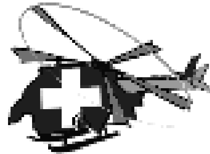
ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Departement Informatik
Institut für Computersysteme

Marco A.A. Sanvido



A Computer System for Model Helicopter Flight Control

Technical Memo Nr. 3: The Software Core

April 20, 1999

317

Contents

1 Introduction **1**

2 Memory Organization **3**
2.1 Heap . . . 3
2.2 Stacks . . . 5
2.3 Interrupt Vector Table (IVT) . . . 5
2.4 Caches . . . 5

3 The File System **7**
3.1 The RAM Disk . . . 7
3.2 The ROM Disk . . . 8

4 Input and Output **9**
4.1 Introduction . . . 9
4.2 Analog to Digital Converters . . . 10
4.3 Pulse Width Modulation and Rotor Frequency . . . 11
4.4 Serial Communication . . . 13
4.5 Digital I/O . . . 14
4.6 FPGA initialization . . . 14

5 The Linker/Loader **16**
5.1 Linking Process . . . 17

6 Host Protocol **19**
6.1 Host to Target . . . 19
6.2 Target to Host . . . 19
6.3 Debugging . . . 20

7 Real-Time Scheduling **21**
7.1 Tasks and Their Priority . . . 21
7.2 Implementation and Performance . . . 22
7.3 Example . . . 24

8 Floating-point Emulation **25**
8.1 Look-ahead Optimization . . . 25
8.2 Performance . . . 25
8.3 Floating-point Library . . . 26

9 Numerical Support **27**
9.1 The Math Module 27
9.2 The MatLib Module 27

10 The Startup Process **28**

11 HelyOS Commands **29**

12 HLog Module **31**

A FPGA **32**

B Helicopter System Modules **36**

Chapter 1

Introduction

This technical memo describes the software core of Olga[1]. The computer architecture was described in memo nr. 1 [13]. The whole system, except the bootstrapping code that resides in ROM was written with the Oberon for StrongARM compiler described in memo nr. 2 [14].

The software core furnishes the basic functionality – device drivers, interrupt handlers, memory management, debugging and linking/loading – to the helicopter auto pilot application [5], therefore it has to be as small and as fast as possible. We think we achieved this goal with less than 10 Kbytes object code. With the use of the new features of the compiler we were able to implement the device drivers and memory management in an efficient way.

The module structure in figure 1.1 shows the organization of the software core. The base module is HKernel. It implements the memory management, UART driver and the RAM disk driver. HFiles and HFileDir modules implement a file system for the RAM disk and ROM disk. HModules implements the linking/loader. HelyOS[2] is the main module; it implements the scheduling and starts the command loop. This procedure then waits for incoming commands to be executed on the target system. The ADC module reads analog values, filters and converts them into the desired unit. The Servo module reads the input PWM[3] signal generated from the pilot transmitter and generates the output PWM signal for the auto pilot functions. We will see afterwards in detail where all these signals are generated and used. The module HLog writes information to the host computer system. The FPE module implements the floating-point emulation. This is necessary because the StrongARM processor has no floating-point unit. The Math and MatLib modules are a set of efficiently implemented mathematical functions.

[1] *O*beron *L*anguage *G*oes *A*irborne

[2] The name HelyOS and not HeliOS was chosen mainly for two reasons: one, because a system named HeliOS, a real-time operating system for the ARM processor, already exists. Second because when I wrote the module for the first time, I mistyped the name

