute(b[],n)` and has after each statement `swap(a[x],a[r])` inserted a `swap(b[x],b[r])`. If you do so, be prepared for disaster! Very likely the real and imaginary element for the same index lie apart in memory by a power of two, leading to one hundred percent cache miss for the typical computer. Even in the most favourable case the cache miss rate will be increased. Do expect to hardly ever win anything noticable but in most cases to lose big. Think about it, whisper "*direct mapped cache*" and forget it.

[FXT: `revbin_permute` in file `permute/revbinpermute.h`]

[7] as opposed to: using a data type 'complex' with real and imaginary part of each number in consecutive places

1.8 Real valued Fourier transforms

The Fourier transform of a purely real sequence $c = \mathcal{F}[a]$ where $a \in \mathbb{R}$ has[8] a symmetric real part ($\Re\bar{c} = \Re c$) and an antisymmetric imaginary part ($\Im\bar{c} = -\Im c$). Simply using a complex FFT for real input is basically a waste of a factor 2 of memory and CPU cycles. There are several ways out:

- sincos wrappers for complex FFTs
- usage of the fast Hartley transform
- a variant of the matrix Fourier algorithm
- special real (split radix algorithm) FFTs

All techniques have in common that they store only half of the complex result to avoid the redundancy due to the symmetries of a complex FT of purely real input. The result of a real to (half-) complex FT (abbreviated R2CFT) must contain the purely real components c_0 (the DC-part of the input signal) and, in case n is even, $c_{n/2}$ (the nyquist frequency part). The inverse procedure, the (half-) complex to real transform (abbreviated C2RFT) must be compatible to the ordering of the R2CFT. The procedures presented here use the following ordering of the real part of the resulting data c in the output array a[]:

$$\begin{aligned} \mathrm{a[0]} &= \Re c_0 \\ \mathrm{a[1]} &= \Re c_1 \\ \mathrm{a[2]} &= \Re c_2 \\ &\dots \\ \mathrm{a[n/2]} &= \Re c_{n/2} \end{aligned} \tag{1.66}$$

The imaginary part of the result is stored like

$$\begin{aligned} \mathrm{a[n/2+1]} &= \Im c_1 \\ \mathrm{a[n/2+2]} &= \Im c_2 \\ \mathrm{a[n/2+3]} &= \Im c_3 \\ &\dots \\ \mathrm{a[n-1]} &= \Im c_{n/2-1} \end{aligned} \tag{1.67}$$

except for the Hartley transform based R2CFT, which uses the reversed order for the imaginary part

$$\begin{aligned} \mathrm{a[n/2+1]} &= \Im c_{n/2-1} \\ \mathrm{a[n/2+2]} &= \Im c_{n/2-2} \\ \mathrm{a[n/2+3]} &= \Im c_{n/2-3} \\ &\dots \\ \mathrm{a[n-1]} &= \Im c_1 \end{aligned} \tag{1.68}$$

Note the absence of the elements $\Im c_0$ and $\Im c_{n/2}$ which are zero.

1.8.1 Real valued FT via wrapper routines

A simple way to use a complex length-$n/2$ FFT for a real length-n FFT (n even) is to use some post- and preprocessing routines. For a real sequence a one feeds the (half length) complex sequence $f = a^{(even)} + i\,a^{(odd)}$ into a complex FFT. Some postprocessing is necessary. This is not the most elegant real FFT available, but it is directly usable to turn complex FFTs of any (even) length into a real-valued FFT.

Here is the

[8] cf. relation 1.20

Code 1.18 (R2CFT with wrap routines) *C++ code for a real to complex FFT (R2CFT):*

```
void
wrap_real_complex_fft(double *f, ulong ldn, int is/*=+1*/)
//
// ordering of output:
// f[0]      = re[0]   (DC part, purely real)
// f[1]      = re[n/2] (nyquist freq, purely real)
// f[2]      = re[1]
// f[3]      = im[1]
// f[4]      = re[2]
// f[5]      = im[2]
//            ...
// f[2*i]    = re[i]
// f[2*i+1]  = im[i]
//            ...
// f[n-2]    = re[n/2-1]
// f[n-1]    = im[n/2-1]
//
// equivalent:
// { fht_real_complex_fft(f, ldn, is); evenodd_permute(f, n); }
//
{
    if ( ldn==0 )  return;

    fht_fft((Complex *)f, ldn-1, +1);

    const ulong n = 1<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]

        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]

        double f1r, f2i;
        sumdiff05(f[i3], f[i1], f1r, f2i);

        double f2r, f1i;
        sumdiff05(f[i2], f[i4], f2r, f1i);

        double c, s;
        double phi = i*phi0;
        sincos(phi, &s, &c);

        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);

        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);


        // f[i4] = is * (ti + f1i);  // im hi
        // f[i2] = is * (ti - f1i);  // im low
        // =^=
        if ( is>0 )  sumdiff( ti,  f1i, f[i4], f[i2]);
        else         sumdiff(-ti,  f1i, f[i2], f[i4]);
    }
    sumdiff(f[0], f[1]);

    if ( n>=4 )  f[nh+1] *= is;
}
```

Code 1.19 (C2RFT, with wrap routines) *C++ code for a complex to real FFT (C2RFT):*

```
void
wrap_complex_real_fft(double *f, ulong ldn, int is/*=+1*/)
//
// inverse of wrap_real_complex_fft()
//
// ordering of input:
// like the output of wrap_real_complex_fft()
{
```

```
    if ( ldn==0 )  return;
    const ulong n = 1<<ldn;
    const ulong nh = n/2, n4 = n/4;
    const double phi0 = -M_PI / nh;
    for(ulong i=1; i<n4; i++)
    {
        ulong i1 = 2 * i;   // re low [2, 4, ..., n/2-2]
        ulong i2 = i1 + 1;  // im low [3, 5, ..., n/2-1]
        ulong i3 = n - i1;  // re hi  [n-2, n-4, ..., n/2+2]
        ulong i4 = i3 + 1;  // im hi  [n-1, n-3, ..., n/2+3]
        double f1r, f2i;
        // double f1r =  f[i1] + f[i3];  // re symm
        // double f2i =  f[i1] - f[i3];  // re asymm
        // =^=
        sumdiff(f[i1], f[i3], f1r, f2i);
        double f2r, f1i;
        // double f2r = -f[i2] - f[i4];  // im symm
        // double f1i =  f[i2] - f[i4];  // im asymm
        // =^=
        sumdiff(-f[i4], f[i2], f1i, f2r);
        double c, s;
        double phi = i*phi0;
        sincos(phi, &s, &c);
        double tr, ti;
        cmult(c, s, f2r, f2i, tr, ti);
        // f[i1] = f1r + tr;  // re low
        // f[i3] = f1r - tr;  // re hi
        // =^=
        sumdiff(f1r, tr, f[i1], f[i3]);
        // f[i2] = ti - f1i;  // im low
        // f[i4] = ti + f1i;  // im hi
        // =^=
        sumdiff(ti, f1i, f[i4], f[i2]);
    }
    sumdiff(f[0], f[1]);
    if ( n>=4 )  { f[nh] *= 2.0; f[nh+1] *= 2.0; }
    fht_fft((Complex *)f, ldn-1, -1);
    if ( is<0 )  reverse_nh(f, n);
}
```

[FXT: `wrap_real_complex_fft` in file `realfft/realfftwrap.cc`]

[FXT: `wrap_complex_real_fft` in file `realfft/realfftwrap.cc`]

1.9 The matrix algorithm (MFA)

The matrix Fourier algorithm[9] (MFA) works for (composite) data lengths $n = RC$. Consider the input array as a $R \times C$-matrix (R rows, C columns).

Idea 1.7 (matrix Fourier algorithm) *The matrix Fourier algorithm (MFA) for the FFT:*

1. *Apply a (length R) FFT on each column.*
2. *Multiply each matrix element (index r, c) by $\exp(\pm 2\pi i\, r\, c/n)$ (sign is that of the transform).*
3. *Apply a (length C) FFT on each row.*
4. *Transpose the matrix.*

[9] A variant of the MFA is called 'four step FFT' in [127].

Note the elegance!

It is trivial to rewrite the MFA as the

Idea 1.8 (transposed matrix Fourier algorithm) *The transposed matrix Fourier algorithm (TMFA) for the FFT:*

1. *Transpose the matrix.*
2. *Apply a (length C) FFT on each column (transposed row).*
3. *Multiply each matrix element (index r, c) by $\exp(\pm 2\pi i\, r\, c/n)$.*
4. *Apply a (length R) FFT on each row (transposed column).*

FFT algorithms are usually very memory nonlocal, i.e. the data is accessed in strides with large skips (as opposed to e.g. in unit strides). In radix 2 (or 2^n) algorithms one even has skips of powers of 2, which is particularly bad on computer systems that use *direct mapped cache* memory: One piece of cache memory is responsible for caching addresses that lie apart by some power of 2. With an 'usual' FFT algorithm one gets 100% cache misses and therefore a memory performance that corresponds to the access time of the main memory, which is very long compared to the clock of modern CPUs. The matrix Fourier algorithm has a much better memory locality (cf. [127]), because the work is done in the short FFTs over the rows and columns.

For the reason given above the computation of the column FFTs should not be done in place. One can insert additional transpositions in the algorithm to have the columns lie in contiguous memory when they are worked upon. The easy way is to use an additional scratch space for the column FFTs, then only the copying from and to the scratch space will be slow. If one interleaves the copying back with the exp()-multiplications (to let the CPU do some work during the wait for the memory access) the performance should be ok. Moreover, one can insert small offsets (a few unused memory words) at the end of each row in order to avoid the cache miss problem almost completely. Then one should also program a procedure that does a 'mass production' variant of the column FFTs, i.e. for doing computation for all rows at once.

It is usually a good idea to use factors of the data length n that are close to $\sqrt{n}$. Of course one can apply the same algorithm for the row (or column) FFTs again: It can be a good idea to split n into 3 factors (as close to $n^{1/3}$ as possible) if a length-$n^{1/3}$ FFT fits completely into the second level cache (or even the first level cache) of the computer used. Especially for systems where CPU clock is much higher than memory clock the performance may increase drastically, a performance factor of two (even when compared to else very good optimised FFTs) can be observed.

1.10 Convolutions

The cyclic convolution of two sequences a and b is defined as the sequence h with elements h_τ as follows:

$$\begin{aligned} h &= a \circledast b \\ h_\tau &:= \sum_{x+y \equiv \tau \pmod n} a_x\, b_y \end{aligned} \tag{1.69}$$

The last equation may be rewritten as

$$h_\tau := \sum_{x=0}^{n-1} a_x\, b_{\tau - x} \tag{1.70}$$

where negative indices $\tau - x$ must be understood as $n + \tau - x$, it's a cyclic convolution.