[3] *P*ulse *W*ith *M*odulation

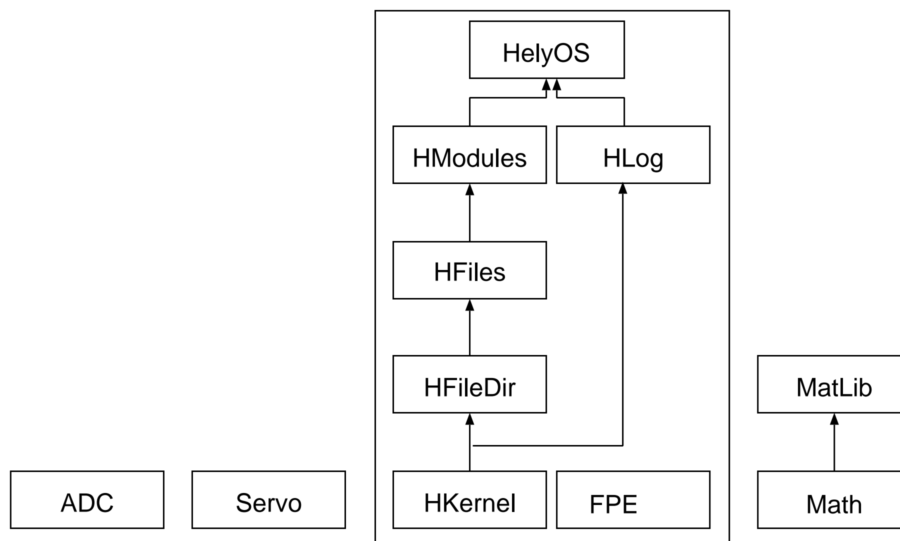


Figure 1.1: Software Core

Chapter 2

Memory Organization

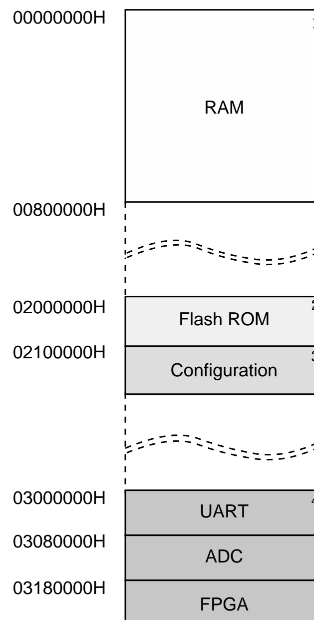


Figure 2.1: Memory Map

Figure 2.1 shows the four different parts of the memory layout adopted. The first block of 8 Mbytes is the RAM of the system, the second block of 1 Mbyte is the system ROM, actually an in-system-programmable FlashROM. The third block is used to execute the special commands for refreshing the SDRAM chips. In the last block all the I/O locations are memory mapped. More details can be found in the report [13]. Figure 2.2 shows in detail how the RAM is subdivided.

2.1 Heap

The heap implementation is very simple. There is a linked list of free blocks. A simple first-fit strategy is used for the allocation of a new block. At first

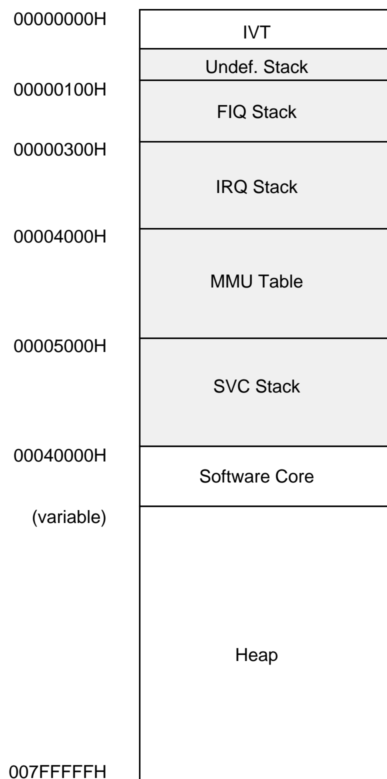


Figure 2.2: Memory Organization

sight this seems to be a simplistic solution, but we have to keep in mind that the system will run only a single application – the controller of the helicopter – and that a complex heap management scheme will not be needed. This is the simplest but also the most effective solution in terms of code size and allocation time. Moreover no garbage collection is implemented since the heap will be used only for the allocation of modules and variables. The procedures that handle the heap are:

```
MODULE HKernel;

   PROCEDURE Alloc(VAR a: Ptr!; size: INTEGER);
   PROCEDURE Release(a: Ptr!);
   PROCEDURE MemAvail(VAR size: INTEGER);

END HKernel.
```

The standard procedure *NEW()* represents a call to the procedure *Alloc()*. Therefore there is no need to directly call *Alloc*. Since no garbage collection is provided the procedure *Release()* has to be called whenever a block needs to be freed. The procedure *MemAvail* returns the amount of heap space still available.

2.2 Stacks

There are four different stacks; one for each mode of operation of the StrongARM [3] that we use (supervisor, undefined, fast-interrupt and normal-interrupt). Figure 2.2 shows the location of the four stacks in RAM. The MMU Table – to be explained later – is placed between the SVC Stack and the IRQ Stack. This unusual location is due to the fact that the MMU table can be placed only in specific locations in memory. Normally the processor is in *Supervisor Mode*. We choose not to run the processor in *User Mode* since the software core does not have to protect itself from malicious applications. The other important operation modes are *IRQ Mode* and *FIQ Mode*. These are entered when the processor starts an interrupt handler. In the first case this happens in response to a timer that interrupts the processor at a frequency of 200 Hz, in the second case in response to an UART request. The fourth and final mode is the *Undefined Instruction*. When and how this mode is used will be discussed in more detail in chapter 8. The StrongARM has other operation modes which are not used in our implementation.

2.3 Interrupt Vector Table (IVT)

The StrongARM uses this table to jump to the interrupt handlers. The table is initialized during startup. An interrupt handler has to register its entry address in the table with the *InstallHandler* procedure. The structure of the table is shown in Figure 2.3. The compiler extends this table for the *NEW()* handler, placed at the address 50H.

```
DEFINITION HKernel;

   PROCEDURE InstallHandler(handle: PROCEDURE(); vector: INTEGER);

END HKernel.
```

2.4 Caches

To accelerate code execution of the processor we use the on-chip caches and memory management. The StrongARM has a 16 Kbytes 32-way associative instruction cache, a 16 Kbytes 32-way associative write-back data cache, a memory management unit (MMU) and a 8-entry write buffer. As mentioned above we want to access the memory via the cache whenever possible. We need to use the MMU to mark the blocks of RAM as being cacheable. This is necessary since we use memory mapped I/O, like UARTS and FPGA. For these addresses no cache and write buffers can be used. Consequently we subdivide the memory in pages of 1 Mbyte and associated each of them with two access types: cacheable or non cacheable. For details on how this can be implemented we refer to [3] and [2]. The MMU table and caches are initialized during the startup phase.

Since the processor has two caches, one for data and one for the instruction (Harvard Architecture), each time that instructions are handled in the data cache, the HKernel has to flush both caches to ensure that the instructions copied will reside in memory and not in the data cache. This happens every

Address	Vector	Use
00000000H	Reset	NOT USED
00000004H	Undefined Instruction	Floating Point Emulation
00000008H	Software Interrupt	NOT USED
0000000CH	Abort (prefetch)	NOT USED
00000010H	Abort(data)	NOT USED
00000014H	reserved	NOT USED
00000018H	IRQ	200Hz generated from FPGA
0000001CH	FIQ	UART Interrupt
00000050H	NEW()	

Figure 2.3: Interrupt Vector Table

time we load a new module. During the loading phase the body of the module has to be executed. Therefore we have to flush the caches to ensure that the code effectively resides in RAM. The StrongARM has no instruction that completely cleans the caches. It only has a FLUSH instruction that fully erases the cache contents. Therefore after each module load, we need to call the procedure *CleanCache()* to ensure that each cached value is consistent with the memory locations referenced.

```
DEFINITION HKernel;

   PROCEDURE CleanCache();

END HKernel.
```

Chapter 3

The File System

The file system implementation is similar to the standard Oberon [15] file system. There are only two differences. First, the files stored in the RAM disk can be made persistent, i.e. moved to the ROM disk. These files cannot be deleted, nor can they be written. Second, the file system can access the host file system: reading files from and writing files to the host.

The file system uses 1 Kbyte sectors, and the maximal length of a file is 3 Mbytes. This is not really a restriction since the RAM disk is only 6 Mbytes.

Every file on the RAM and ROM disks have a mark value (HeaderMark) stored in the header. This value allows the file system to scan the disks during startup to find previously stored files. The files found are inserted in the directory structure and the sectors used are marked as used in the sector allocation table.

3.1 The RAM Disk

The following text is the definition of the HFiles interface for files resident in the RAM disk.

```
DEFINITION HFiles;

   CONST
      HeaderMark = 09BA71D86H;

   TYPE
     File = POINTER TO RECORD
        name: ARRAY 32 OF CHAR;
     END;
     Rider = RECORD
        eof : BOOLEAN;
     END;

   PROCEDURE Old(VAR name: ARRAY OF CHAR): File;
   PROCEDURE New(VAR name: ARRAY OF CHAR): File;
   PROCEDURE Length(f: File): INTEGER;
   PROCEDURE Register(f: File);
   PROCEDURE Close(f: File);
```

```
   PROCEDURE Purge(f: File);
   PROCEDURE Delete(f: File);

   PROCEDURE Set(VAR r: Rider; f: File; pos: INTEGER);
   PROCEDURE Pos(VAR r: Rider): INTEGER;
   PROCEDURE Read(VAR r: Rider; VAR ch: CHAR);
   PROCEDURE Write(VAR r: Rider; ch: CHAR; VAR res: INTEGER);

   PROCEDURE RemoteReadFile(VAR name: ARRAY OF CHAR): File;
   PROCEDURE RemoteWriteFile(f: File);

END HFiles.
```