Code 1.20 (cyclic convolution by definition) *Compute the cyclic convolution of* `a[]` *with* `b[]` *using the definition, result is returned in* `c[]`

```
procedure convolution(a[],b[],c[],n)
{
    for tau:=0 to n-1
    {
        s := 0
        for x:=0 to n-1
        {
            tx := tau-x
            if tx<0 then  tx := tx+n
            s := s + a[x]*b[tx]
        }
        c[tau] := s
    }
}
```

This procedure uses (for length-n sequences a, b) proportional n^2 operations, therefore it is slow for large values of n. The Fourier transform provides us with a more efficient way to compute convolutions that only uses proportional n log(n) operations. First we have to establish the convolution property of the Fourier transform:

$$\mathcal{F}[a \circledast b] = \mathcal{F}[a]\,\mathcal{F}[b] \tag{1.71}$$

i.e. convolution in original space is ordinary (elementwise) multiplication in Fourier space.

Here is the proof:

$$\begin{aligned}
\mathcal{F}[a]_k\,\mathcal{F}[b]_k &= \sum_x a_x\, z^{kx} \sum_y b_y\, z^{ky} \qquad (1.72)\\
&\quad \text{with} \quad y := \tau - x\\
&= \sum_x a_x\, z^{kx} \sum_{\tau-x} b_{\tau-x}\, z^{k(\tau-x)}\\
&= \sum_x \sum_{\tau-x} a_x\, z^{kx} b_{\tau-x}\, z^{k(\tau-x)}\\
&= \sum_\tau \left(\sum_x a_x\, b_{\tau-x} \right) z^{k\tau}\\
&= \left(\mathcal{F}\left[\sum_x a_x\, b_{\tau-x} \right] \right)_k\\
&= (\mathcal{F}[a \circledast b])_k
\end{aligned}$$

Rewriting formula 1.71 as

$$a \circledast b = \mathcal{F}^{-1}[\mathcal{F}[a]\,\mathcal{F}[b]] \tag{1.73}$$

tells us how to proceed:

Code 1.21 (cyclic convolution via FFT) *Pseudo code for the cyclic convolution of two complex valued sequences* `x[]` *and* `y[]`*, result is returned in* `y[]`*:*

```
procedure fht_cyclic_convolution(x[],y[],n)
{
    complex x[0..n-1], y[0..n-1]

    // transform data:
    fft(x[],n,+1)
    fft(y[],n,+1)

    // convolution in transformed domain:
    for i:=0 to n-1
    {
        y[i] := y[i] * x[i]
    }

    // transform back:
    fft(y[],n,-1)

    // normalise:
    for i:=0 to n-1
    {
        y[i] := y[i]/n
    }
}
```

It is assumed that the procedure `fft()` does no normalisation. In the normalisation loop you precompute `1.0/n` and multiply as divisions are much slower than multiplications.

real convolutions: ... [FXT: `fht_fft_convolution` in file `fft/fftcnvl.cc`] [FXT: `split_radix_fft_convolution` in file `fft/fftcnvl.cc`]

Auto (or self) convolution is defined as

$$\begin{aligned}
h &= a \circledast a \qquad (1.74)\\
h_\tau &:= \sum_{x+y \equiv \tau (n)} a_x\, a_y
\end{aligned}$$

The corresponding procedure should be obvious. [FXT: `fht_convolution` in file `fht/fhtcnvl.cc`] [FXT: `fht_convolution0` in file `fht/fhtcnvl.cc`]

In the definition of the cyclic convolution (1.69) one can distinguish between those summands where the $x+y$ 'wrapped around' (i.e. $x+y = n+\tau$) and those where simply $x+y = \tau$ holds. These are (following the notation in [77]) denoted by $h^{(1)}$ and $h^{(0)}$ respectively. Then

$$h = h^{(0)} + h^{(1)} \tag{1.75}$$

where

$$\begin{aligned}
h^{(0)} &= \sum_{x \le \tau} a_x\, b_{\tau-x}\\
h^{(1)} &= \sum_{x > \tau} a_x\, b_{n+\tau-x}
\end{aligned}$$

There is a simple way to seperate $h^{(0)}$ and $h^{(1)}$ as the left and right half of a length-$2n$ sequence. This is just what the *acyclic* (or *linear*) convolution does: Acyclic convolution of two (length-n) sequences a and b can be defined as that length-$2n$ sequence h which is the cyclic convolution of the *zero padded* sequences A and B:

$$A := \{a_0, a_1, a_2, ..., a_{n-1}, 0, 0, ..., 0\} \tag{1.76}$$

Same for B. Then

$$h_\tau := \sum_{x=0}^{2n-1} A_x\, B_{\tau-x} \qquad \tau = 0, 1, 2, ..., 2n-1 \tag{1.77}$$

$$\sum_{\substack{x+y \equiv \tau (2n)\\ x,y<2n}} a_x\, b_y = \sum_{0 \le x < n} a_x\, b_y + \sum_{n \le x < 2n} a_x\, b_y \tag{1.78}$$

where the right sum is zero because $a_x = 0$ for $n \le x < 2n$. Now

$$\sum_{0 \le x < n} a_x\, b_y = \sum_{x \le \tau} a_x\, b_{\tau-x} + \sum_{x > \tau} a_x\, b_{2n+\tau-x} =: R_\tau + S_\tau \tag{1.79}$$

where the rhs. sums are silently understood as restricted to $0 \le x < n$.

For $0 \le \tau < n$ the sum S_τ is always zero because $b_{2n+\tau-x}$ is zero ($n \le 2n+\tau-x < 2n$ for $0 \le \tau - x < n$); the sum R_τ is already equal to $h_\tau^{(0)}$. For $n \le \tau < 2n$ the sum S_τ is again zero, this time because it extends over nothing (simultaneous conditions $x < n$ and $x > \tau \ge n$); R_τ can be identified with $h_{\tau'}^{(1)}$ ($0 \le \tau' < n$) by setting $\tau = n + \tau'$.

As an illustration consider the convolution of the sequence $\{1,1,1,1\}$ with itself: its linear self convolution is $\{1,2,3,4,3,2,1,0\}$, its cyclic self convolution is $\{4,4,4,4\}$, i.e. the right half of the linear convolution elementwise added to the left half.

By the way, relation 1.71 is also true for the more general z-transform, but there is no (simple) back-transform, so we cannot turn (the analogue of 1.73)

$$a \circledast b \quad = \quad \mathcal{Z}^{-1}\left[\mathcal{Z}[a]\,\mathcal{Z}[b]\right] \tag{1.80}$$

into a practical algorithm.

1.11 Mass storage convolution using the MFA

The matrix Fourier algorithm is also an ideal candidate for (adaption for) mass storage FFTs, i.e. FFTs for data sets that do not fit into physical RAM[10].

In convolution computations it is straightforward to save the transpositions by using the MFA followed by the TMFA. (The data is assumed to be in memory as row_0, row_1, ..., row_{R-1}, i.e. the way array data is stored in memory in the C language, as opposed to the Fortran language.) For simplicity auto convolution is considered here:

Idea 1.9 (matrix convolution algorithm) *The matrix convolution algorithm:*

1. *Apply a (length R) FFT on each column.*
   (memory access with C-skips)
2. *Multiply each matrix element (index r, c) by* $\exp(\pm 2\pi i\, r\, c/n)$.
3. *Apply a (length C) FFT on each row.*
   (memory access without skips)
4. *Complex square row (elementwise).*
5. *Apply a (length C) FFT on each row (of the transposed matrix).*
   (memory access is without skips)
6. *Multiply each matrix element (index r, c) by* $\exp(\mp 2\pi i\, r\, c/n)$.
7. *Apply a (length R) FFT on each column (of the transposed matrix).*
   (memory access with C-skips)

Note that steps 3, 4 and 5 constitute a length-C convolution.

[FXT: `matrix_convolution` in file `matrix/matrixcnvl.cc`] [FXT: `matrix_convolution0` in file `matrix/matrixcnvl.cc`] [FXT: `matrix_auto_convolution` in file `matrix/matrixcnvla.cc`] [FXT: `matrix_auto_convolution0` in file `matrix/matrixcnvla.cc`]

A simple consideration lets one use the above algorithm for *mass storage convolutions*, i.e. convolutions of data sets that do not fit into the RAM workspace. An important consideration is the

[10] The naive idea to simply try such an FFT with the virtual memory mechanism will of course, due to the nonlocality of FFTs, end in eternal harddisk activity

Minimisation of the number of disk seeks

The number of disk seeks has to be kept minimal because these are slow operations which, if occur too often, degrade performance unacceptably.

The crucial modification of the use of the MFA is *not* to choose R and C as close as possible to $\sqrt{n}$ as usually done. Instead one chooses R minimal, i.e. the row length C corresponds to the biggest data set that fits into the RAM memory[11]. We now analyse how the number of seeks depends on the choice of R and C: in what follows it is assumed that the data lies in memory as row_0, row_1, ..., row_{R-1}, i.e. the way array data is stored in the C language, as opposed to the Fortran language convention. Further let $\alpha \ge 2$ be the number of times the data set exceeds the RAM size.

In step 1 and 3 of algorithm 1.14 one reads from disk (row by row, involving R seeks) the number of colums that just fit into RAM, does the (many, short) column-FFTs[12], writes back (again R seeks) and proceeds to the next block; this happens for α of these blocks, giving a total of $4\,\alpha\,R$ seeks for steps 1 and 3.

In step 2 one has to read (α times) blocks of one or more rows, which lie in contiguous portions of the disk, perform the FFT on the rows and write back to disk, leading to a total of $2\,\alpha$ seeks.

Thereby one has a number of $2\,\alpha + 4\,\alpha\,R$ seeks during the whole computation, which is minimised by the choice of maximal C. This means that one chooses a shape of the matrix so that the rows are as big as possible subject to the constraint that they have to fit into main memory, which in turn means there are $R = \alpha$ rows, leading to an optimal seek count of $K = 2\,\alpha + 4\,\alpha^2$.

If one seek takes 10 milliseconds then one has for $\alpha = 16$ (probably quite a big FFT) a total of $K \cdot 10 = 1056 \cdot 10$ milliseconds or approximately 10 seconds. With a RAM workspace of 64 Megabytes[13] the CPU time alone might be in the order of several minutes. The overhead for the (linear) read and write would be (throughput of 10MB/sec assumed) $6 \cdot 1024MB/(10MB/sec) \approx 600sec$ or approximately 10 minutes.

With a multithreading OS one may want to produce a 'double buffer' variant: choose the row length so that it fits twice into the RAM workspace; then let always one (CPU-intensive) thread do the FFTs in one of the scratch spaces and another (hard disk intensive) thread write back the data from the other scratch-space and read the next data to be processed. With not too small main memory (and not too slow hard disk) and some fine tuning this should allow to keep the CPU busy during much of the hard disk operations.

The remarks about the computation of the column FFTs on page 27 also apply here.

1.12 Weighted Fourier transforms

Let us define a new kind of transform by slightly modifying the definition of the FT (cf. formula 1.1):

$$\begin{aligned} c &= \mathcal{W}_v[a] \\ c_k &:= \sum_{x=0}^{n-1} v_x\, a_x\, z^{x\,k} \qquad v_x \neq 0 \quad \forall x \end{aligned} \tag{1.81}$$

where $z := e^{\pm 2\pi i/n}$. The sequence c shall be called weighted (discrete) transform of the sequence a with the weight (sequence) v. Note the v_x that entered: the weighted transform with $v_x = \frac{1}{\sqrt{n}}\ \forall x$ is just the usual Fourier transform. The inverse transform is

$$a \quad = \quad \mathcal{W}_v^{-1}[c] \tag{1.82}$$

[11] more precisely: the amount of RAM where no swapping will occur, some programs plus the operating system have to be there, too.

[12] real-complex FFTs in step 1 and complex-real FFTs in step 3.

[13] allowing for 8 million 8 byte floats, so the total FFT size is $S = 16 \cdot 64 = 1024$ MB or 32 million floats

$$a_x = \frac{1}{n\,v_x}\sum_{k=0}^{n-1} c_k\, z^{-x\,k}$$

This can be easily seen:

$$\begin{aligned}\mathcal{W}_v^{-1}\left[\mathcal{W}_v\,[a]\right]_y &= \frac{1}{n\,v_y}\sum_{k=0}^{n-1}\sum_{x=0}^{n-1} v_x\,a_x\,z^{x\,k}\,z^{-y\,k}\\ &= \frac{1}{n}\sum_{k=0}^{n-1}\sum_{x=0}^{n-1} v_x\,\frac{1}{v_y}\,a_x\,z^{x\,k}\,z^{-y\,k}\\ &= \frac{1}{n}\sum_{x=0}^{n-1} v_x\,\frac{1}{v_y}\,a_x\,\delta_{x,y}\,n\\ &= a_y\end{aligned}$$

(cf. section 1.1). That $\mathcal{W}_v\left[\mathcal{W}_v^{-1}\,[a]\right]$ is also identity is apparent from the definitions.

Given an implemented FFT it is trivial to set up a weighted Fourier transform:

Code 1.22 (weighted transform) *Pseudo code for the discrete weighted Fourier transform*

```
procedure weighted_ft(a[], v[], n, is)
{
    for x:=0 to n-1
    {
        a[x] := a[x] * v[x]
    }
    fft(a[],n,is)
}
```

Inverse weighted transform is also easy:

Code 1.23 (inverse weighted transform) *Pseudo code for the inverse discrete weighted Fourier transform*

```
procedure inverse_weighted_ft(a[], v[], n, is)
{
    fft(a[],n,is)
    for x:=0 to n-1
    {
        a[x] := a[x] / v[x]
    }
}
```

is must be negative wrt. the forward transform.

[FXT: `weighted_fft` in file `weighted/weightedfft.cc`]

[FXT: `weighted_inverse_fft` in file `weighted/weightedfft.cc`]

Introducing a *weighted (cyclic) convolution* h_v by

$$\begin{aligned}h_v &= a \circledast_{\{v\}} b\\ &= \mathcal{W}_v^{-1}\left[\mathcal{W}_v\,[a]\;\mathcal{W}_v\,[b]\right]\end{aligned} \tag{1.83}$$

(cf. formula 1.73)

Then for the special case $v_x = V^x$ one has

$$h_v = h^{(0)} + V^n\,h^{(1)} \tag{1.84}$$

($h^{(0)}$ and $h^{(1)}$ were defined by formula 1.75). It is not hard to see why: Up to the final division by the weight sequence, the weighted convolution is just the cyclic convolution of the two weighted sequences, which is for the element with index τ equal to

$$\sum_{x+y\equiv\tau\,(\mathrm{mod}\;n)} (a_x\,V^x)\,(b_y\,V^y) = \sum_{x\le\tau} a_x\,b_{\tau-x}\,V^\tau + \sum_{x>\tau} a_x\,b_{n+\tau-x}\,V^{n+\tau} \tag{1.85}$$

Final division of this element (by V^τ) gives $h^{(0)} + V^n\,h^{(1)}$ as stated.

The cases when V^n is some root of unity are particularly interesting: For $V^n = \pm i = \pm\sqrt{-1}$ one gets the so called *right-angle convolution*:

$$h_v = h^{(0)} \mp i\,h^{(1)} \tag{1.86}$$

This gives a nice possibility to directly use complex FFTs for the computation of a linear (acycclic) convolution of two real sequences: for length-n sequences the elements of the linear convolution with indices $0, 1, ..., n-1$ are then found in the real part of the result, the elements $n, n+1, ..., 2\,n-1$ are the imaginary part. Choosing $V^n = -1$ leads to the *negacyclic convolution* (or skew circular convolution):

$$h_v = h^{(0)} - h^{(1)} \tag{1.87}$$

Cyclic, negacyclic and right-angle convolution can be understood as a polynomial product modulo $z^n - 1$, $z^n + 1$ and $z^n \pm i$, respectively (cf. [3]).

[FXT: `weighted_complex_auto_convolution` in file `weighted/weightedconv.cc`]

[FXT: `negacyclic_complex_auto_convolution` in file `weighted/weightedconv.cc`]

[FXT: `right_angle_complex_auto_convolution` in file `weighted/weightedconv.cc`]

1.13 Half cyclic convolution for half the price ?

The computation of $h^{(0)}$ from formula 1.75 (without computing $h^{(1)}$) is called *half cyclic convolution*. Clearly, one asks for less information than one gets from the acyclic convolution. One might hope to find an algorithm that computes $h^{(0)}$ and uses only half the memory compared to the linear convolution or that needs half the work, possibly both. It may be a surprise that no such algorithm seems to be known currently[14].

Here is a clumsy attempt to find $h^{(0)}$ alone: Use the weighted transform with the weight sequence $v_x = V^x$ where V^n is very small. Then $h^{(1)}$ will in the result be multiplied with a small number and we hope to make it almost disappear. Indeed, using $V^n = 1000$ for the cyclic self convolution of the sequence $\{1, 1, 1, 1\}$ (where for the linear self convolution $h^{(0)} = \{1, 2, 3, 4\}$ and $h^{(1)} = \{3, 2, 1, 0\}$) one gets $\{1.003, 2.002, 3.001, 4.000\}$. At least for integer sequences one could choose V^n (more than two times) bigger than biggest possible value in $h^{(1)}$ and use rounding to nearest integer to isolate $h^{(0)}$. Alas, even for modest sized arrays numerical overflow and underflow gives spurious results. Careful analysis shows that this idea leads to an algorithm far worse than simply using linear convolution.

1.14 Convolution using the MFA

With the weighted convolutions in mind we reformulate the matrix (self-) convolution algorithm (section 1.9):

1. Apply a FFT on each column.

[14] If you know one, tell me about it!

2. On each row apply the weighted convolution with $V^C = e^{2\pi ir/R} = 1^{r/R}$ where R is the total number of rows, $r = 0..R-1$ the index of the row, C the length of each row (or, equivalently the total number columns)
3. Apply a FFT on each column (of the transposed matrix).

First consider

The case $R = 2$

The cyclic auto convolution of the sequence x can be obtained by two half length convolutions (one cyclic, one negacyclic) of the sequences[15] $s := x^{(0/2)} + x^{(1/2)}$ and $d := x^{(0/2)} - x^{(1/2)}$ using the formula

$$x \circledast x = \frac{1}{2}\{s \circledast s + d \circledast_- d, \quad s \circledast s - d \circledast_- d\} \tag{1.88}$$

The equivalent formula for the cyclic convolution of two sequences x and y is

$$x \circledast y = \frac{1}{2}\{s_x \circledast s_y + d_x \circledast_- d_y, \quad s_x \circledast s_y - d_x \circledast_- d_y\} \tag{1.89}$$

where

$$\begin{aligned} s_x &:= x^{(0/2)} + x^{(1/2)} \\ d_x &:= x^{(0/2)} - x^{(1/2)} \\ s_y &:= y^{(0/2)} + y^{(1/2)} \\ d_y &:= y^{(0/2)} - y^{(1/2)} \end{aligned}$$

For the acyclic (or linear) convolution of sequences one can use the cyclic convolution of the zero padded sequences $z_x := \{x_0, x_1, ..., n_{n-1}, 0, 0, ..., 0\}$ (i.e. x with n zeros appended). Using formula 1.88 one gets for the two sequences x and y (with $s_x = d_x = x$, $s_y = d_y = y$):

$$x \circledast_{ac} y = z_x \circledast z_y = \frac{1}{2}\{x \circledast y + x \circledast_- y, \quad x \circledast y - x \circledast_- y\} \tag{1.90}$$

And for the acyclic auto convolution:

$$x \circledast_{ac} x = z \circledast z = \frac{1}{2}\{x \circledast x + x \circledast_- x, \quad x \circledast x - x \circledast_- x\} \tag{1.91}$$

The case $R = 3$

Let $\omega = \frac{1}{2}(1 + \sqrt{3})$ and define

$$\begin{aligned} A &:= x^{(0/3)} + x^{(1/3)} + x^{(2/3)} \\ B &:= x^{(0/3)} + \omega x^{(1/3)} + \omega^2 x^{(2/3)} \\ C &:= x^{(0/3)} + \omega^2 x^{(1/3)} + \omega x^{(2/3)} \end{aligned}$$

Then, if $h := x \circledast_{ac} x$, there is

$$\begin{aligned} x^{(0/3)} &= A \circledast A + B \circledast_{\{\omega\}} B + C \circledast_{\{\omega^2\}} C \\ x^{(1/3)} &= A \circledast A + \omega^2 (B \circledast_{\{\omega\}} B) + \omega (C \circledast_{\{\omega^2\}} C) \\ x^{(2/3)} &= A \circledast A + \omega (B \circledast_{\{\omega\}} B) + \omega^2 (C \circledast_{\{\omega^2\}} C) \end{aligned} \tag{1.92}$$

For real valued data C is the complex conjugate of B and (with $\omega^2 = cc.\omega$) $B \circledast_{\{\omega\}} B$ is the cc. of $C \circledast_{\{\omega^2\}} C$ and therefore every $B \circledast_{\{\}} B$-term is the cc. of the $C \circledast_{\{\}} C$-term in the same line. Is there a nice and general scheme for real valued convolutions based on the MFA? Read on for the positive answer.