3.2 The ROM Disk

The files stored in the FlashROM memory can be used like normal files. However, as these files are marked as read only, it is not possible to change them, nor to delete them. The only way to delete a file from the ROM disk is to completely erase the ROM disk, thus erasing every file previously stored in the ROM. This is necessary since the FlashROM does not support erasing small portions of the ROM. The ROM disk is 512 Kbytes in size, since the other half of the ROM is used for the bootloader, FPGA bitstream and core software.

```
DEFINITION HFiles;

   PROCEDURE MakePersistent(VAR f: File);
   PROCEDURE ReadOnly(f: File): BOOLEAN;
   PROCEDURE ResetROMDisk();

END HFiles.
```

Chapter 4

Input and Output

4.1 Introduction

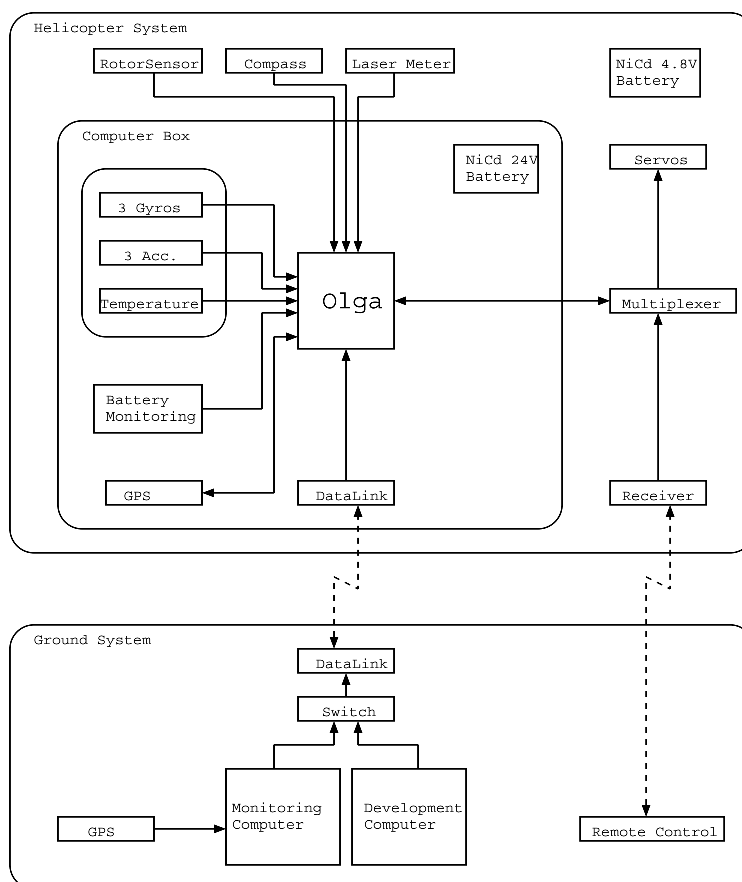


Figure 4.1 shows how the Olga computer is connected to the Helicopter and which signals have to be generated and read by the Olga system. There are four types of signals.

- Analog signals, generated by the inertial unit, a device with three gyroscopes, three accelerometers and a temperature sensor. The temperature sensor is needed to compensate the temperature drift of the accelerometers.
- PWM[1] signals, are the input signals of the servos used to control the position of the main rotor (4 servos), the tail rotor (1 servo) and the engine (1 servo). The PWM signals are also generated by the Pilot receiver. These signals are needed in case of am emergency, to allow the pilot to get back the control of the helicopter in dangerous situations. The signals can be read by Olga, in order to have a smooth transition from auto pilot to human pilot and vice versa.
- The third type of signal are the four RS-232 UART lines used for the on-board GPS, the compass, the altimeter sensor, and DataLink. The latter, is the "umbilical cord" to the ground. At startup teh Datalink is used to start the auto pilot system. During autonomous flight it is used to log data on the ground, and to send the differential correction messages to the on-board GPS system.
- The latter type of signals are pure Digital I/O used for various switches.