[15] s, d lower half plus/minus higher half of x

1.15 Convolution of real valued data using the MFA

For row 0 (which is real after the column FFTs) one needs to compute the (usual) cyclic convolution; for row $R/2$ (also real after the column FFTs) a negacyclic convolution is needed[16]. the pseudo code for that task is given on page 58.

All other weighted convolutions involve complex computations, but it is easy to see how cut the work by 50 percent: As the result must be real the data in row number $R - r$ must, because of the symmetries of the real and imaginary part of the (inverse) Fourier transform of real data, be the complex conjugate of the data in row r. Therefore one can use real FFTs (R2CFTs) for all column-transforms for step 1 and half-complex to real FFTs (C2RFTs) for step 3.

Let the computational cost of a cyclic (real) convolution be q, then

For R even one must perform 1 cyclic (row 0), 1 negacyclic (row $R/2$) and $R/2 - 2$ complex (weighted) convolutions (rows $1, 2, ..., R/2 - 1$)

For R odd one must perform 1 cyclic (row 0) and $(R - 1)/2$ complex (weighted) convolutions (rows $1, 2, ..., (R - 1)/2$)

Now assume, slightly simplifying, that the cyclic and the negacyclic real convolution involve the same number of computations and that the cost of a weighted complex convolution is twice as high. Then in both cases above the total work is exactly half of that for the complex case, which is about what one would expect from a real world real valued convolution algorithm.

For acyclic convolution one may want to use the right angle convolution (and complex FFTs in the column passes).

1.16 Convolution without transposition using MFA

An algorithm for convolution using the MFA that uses revbin_permute instead of transpose (works for sizes that are a power of two, generalizes for sizes a power of some prime):

```
rows=8  columns=4
input data (symbolic format: ROOC):
  0:     0     1     2     3
  1:  1000  1001  1002  1003
  2:  2000  2001  2002  2003
  3:  3000  3001  3002  3003
  4:  4000  4001  4002  4003
  5:  5000  5001  5002  5003
  6:  6000  6001  6002  6003
  7:  7000  7001  7002  7003

FULL REVBIN_PERMUTE for transposition:
  0:     0  4000  2000  6000  1000  5000  3000  7000
  1:     2  4002  2002  6002  1002  5002  3002  7002
  2:     1  4001  2001  6001  1001  5001  3001  7001
  3:     3  4003  2003  6003  1003  5003  3003  7003

DIT FFTs on revbin_permuted rows (in revbin_permuted sequence), i.e. unrevbin_permute rows:
 (apply weight after each FFT)
  0:     0  1000  2000  3000  4000  5000  6000  7000
  1:     2  1002  2002  3002  4002  5002  6002  7002
  2:     1  1001  2001  3001  4001  5001  6001  7001
  3:     3  1003  2003  3003  4003  5003  6003  7003
```

[16] For R odd there is no such row and no negacyclic convolution is needed.

```
FULL REVBIN_PERMUTE for transposition:
  0:       0     1     2     3
  1:    4000  4001  4002  4003
  2:    2000  2001  2002  2003
  3:    6000  6001  6002  6003
  4:    1000  1001  1002  1003
  5:    5000  5001  5002  5003
  6:    3000  3001  3002  3003
  7:    7000  7001  7002  7003

CONVOLUTIONS on rows (don't care revbin_permuted sequence), no reordering.

FULL REVBIN_PERMUTE for transposition:
  0:       0  1000  2000  3000  4000  5000  6000  7000
  1:       2  1002  2002  3002  4002  5002  6002  7002
  2:       1  1001  2001  3001  4001  5001  6001  7001
  3:       3  1003  2003  3003  4003  5003  6003  7003

 (apply inverse weight before each FFT)
DIF FFTs on rows (in revbin_permuted sequence), i.e. revbin_permute rows:
  0:       0  4000  2000  6000  1000  5000  3000  7000
  1:       2  4002  2002  6002  1002  5002  3002  7002
  2:       1  4001  2001  6001  1001  5001  3001  7001
  3:       3  4003  2003  6003  1003  5003  3003  7003

FULL REVBIN_PERMUTE for transposition:
  0:       0     1     2     3
  1:    1000  1001  1002  1003
  2:    2000  2001  2002  2003
  3:    3000  3001  3002  3003
  4:    4000  4001  4002  4003
  5:    5000  5001  5002  5003
  6:    6000  6001  6002  6003
  7:    7000  7001  7002  7003
```

1.17 Split radix Fourier transforms (SRFT)

Code 1.24 (split radix DIF FFT) *Pseudo code for the split radix DIF algorithm,* is *must be -1 or +1:*

```
procedure fft_splitradix_dif(x[],y[],ldn,is)
{
    n := 2**ldn
    if n<=1  return
    n2 := 2*n
    for k:=1 to ldn
    {
        n2 := n2 / 2
        n4 := n2 / 4
        e := 2 * PI / n2
        for j:=0 to n4-1
        {
            a := j * e
            cc1 := cos(a)
            ss1 := sin(a)
            cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
```



```
            ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)
            ix := j
            id := 2*n2
            while ix<n-1
            {
                i0 := ix
                while i0 < n
                {
                    i1 := i0 + n4
                    i2 := i1 + n4
                    i3 := i2 + n4
                    {x[i0], r1} := {x[i0] + x[i2], x[i0] - x[i2]}
                    {x[i1], r2} := {x[i1] + x[i3], x[i1] - x[i3]}
                    {y[i0], s1} := {y[i0] + y[i2], y[i0] - y[i2]}
                    {y[i1], s2} := {y[i1] + y[i3], y[i1] - y[i3]}
                    {r1, s3} := {r1+s2, r1-s2}
                    {r2, s2} := {r2+s1, r2-s1}
                    // complex mult: (x[i2],y[i2]) := -(s2,r1) * (ss1,cc1)
                    x[i2] :=  r1*cc1 - s2*ss1
                    y[i2] := -s2*cc1 - r1*ss1
                    // complex mult: (y[i3],x[i3]) := (r2,s3) * (cc3,ss3)
                    x[i3] :=  s3*cc3 + r2*ss3
                    y[i3] :=  r2*cc3 - s3*ss3
                    i0 := i0 + id
                }
                ix := 2 * id - n2 + j
                id := 4 * id
            }
        }
    }
    ix := 1
    id := 4
    while ix<n
    {
        for i0:=ix-1 to n-id step id
        {
            i1 := i0 + 1
            {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}
            {y[i0], y[i1]} := {y[i0]+y[i1], y[i0]-y[i1]}
        }
        ix := 2 * id - 1
        id := 4 * id
    }
    revbin_permute(x[],n)
    revbin_permute(y[],n)
    if is>0
    {
        for j:=1 to n/2-1
        {
            swap(x[j],x[n-j])
            swap(y[j],y[n-j])
        }
    }
}
```

[FXT: split_radix_fft in file fft/fftsplitradix.cc] [FXT: split_radix_fft in file fft/cfftsplitradix.cc]

1.17.1 Real to complex SRFT

Code 1.25 (split radix R2CFT) *Pseudo code for the split radix R2CFT algorithm*

```
procedure r2cft_splitradix_dit(x[],ldn)
{
    n := 2**ldn
    ix := 1;
    id := 4;
```

```
do
{
    i0 := ix-1
    while i0<n
    {
        i1 := i0 + 1

        {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}

        i0 := i0 + id
    }

    ix := 2*id-1
    id := 4 * id
}
while ix<n

n2 := 2
nn := n/4
while nn!=0
{
    ix := 0
    n2 := 2*n2
    id := 2*n2
    n4 := n2/4
    n8 := n2/8

    do  // ix loop
    {
        i0 := ix
        while i0<n
        {
            i1 := i0
            i2 := i1 + n4
            i3 := i2 + n4
            i4 := i3 + n4

            {t1, x[i4]} := {x[i4]+x[i3], x[i4]-x[i3]}

            {x[i1], x[i3]} := {x[i1]+t1, x[i1]-t1}

            if  n4!=1
            {
                i1 := i1 + n8
                i2 := i2 + n8
                i3 := i3 + n8
                i4 := i4 + n8

                t1 := (x[i3]+x[i4]) * sqrt(1/2)
                t2 := (x[i3]-x[i4]) * sqrt(1/2)

                {x[i4], x[i3]} := {x[i2]-t1, -x[i2]-t1}
                {x[i1], x[i2]} := {x[i1]+t2,  x[i1]-t2}
            }

            i0 := i0 + id
        }

        ix := 2*id - n2
        id := 2*id
    }
    while ix<n

    e := 2.0*PI/n2
    a := e

    for j:=2 to n8
    {
        cc1 := cos(a)
        ss1 := sin(a)
        cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
        ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)

        a := j*e

        ix := 0
        id := 2*n2

        do  // ix-loop
        {
            i0 := ix
            while  i0<n
            {
                i1 := i0 + j - 1
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                i5 := i0 + n4 - j + 1
                i6 := i5 + n4
                i7 := i6 + n4
                i8 := i7 + n4
```



```
                // complex mult: (t2,t1) := (x[i7],x[i3]) * (cc1,ss1)
                t1 := x[i3]*cc1 + x[i7]*ss1
                t2 := x[i7]*cc1 - x[i3]*ss1

                // complex mult: (t4,t3) := (x[i8],x[i4]) * (cc3,ss3)
                t3 := x[i4]*cc3 + x[i8]*ss3
                t4 := x[i8]*cc3 - x[i4]*ss3

                t5 := t1 + t3
                t6 := t2 + t4
                t3 := t1 - t3
                t4 := t2 - t4

                {t2, x[i3]} := {t6+x[i6], t6-x[i6]}

                x[i8] := t2

                {t2,x[i7]} := {x[i2]-t3, -x[i2]-t3}

                x[i4] := t2

                {t1, x[i6]} := {x[i1]+t5, x[i1]-t5}

                x[i1] := t1

                {t1, x[i5]} := {x[i5]+t4, x[i5]-t4}

                x[i2] := t1

                i0 := i0 + id
            }

            ix := 2*id - n2
            id := 2*id

        }
        while ix<n
    }
    nn := nn/2
  }
}
```

1.17.2 Complex to real SRFT

Code 1.26 (split radix C2RFT) *Pseudo code for the split radix C2RFT algorithm*

```
procedure c2rft_splitradix_dif(x[],ldn)
{
    n := 2**ldn

    n2 := n/2
    nn := n/4
    while nn!=0
    {
        ix := 0
        id := n2
        n2 := n2/2
        n4 := n2/4
        n8 := n2/8

        do  // ix loop
        {
            i0 := ix
            while i0<n
            {
                i1 := i0
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                {x[i1], t1} := {x[i1]+x[i3], x[i1]-x[i3]}

                x[i2] := 2*x[i2]

                x[i4] := 2*x[i4]
                {x[i3], x[i4]} := {t1+x[i4], t1-x[i4]}

                if  n4!=1
                {
                    i1 := i1 + n8
                    i2 := i2 + n8
                    i3 := i3 + n8
                    i4 := i4 + n8

                    {x[i1], t1} := {x[i2]+x[i1], x[i2]-x[i1]}
                    {t2, x[i2]} := {x[i4]+x[i3], x[i4]-x[i3]}

                    x[i3] := -sqrt(2)*(t2+t1)
```

```
                x[i4] :=  sqrt(2)*(t1-t2)
            }
            i0 := i0 + id
        }
        ix := 2*id - n2
        id := 2*id
    }
    while ix<n

    e := 2.0*PI/n2
    a := e
    for j:=2 to n8
    {
        cc1 := cos(a)
        ss1 := sin(a)
        cc3 := cos(3*a)  // == 4*cc1*(cc1*cc1-0.75)
        ss3 := sin(3*a)  // == 4*ss1*(0.75-ss1*ss1)
        a := j*e

        ix := 0
        id := 2*n2

        do  // ix-loop
        {
            i0 := ix
            while  i0<n
            {
                i1 := i0 + j - 1
                i2 := i1 + n4
                i3 := i2 + n4
                i4 := i3 + n4

                i5 := i0 + n4 - j + 1
                i6 := i5 + n4
                i7 := i6 + n4
                i8 := i7 + n4

                {x[i1], t1} := {x[i1]+x[i6], x[i1]-x[i6]}

                {x[i5], t2} := {x[i5]+x[i2], x[i5]-x[i2]}

                {t3, x[i6]} := {x[i8]+x[i3], x[i8]-x[i3]}

                {t4, x[i2]} := {x[i4]+x[i7], x[i4]-x[i7]}

                {t1, t5} := {t1+t4, t1-t4}

                {t2, t4} := {t2+t3, t2-t3}

                // complex mult: (x[i7],x[i3]) := (t5,t4)  * (ss1,cc1)
                x[i3] := t5*cc1 + t4*ss1
                x[i7] := -t4*cc1 + t5*ss1

                // complex mult: (x[i4],x[i8]) := (t1,t2)  * (cc3,ss3)
                x[i4] := t1*cc3 - t2*ss3
                x[i8] := t2*cc3 + t1*ss3

                i0 := i0 + id
            }
            ix := 2*id - n2
            id := 2*id
        }
        while ix<n
    }

    nn := nn/2
}

ix := 1;
id := 4;
do
{
    i0 := ix-1
    while i0<n
    {
        i1 := i0 + 1

        {x[i0], x[i1]} := {x[i0]+x[i1], x[i0]-x[i1]}

        i0 := i0 + id
    }
    ix := 2*id-1
    id := 4 * id
}
while ix<n
}
```


1.18 Multidimensional FTs

1.18.1 Definition

Let $a_{x,y}$ ($x = 0,1,2,..,C$ and $y = 0,1,2,...,R$) be a 2-dimensional array of data[17]. Its 2-dimensional Fourier transform $c_{k,h}$ is defined by:

$$c = \mathcal{F}[a] \tag{1.93}$$

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} \sum_{x=0}^{R-1} a_{x,y}\, z^{x\,k+y\,h} \qquad \text{where} \quad z = e^{\pm 2\,\pi\, i/n}, \quad n = R\,C \tag{1.94}$$

Its inverse is

$$a = \mathcal{F}^{-1}[c] \tag{1.95}$$

$$a_{x} = \frac{1}{\sqrt{n}} \sum_{k=0}^{C-1} \sum_{h=0}^{R-1} c_{k,h}\, z^{-(x\,k+y\,h)} \tag{1.96}$$

For a m-dimensional array $a_{\vec{x}}$ ($\vec{x} = (x_1, x_2, x_3, ..., x_m)$, $x_i \in 0,1,2,...,S_i$) the m-dimensional Fourier transform $c_{\vec{k}}$ ($\vec{k} = (k_1, k_2, k_3, ..., k_m)$, $k_i \in 0,1,2,...,S_i$) is defined as

$$c_{\vec{k}} := \frac{1}{\sqrt{n}} \sum_{x_1=0}^{S_1-1} \sum_{x_2=0}^{S_2-1} \cdots \sum_{x_m=0}^{S_m-1} a_{\vec{x}}\, z^{\vec{x}.\vec{k}} \tag{1.97}$$

$$\text{where} \quad z = e^{\pm 2\,\pi\, i/n}, \quad n = S_1\, S_2 \ldots Sm$$

The inverse transform is again the one with the minus in the exponent of z.

1.18.2 The row column algorithm

The equation of the definition of the two dimensional FT (1.93) can be recast as

$$c_{k,h} := \frac{1}{\sqrt{n}} \sum_{x=0}^{C-1} z^{x\,k} \sum_{x=0}^{R-1} a_{x,y}\, z^{y\,h} \tag{1.98}$$

which shows that the 2-dimensional FT can be accomplished by using 1-dimensional FTs to transform first the rows and then the columns[18]. This leads us directly to the row column algorithm:

Code 1.27 (row column FFT) *Compute the two dimensional FT of* `a[][]` *using the row column method*

```
procedure rowcol_ft(a[][],R,C)
{
    complex a[R][C]  // R (length-C) rows, C (length-R) columns
    for r:=0 to R-1  // FFT rows
    {
        fft(a[r][],C,is)
    }
    complex t[R]     // scratch array for columns
    for c:=0 to C-1  // FFT columns
    {
        copy a[0,1,...,R-1][c] to t[]  // get column
        fft(t[],R,is)
        copy t[] to a[0,1,...,R-1][c]  // write back column
    }
}
```

[17] Imagine a $R \times C$ matrix of R rows (of length C) and C columns (of length R).
[18] or first the rows, then the columns, the result is the same

Here it is assumed that the rows lie in contiguous memory (as in the C language).

Transposing the array before the column pass in order to avoid the copying of the columns to extra scratch space will probably do good for the performance. The transposing back before returning can be avoided if a backtransform will follow[19], the backtransform must then be called with R and C swapped.

[19] as typical for convolution etc.

E Crash Course C for Embedded Systems

By W.M. van Oijen, Nov 2005, Edited by X. van Rijnsoever, Sept 2006

This *Crash Course C for Embedded Systems* is intented for students that are doing the Embedded Systems Lab but have never programmed in C. It is certainly not a complete course covering all the important subjects, but it aims to cover enough of the topics to be able to complete this lab succesfully. Furthermore it includes certain topics that are especially important for the 8051 platform used in the lab. I hope this document will also be a good refresher for students that have experience with C and that it will contain useful tips to write better or faster code. Because most of the students currently doing this course have experience with Java, I will make comparisons to that language at some places. For people that want to know more about C, or want to have a good book about it, I highly recommend "The C Programming Language" of Brian W. Kernighan and Dennis M. Ritchie, the makers of C. This book is available at the ETV and is not very expensive. It's a good learning book and a good reference.

E.1 Hello, world!

The traditional way of learning a new programming language is to write the famous "hello, world" program. The purpose of this program is to write the text "hello, world" to the console. In C such a program would look like this:

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

There are three important things that can be noticed in above program:

1. The program starts with "**#include <stdio.h>**". This is to tell the compiler which functions and definitions to include in the program. It's comparable with the "import java.io.*'; statement in Java programs.
2. The function **main()** which must always exist in every C program. This is the function that will be called when your program is executed. Unlike in Java, this program does not have to be a part of a class. This is because C is not a object-oriented programming language.
3. The function **printf()** is used for outputting text to the console. Although there are more functions available for printing text, this one is the most used in C because it's a very powerful routine that can print and format all standard data types in an easy way.

These topics will be discussed more extensively in the following chapters, but for now we give another example:

```
#include <stdio.h>          /* contains printf */

int foo (int bar)           /* function returns an int and has
                               an int as parameter */
{
     if (bar < 10)
          return 1;
     else
          return 0;
}
```

```
int main (void)
{
      int a = 10;

      if (foo (a))          // functioncall
            printf ("a < 10\n");
      else
            printf ("a >= 10\n");

      return 0;
}
```

Notice the following things in this example program:

- Comments are placed inside a `/* … */` block. The Keil compiler also accepts C++ style single-line comments `// …`
- `foo` is a function that returns an int and has an int as parameter.
- The result of the `foo` function is used as the condition of an `if`-statement. This is because C doesn't have a special boolean type. Instead an `int` with a value 0 is used as `false`, while all other values are interpreted as `true`.
- The variable `a` is declared as an `int` and initialized

E.2 Variables and functions

In C both variables and functions need to be declared before they can be used. A variable can only be declared at the beginning of a block { … }. A variable declaration looks like this:

```
[specifier] [data type] variable_name [= init_value];
```

The `specifier` part contains information on the use or location of this variable. The `data type` specifies the type of information the variable can hold. The `init_value` contains the option startvalue of the variable.
A function declaration looks like this:

```
[specifier] [return data type] function_name (parameter list);
```

Both variables and functions are considered declared when they are defined before they are used. One special function always needs to be present: `int main ()`. This function locates the starting point of the program.

E.2.1 Specifiers

A variable can be given the following specifiers:

Specifier	Effect
const	This is a constant value
extern	The variable is defined elsewhere
static	The variable keeps its value in between functioncalls
volatile	The variable can be changed due to external factors. This is useful for variables that are changed by interrup routines or external hardware

A function can be given these specifiers:

Specifier	Effect
extern	The function is defined elsewhere
static	The function is only accessable from within this file

E.2.2 Data types

Data types in C are defined less strict than in most other programming languages. For inexperienced programmers this might be very confusing. But with a few simple rules it's easy to write software that is portable across different platforms, and as efficient as possible without completely rewriting the code for a specific platform.
The confusing part is the width of the integer data types. The rules are as follows. A **char** is always 8 bits (one byte). A **short**, **int** or **long** can be 16 or 32 bits, depending on the platform. If you use the Keil compiler for the 8051 platform, the size of **int** is 16 bits and **long** is 32 bits. If you want to run your code also on other platforms (e.g. if you want to test some functions at home), you should carefully consider the following rules:

- **shorts** and **ints** are at least 16 bits
- **longs** are at least 32 bits
- **short** is no longer than **int**, which is no longer than **long**

Here are some guidelines for writing fast and portable code:

Data type	Application
void	use **void** for function without a return value, an empty parameter list and, when used as **void***, as a generic pointer.
(signed) char	use **(signed) char** when the result fits in 8 bits, and memory usage is more important than speed.
short	use **short** when the result fits in 16 bits, and memory usage is more important than speed.
int	use **int** when the result fits in 16 bits, and speed is more important than memory usage.
long	use **long** when the result does not fit in 16 bits, but does fit in 32 bits.
float	use **float** only when floating point is required

You also have to realize that the 8051 is an 8-bit processor, meaning that 32-bit operations are divided into lots of 8-bit operations. So in some cases it might be faster to use smaller data types like **char** instead of **int**. There is no easy rule to find out which is faster. You'll have to try it out and analyze the results if you want the fastest code possible.

E.2.2.1 Arrays

Arrays in C are defined as follows:

```
char text[6];
```

This defines an array of 6 characters. It can hold a string of 5 characters, because the terminating zero takes one place. It is possible to initialize arrays:

```
char text[6] = "hello";
char text[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char text[] = "hello";
```

These three examples have the same effect. If no size is specified, the compiler determines the size of the string. The following is an error:

```
char text[3] = "hello"; /* error: not enough memory reserved */
```

NOTE

Arrays in C have no boundary checking. Therefore it is very important that you make sure that index does not exceed its maximum value. If you do so, memory corruption will occur and your program will probably crash.

E.2.2.2 Structs

Structures (**structs**) provide an easy way to group variables that are related, like different properties of a file on a filesystem (name, size, attributes, etc.), or the real and imaginary part of a complex number. The latter is what you should use in the Embedded Systems lab. A complex number can be defined like this:

```
struct complex
{
      double real;     /* real part */
      double imag;     /* imaginary part */
} x, y;
```

This statement defines the type **complex** and two variables **x** and **y** of that type. It is also possible to define the type without defining one or more variables of that type. This is what you would normally do in header files. Just leave the variable names away. But do not forget the semicolon in that case!

```
struct complex
{
      double real;     /* real part */
      double imag;     /* imaginary part */
};                     /* yes, a semicolon!! */
```

By the way, it is not necessary to specify a name for the type (**complex** in this case). But it certainly makes live easier for you, because it allows you to refer to the type by its name. For example in the declaration of a function that has two complex variables as arguments:

```
int equal(struct { double real; double imag; } x,
      struct { double real; double image; } y);
```

can be written as:

```
int equal(struct complex x, struct complex y);
```

which is much better readable. If you think this is still too much work, then read on.

In C there is a **typedef** operator which let you define types. Its syntax is:

```
typedef existing_type new_name;
```

for example:

```
typedef int foo;
```

After this definition it is possible to use the word **foo** as a type:

```
foo x = 3, y;
y = x*x;
```

This is especially useful for structs, because it allows you to write this:

```
typedef struct
{
      double real;     /* real part */
      double imag;     /* imaginary part */
} complex;
complex x, y;

int equal(complex x, complex y);
```

But note that the name of the type is given *after* the definition {...} in this case! It is not possible to define variables of the new type in the same statement like in the first example.

Of course, after you have defined the struct you also want to store values in it. This can be done in two different ways. Firstly, the members of the struct can be assigned values individually:

```
x.real = 3.0;
x.imag = 4.0;
```

Secondly, it is possible to **initialize** structs during the definition:

```
complex x = { 3.0, 4.0 };
```

Note that an **initialization** is not the same as an **assignment**, although their syntaxes look the same. It is not possible use the braces notation in an assignment like this:

```
complex x;
x = { 3.0, 4.0 };          /* error, not allowed */
```

However, it is possible to copy structs just like other data types:

```
complex x;                 /* definition                    */
complex y = { 3.0, 4.0 };  /* definition w/ initialization */
x = y;                     /* assignment                    */
```

E.2.2.3 Pointers

Pointers are considered to be a very difficult topic. Some programming languages, like Java, do not have pointers for that reason. But pointers are essential in C. It's almost impossible to write a useful C-program without using pointers. For example, strings are of type **char *** (pointer to char), so to print something you have to use pointers already.
One of the advantages of pointers is that it gives the programmer very much control over his program. Because pointers are much closer to the hardware than for example objects in Java, it is possible to optimize algorithms for efficient (=fast) execution. But this power comes at a price. [And as each Electrical Engineering student should know: "*With great power comes great responsibility.*"] It is the responsibility of the programmer to access only those memory locations that he is allowed to access. In other words, pointers should always point to a valid memory location. It is a good programming habit to always initialize pointers to **NULL**, and to always check for NULL-pointers before using them.

In general, the rules for pointers are very simple. But it takes some time to get used to the concept. Here are some of the rules you should understand before using pointers:

- A pointer contains an address.

- The address of a variable can be obtained by the **&**-operator: **&x** is the address of variable x.
- The character ***** in a definition denotes a pointer: **int *p;** means that **p** is a pointer to an int.
- The character ***** in an expression means dereference. In that case the pointer is "followed" to the location it points to. If **x** contains the value 3, and **int *p = &x;** then **p** contains the address of **x** and ***p** contains the value 3.

One of the useful features of pointers is in parameter passing. In C, function parameters are passed using the call-by-value parameter-transfer-mechanism. This means that it isn't possible to change values of calling variables of the function. For example:

```
void swap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}

int x=1, y=2;
swap(x, y);
```

The calling of the function swap(x,y) will have no effect on x and y, x=1 and y =2. We can also make a swap function by passing the variables a and b using pointers.

```
void swap(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}

int x=1, y = 2;
swap(&x, &y);
```

This will have the desired effect on x and y: now x=2 and y =1.

E.3 Expressions and operators

E.3.1 Assignments as expressions

One of the special features of C is the fact that an assignment doubles as an expression. It has the value of the assignment. This means that it is possible to use the following assignment structure:

```
a = b = c = 1;
```

The result is that all these variables are assigned the value 1. One of the things that can lead to difficult to find errors is that since an assignment is also an expression, its value can be used in the conditional part of an `if`-statement:

```
if (a = 10)
      …
else
```

...

Since 10 is true, the else part will never execute. Of course the = should be replaced with the compare operator == to prevent this.

E.3.2 Operators

The C language has a large number of operators, they are ordered here according to there working. This list is not complete, but contains the operators you will most likely need for the SpectrA assignment.

E.3.2.1 Arithmetic operators

Operator	Effect
+ (unary)	Parameter is positive
+ (binary)	Addition
++ (prefix)	Evaluate parameter = parameter + 1 beforehand
++ (postfix)	Evaluate parameter = parameter + 1 afterwards
- (unary)	Negate parameter
- (binary)	Subtraction
-- (prefix)	Evaluate parameter = parameter - 1 beforehand
-- (postfix)	Evaluate parameter = parameter - 1 afterwards
/	Division
*	Multiplication
%	(integer only) modulo operation

E.3.2.2 Logic operators

The logic operators of C work on logic values, meaning they interpret a value 0 as false and a non-zero value as true. In addition, the binary logic operators implement so-called lazy evaluation. This means that evaluation is stopped when the outcome can't change by evaluating more information.
For the logic and (&&) this means that the evaluation stops when a false part of the expression is found, the outcome will always be false, no matter how many additional information is provided. For the logic or (||) this means that the evaluation will stop on the first true.

Operator	Effect
!	Logic inversion (true -> false, false -> true)
&&	Logic and (lazy evaluation)
||	Logic or (lazy evaluation)

E.3.2.3 Bitwise operators

Bitwise operators operate on all the bits of their parameters.

Operator	Effect
~	Bitwise inversion
&	Bitwise and
^	Bitwise xor
|	Bitwise or

>>	Bitwise shift right
<<	Bitwise shift left

E.3.2.4 Compare operators

Operator	Effect
>	Greater than
>=	Greater than or equal
==	Equal to
!=	Unequal to
<	Smaller than
<=	Smaller than or equal

E.4 Input / Ouput

The C language comes with powerfull routines for input and output. These can be very useful for debugging your SpectrA implementation.

E.4.1 Output with printf

As written above and shown in the first two examples, the **printf** function is a powerful routine that displays formatted text. Its usage is very easy once you've got to know it. The formal declaration of **printf** is the following:

```
int printf (const char *fmt, ...);
```

The function returns the number of characters printed. For this course you will most likely never use this, so you may ignore it.

E.4.1.1 The format string

The first argument, `fmt`, is the **format string**. This string can contain normal characters, which will be copied to the output stream, and **conversion specifications**, which tell the program to fetch and print the next argument from the list. The format string is of type **char *** which is a standard type for strings in C. The keyword **const** implies that the format string is not modified by the printf function.

E.4.1.2 Conversion specifications

The **conversion specification** always starts with the character **%** and ends with a character specifying the type of the argument that should be printed. Between those characters some optional formatting options can be included. There are many of them, but these are not very useful for the Embedded Systems course. They will not be discussed in this document. If you want to know these options, read [KR], the documentation of your compiler or the man page of printf (do a google search for "man printf").
A list of conversion specifications you might use is listed in the following table.

Specifier	Effect
%d / %i	**int**; decimal number
%ld / %li	**long int;** long decimal number
%x / %X	**(unsigned) int**; hexadecimal number, using lower / uppercase letters
%u	**unsigned int**; unsigned decimal number
%lu	**unsigned long int;** unsigned long decimal number
%c	**char**; single character
%s	**char ***; zero-terminated string

%f	**double**; floating-point number, in the format [-]*m.dddddd*
%e / %E	**double**; floating-point number, in the format [-]*m.dddddd* e / E ±*xx*
%p	**void ***; pointer, prints the address in an implementation-dependent representation
%%	prints a % character

E.4.1.3 Variable-length argument lists

The last argument is "**...**", which specifies that zero or more arguments may be following the format string. This concept of **variable argument lists** is not known in many programming languages. It is an advanced topic that will not be discussed in this course. It's only important to understand that it's a special case where the compiler does not know in advance how many parameters it can expect, and of what type. It is therefore very important that you are extra careful in using the correct conversion specifiers. If the specifiers do not match the type of the argument, the compiler will (in general) not be able to detect your mistake, and the results will be undefined.

If you are using **printf** and get unexpected results (for instance very large numbers when you know for sure that the numbers are much smaller, or binary data printed to the console), then check if the conversion specifiers match the data types of the arguments. If you're unsure about the type of your data, use explicit type casts to make sure! (type casts will be discussed further in this document.)

E.4.1.4 Example

```
char ch = 'A';
printf("The value of ASCII character '%c' is %d\n", ch, (int)
ch);
/* prints the following line:
 * The value of ASCII character 'A' is 65. */
```

E.4.1.5 Memory restriction

NOTE

Normally, function arguments in C are pushed onto the stack but the Keil compiler uses a different approach. For most functions the arguments are passed using registers or fixed memory locations. Because it is not known in advance how many arguments will be passed to a function with a variable-length argument list, a fixed amount of memory is reserved. By default this is 15 bytes. The consequence of this is that **you should take care that no more than 15 bytes are passed to the printf function**. Please keep in mind that the format string already takes up three bytes. So there are only 12 bytes left for the other arguments. This is enough for 3 longs, or 6 ints, or 2 longs and 2 ints, and so on. If you pass more arguments the results are undefined.

E.4.2 Other output functions

Besides the printf functions, C has other functions output functions. These functions aren't as multifunctional as the printf function, but in return they are smaller. For this lab the following functions may be useful

Function	Effect
`putchar (char c)`	Outputs char `c` to stdout
`puts (const char *s)`	Outputs string `s` to stdout

E.4.3 Input with scanf

The input counterpart of the `printf` function is the `scanf` function. The formal declaration of **scanf** is the following:

```
int scanf (const char *fmt, ...);
```

The function returns the number of succesfully read elements. The `fmt` string specifies the format of the input data. The format specifiers are the same as the `printf` format specifiers. The additional arguments must be pointers to locations for storing the data.

E.4.3.1 Example

```
int number;
scanf("%d", &number);
/* note the '&' since we need the address of number */
```

NOTE

Be careful with the scanf function, the function doesn't allocate memory for storing the input data and doesn't have a strict checking of the input data. Inputting a character instead of a number can lead to serious problems with your program.

E.4.4 Other input functions

The C language also has several other input functions besides the `scanf` function. For this lab only one is useful:

Function	Effect
`int getchar (void)`	Read char from stdin

E.5 The Preprocessor

One of the differences between C and Java is that C has a preprocessor. Its purpose is "preprocess" source files before they are being compiled. Although you are probably not interested in the compilation process, it is important to know how a little bit of the compilation process. Besides that, there are some useful features that can reduce your development and/or debugging time if used properly. Commands that are meant for the preprocessor start

with the character # and are called **preprocessor directives**. The most important ones will be discussed.

E.5.1 #include

The purpose of this directive is to include header files into the source files. The result might be seen as copying the complete contents of the file into your source file, at the position of the include directive. It is possible to include other files (for example .c files) as well, but this is considered bad programming behaviour, and will most likely lead to compiler errors that are difficult to explain. You have been warned!
The syntax of include directives can be one of the following:

```
#include <stdio.h>
#include "myheader.h"
```

In the first case the files are searched in the library path, so the default headers (like stdio.h) can be found. In the second case (using double quotes) the files are searched in the working directory. You should use this for header files that are not in the library path, but copied into your project directory.

E.5.2 #define

This directive is very useful for programmers. It is used to define a constant. Before compilation all the constants are replaced with their value. So if you write this:

```
#define N 42
printf("The magic number is %d\n", N);
```

the compiler will actually read this as:

```
printf("The magic number is %d\n", 42);
```

This means that there is no variable N in the executable code that will require a storage location. Therefore it's faster and requires less memory than when using a variable. However, it's main purpose is to keep code readable and to prevent "magic numbers" in the code. It also keeps code maintainable. Consider the following example:

```
#define BUFSIZE 1024
char buf[BUFSIZE];
main()
{
      /* initialize array */
      for (i = 0; i < BUFSIZE; i++)
      {
            buf[i] = 0;
      }
}
```

If you decide to change the size of the buffer you only have to modify the constant BUFSIZE. Without using constants you would have to modify the value everywhere in the code. In case you would have forgotten to modify the number in the for-loop, your program would not have worked correctly anymore.
This directive is also used to define constants that can be checked for using the **ifdef** or **ifndef** directive.

E.5.3 #ifdef, #ifndef

Another very easy but powerful directive is the **ifdef**. It is used for conditional compilation. That means that a part of the code is only compiled (and thus included in your final program) if a certain constant is defined. This can be used for writing code for different platforms, or for adding debugging code. An #ifdef block should be closed with #endif. It's also possible to use #else. A few example will best explain how this works. The following examples are rather self-explaining.

Example 1:

```
#define DEBUG 1
char ch;

ch = getch();
#ifdef DEBUG
     printf("Received character '%s'\n", ch);
#endif
/* do something with the character */
```

Example 2:

```
#ifdef MEMORY_SAVING_ON
     #define mytype char
#else
     #define mytype int
#endif
```

Another very important use of #ifndef is to prevent header files from being included multiple times in the source code:

```
#ifndef _SOME_HEADER_
#define _SOME_HEADER_

/* define some stuff */

#endif
```

NOTE

Use **#ifdef** or **#ifndef** to enable/disable certain parts of your code, like test functions or debug code. In this way you don't have to include all code in your main project (you will likely run out of memory soon if you do), but you will still be able to test or debug your code by changing one line of code.

E.6 References

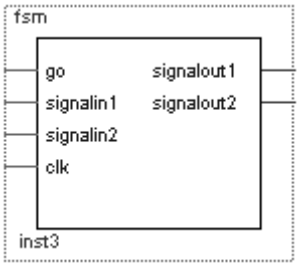
[KR] Brian W. Kernighan & Dennis M. Ritchie.*The C Programming Language*. Second Edition. Prentice Hall, Inc. 1988.

F Appendix F: Hardware components

This appendix presents a number of hardware components that are available for the design of the hardware accelerators. Most of these components are part of the Quartus component library. The others are provided as VHDL files during the lab. Besides the components that are stated below it is allowed to use all of the components that are part of the standard library of Quartus.

FSM

Block scheme:



Function : FSM (control)

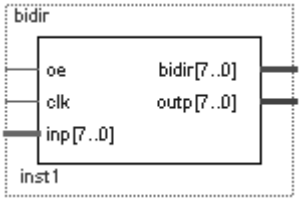
The inputs and the outputs of the FSM may be changed.

VHDL Entity description:

```
ENTITY FSM IS
  port (
        go              : in std_logic;
        signalin1       : in std_logic;
        signalin2       : in std_logic;
        signalout1      : out std_logic;
         signalout2     : out std_logic;
         clk            : in std_logic);
end FSM;
```

bidir

Block scheme:



Function : Bidirectional 8-bits bus

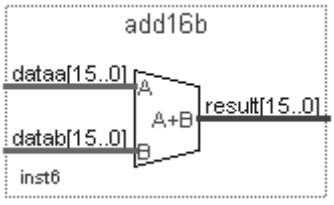
Bidir are the in/output pins. On output you get the incoming signals from the databus (when ‘oe’ is 0) and on input you can set the outgoing signals to the databus (when ‘oe’ is 1).

VHDL Entity description:

```
ENTITY bidir IS
   PORT(
      bidir   : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
      oe, clk : IN STD_LOGIC;
      inp     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      outp    : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END bidir;
```

add16b

Block scheme:



Function : 16bit signed adder/subtractor

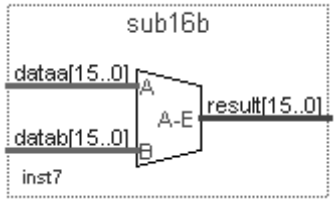
Adds two 16 bit numbers

VHDL Entity description:

```
ENTITY add16b IS
  PORT (
            dataa               : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            datab               : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            result              : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END add16b;
```

sub16b

Block scheme:



Function : 16bit signed adder/subtractor

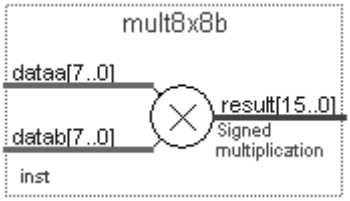
Subtracts two 16 bit numbers

VHDL Entity description:

```
ENTITY sub16b IS
  PORT (
            dataa           : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            datab           : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            result          : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END sub16b;
```

mult8x8b

Block scheme:



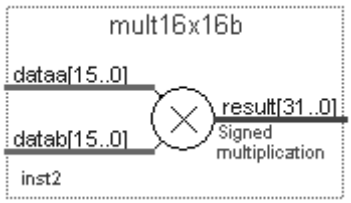
Function : 8-bit signed staged multiplier

VHDL Entity description:

```
ENTITY mult8x8b IS
  PORT(
            dataa           : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            datab           : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            result          : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
END mult8x8b;
```

mult16x16b

Block scheme:



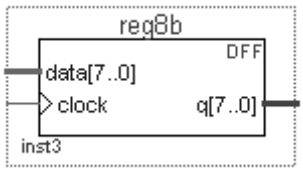
Function : 16-bit signed staged multiplier

VHDL Entity description:

```
ENTITY mult16x16b IS
  PORT(
            dataa           : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            datab           : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            result          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0));
END mult16x16b;
```

reg8b

Block scheme:



Function : Function : 8-bit register

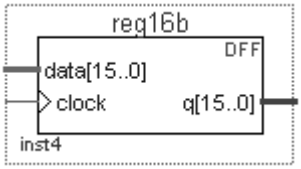
The input data is stored on the positive clock edge

VHDL Entity description:

```
ENTITY reg8b IS
  PORT(
            datain          : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            clk             : IN STD_LOGIC;
            dataout         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END reg8b;
```

reg16b

Block scheme:



Function : 16-bit register

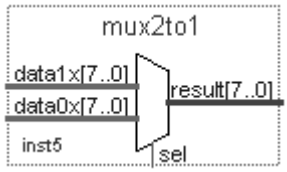
load = 1 means store input data on positive clock edge

VHDL Entity description:

```
ENTITY reg16b IS
PORT(
        datain          : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
        clk             : IN STD_LOGIC;
        load            : IN STD_LOGIC;
        dataout         : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END reg16b;
```

mux2to1

Block scheme:



Function : 8-bit 2 to 1 multiplexer

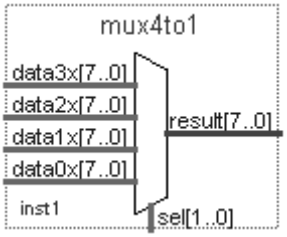
sel = 0 means result <= data0x
sel = 1 means result <= data1x

VHDL Entity description:

```
entity mux2to1 is
  port (
        data0x : in STD_LOGIC_VECTOR(7 DOWNTO 0);
        data1x : in STD_LOGIC_VECTOR(7 DOWNTO 0);
        sel    : in STD_LOGIC;
        result : out STD_LOGIC_VECTOR(7 DOWNTO 0));
end mux2to1;
```

mux4to1

Block scheme:



Function : 8-bit 4 to 1 multiplexer

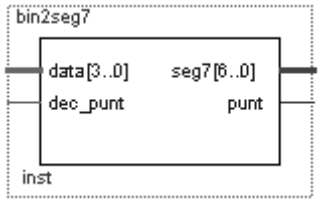
sel = 00 means result <= data0x
sel = 01 means result <= data1x
sel = 10 means result <= data2x
sel = 11 means result <= data3x

VHDL Entity description:

```
entity mux4to1 is
  port (
        data0x : in STD_LOGIC_VECTOR(7 DOWNTO 0);
        data0x: in STD_LOGIC_VECTOR(7 DOWNTO 0);
        data0x: in STD_LOGIC_VECTOR(7 DOWNTO 0);
        data0x: in STD_LOGIC_VECTOR(7 DOWNTO 0);
        sel    : in STD_LOGIC_VECTOR (1 DOWNTO 0);
        result : out STD_LOGIC_VECTOR(7 DOWNTO 0));
end mux4to1;
```

bin2seg7

Block scheme:



Function : 7 LED-segment driver

Converts values in a 4 bit binary format to a format that can be used to drive a 7-segments led display.

VHDL Entity description:

```
ENTITY bin2seg7 IS
  PORT(
        data        : in std_logic_vector(3 downto 0);
        dec_punt    : in std_logic;
        seg7        : out std_logic_vector(6 downto 0);
        punt        : out std_logic);
END bin2seg7;
```

Appendix

G A simple Quartus example.

An example how to make a 4by4 multiplier in Quartus.

1. Starting a project

File -> New Project Wizard

Give working directory.
Name of this project.
Top-level entity (no space's in name).
Click on the Next button.

Add Files

Not necessary, click on the Next button.

Specify other EDA tools

Not necessary, click on the Next button.

Choose the device

Flex10k,
Click on the Next button.
Select, *Flex 10k20 RC 240-4* or,
Flex 10k70 RC 240-4 depends on the board you are using.
Click on the Finish button.

2. Make a schematic design

In Quartus it's possible to make a logic design with symbol blocks. This give the you advantage that you don't have to use VHDL with the difficulty of the syntax. Now we show you how to make a schematic design.

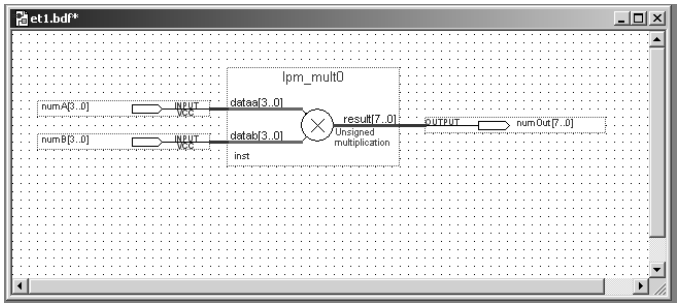
File -> New -> Block Diagram / Schematic File

Click on: **Symbol Tool** button
Open: **+/quartus/libraries +megafunctions +arithmetic +lpm_mult** symbol
Click on the OK button
The Mega Wizard Plug-In Manager will be started now.
Choose *VHDL* output file.
Click on the Next button.
Select the wide of dataa input bus *4* bits.

Select the wide of datab input bus 4 *bits*

Click on the Next button.
Click on the Next button.
Click on the Next button.
Click on the Finish button.

Click the mouse and the lpm_mult symbol will be placed in the schematic design. Push the Esc button to leave the placing of the symbol.



3. Make the in- output labels

Click on **Symbol Tool** button
Open **/quartus/libraries/ +primitives+pin +input** label

Click on OK button

Click the mouse and the input label will be placed, click a second time for placing the second input label.

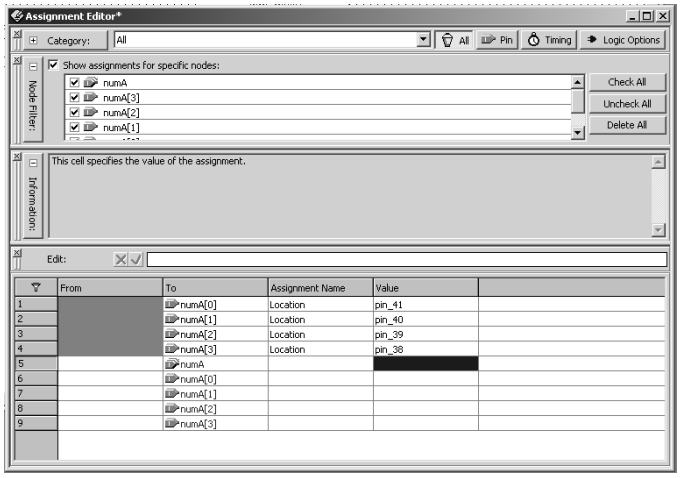
Give label names:
Select the first input label
Properties (right mouse click)
Pin name(s): *numA[3..0]*
Click on OK button
Name the second input label *numB[3..0]*
Make an output label by selecting the **output** symbol in the **/quartus/libraries/ +primitives +pin**
Give the output label in the **Properties** menu the name: *numOut[7..0].*

Connect the inputs and the output to the lpm_mult symbol with the **Orthogonal Bus Tool** Button.
Save the Block Diagram/Schematic file with the same name as the top-entity .
File -> Save As

Compile with the **Start Compilation** button or with **Processing -> Start Compilation** menu.

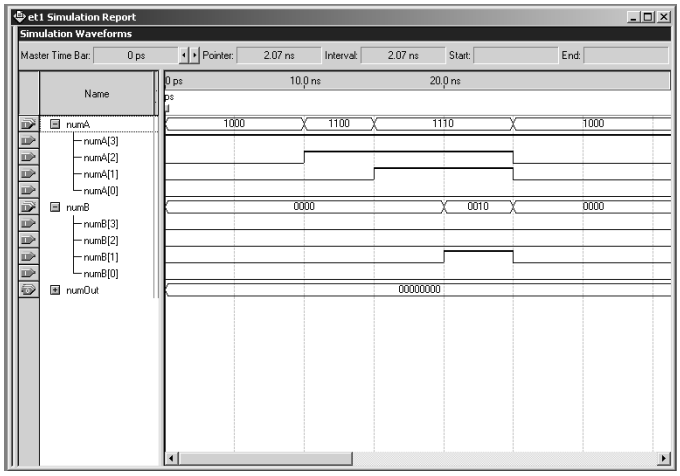
4. Pin Assignments

The pin numbers 33 till 41 of the Flex 10k are connected to the flex_sw1 switch on the board. If we want to use the sw1 switch as input for the 4by4 multiplier, than we have to make the following pin assignments.



Select the input label NumA. Open the **Assignment Editor** in the right click mouse menu.
The **Assignment Name** of the nodes numA[0] till numA[3] must be set on **Location** (double the field and select). On the fields under the **Value** you must type the correct pin number for all the nodes. It's possible to type only the pin number, the assignment editor change it to Pin_num.
Do the same for the labels numB and numOut. On page C-8 you will find the pin number switches. For output its possible to use the leds on the board which you should connected to I/O pins on the flex_explan_a connector by wires. You have to find out witch connectors are free and what there pin numbers are.

5. Simulator.



Make a Vector Waveform file.
File -> New -> Other Files->Vector Waveform File

Go with the mouse to the **Name** Box of the Vector Waveform.
Right mouse click.
Select **Insert Node or Bus…**
Click on **Node Finder…**
Click on List button
Copy the node names NumA, NumB, NumOut to the **Selected Nodes** box.
Click on OK button

Open the nodes in the vector waveform, select one of the lines or select a piece of the line and give a value.

Before running the simulation save the wave form **File -> Save As** give the same name as the top-entity.

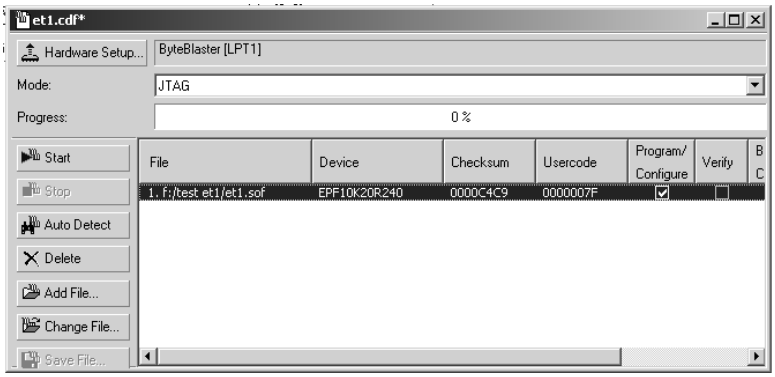
Run the simulation with the  button.
With the **TOOLS -> Simulation Tool** menu its possible to set the duration of the simulation.

With the  button it's possible to set a line as a clock signal with an adjustable frequency.

6. Programmer

Go to **TOOLS -> Programmer**
The **Programmer/ Configure** must be checked on.
Click on the 

button and the FPGA will be programmed.



7. Create a symbol file

If you have a VHDL file it's possible to create a symbol block file.
First open the VHDL file: **File -> Open -> VHDL File**
Go to the menu **File -> Create/Update -> Create Symbol Files for Current File**.
In the symbol tool button you can find the symbol under the Project library. The symbol has the same name as the entity name with the same inputs and outputs as the VHDL file contains.

8. Archive a project

When you archive a project all your project files will be zipped to one file. Now your whole project is easy to transport to a save location (save to you student account) and you can used it the next time.

Project -> Archive -> Project…
All the files with the directory structure will be saved under *filename.qar*

Project -> Restore -> Archived Project…
The file *filename.qar* will be extract to all the project files.