4.2 Analog to Digital Converters

The Oberon module ADC implements the interface to the analog converters. Two MAX196 converters are used with a 12-bit resolution and with 6 input channels each. The procedure *Read* initiates a conversion on the specified channel (0 .. 11) and polls the ADC channel until it has finished the conversion. This takes between 3 μs and 5 μs. Thereafter it reads the converted data. The other procedures read directly from the appropriate channel, filters and transform the read voltage into the required unit. For instance the procedure *Temperature* returns the temperature in degrees Celsius. Further information and details on the chip are contained in [9].

The procedure *Temperature* implements the following low-pass filter:

$$temp \quad := \quad 0.998 * temp + 0.002 * sensortemp$$

The procedure *MotionPak* transforms the input data into m/s^2 and into $grad/s$. Moreover, since the accelerometers are temperature sensitive, their output voltages have to be adjusted based on the current temperature. These are the equations used for the transformations of the gyros and accelerometers:

$$\begin{aligned} omegax &:= scaleox * sensorgyrox; \\ oemgay &:= scaleoy * sensorgyroy; \\ omegaz &:= scaleoz * sensogyroz; \\ accx &:= ax1 * temp^2 + ax2 * temp + ax3 + ax4 * sensoraccx; \\ accy &:= ay1 * temp^2 + ay2 * temp + ay3 + ay4 * sensoraccy; \\ accz &:= az1 * temp^2 + az2 * temp + az3 + az4 * sensoraccz; \end{aligned}$$

[1] *P*ulse *W*ith *M*odulation

The values *scaleox*, *scaleoy* and *scaleoz* are the actual scaling factor from sensor value to physical value. The values $ax1...ax4$, $ay1...ay4$ and $az1...az4$ compensate the temperature drift of the accelerometers. These values were computed on the basis of measurements, taken in a special oven.

```
DEFINITION ADC;

   PROCEDURE Read (channel: INTEGER; VAR analog: INTEGER);
   PROCEDURE Temperature (): REAL; (* C *)
   PROCEDURE Battery (): REAL; (* V *)
   PROCEDURE MotionPak (VAR ox, oy, oz,
                            ax, ay, az: REAL); (* rad/s & m/(s*s) *)
   PROCEDURE SetMPOffset (oxoff, oyoff, ozoff,
                          axoff, ayoff, azoff: REAL)

END ADC.
```

4.3 Pulse Width Modulation and Rotor Frequency

The servos are the actuators that actually control the helicopter. There are 6 servos – one for the engine, one for the yaw (tail rotor), the other four [2] for pitch, roll and collective (main rotor). Their input signals are pulse width modulated (PWM). These signals come from a multiplexer (see Figure 4.1) that switches the PWM signals generated by a radio receiver with the PWM signals generated from the auto pilot computer. For development purposes we need to be able to read the signals of the radio receiver in order to smoothly change from the pilot control to the auto pilot.

These functions are implemented in an FPGA: a Xilinx XC6200 [16]. In the FPGA we implemented 6 PWM generators. The Lola [11] specification and the design are contained in Appendix A. To implement and program the FPGA we used the Trianus and Hades systems. These tools are described in detail in [4] and [8].

The PWM signal is a pulse with a frequency of about 50 Hz (see figure 4.2). The pulse length is the relevant signal information and varies from 1 ms to 2 ms. The servo accepting this signal positions the actuator proportionally to the signal length. The main circuit is a counter that counts 20 ms. Actually since the clock frequency is 230,4 kHz we need to count to 4608 with a 13-bit counter. The PWM generator is simply a comparator that resets a flip-flop when the counter reaches the desired pulse length. Thereafter, when the counter starts the next cycle, it will reset the flip-flop. The number of FPGA cells used for this purpose is 362 of the 4096 available, that is less than 9%.

The FPGA also contains 6 PWM inputs and a rotor frequency timer (RFT). The latter is similar to a PWM signal, but instead of counting the length of the pulse, the circuit measures the length between pulses. The main clock is 230.4 kHz, therefore at least 9 bits are needed to cover the maximal length of a PWM pulse of 2 ms. The maximal pulse length is $2^9/(230.4kHz) = 2.223ms$. Logically the RFT needs a larger number of bits. Since we suppose a maximal period of

[2] one of them is redundant

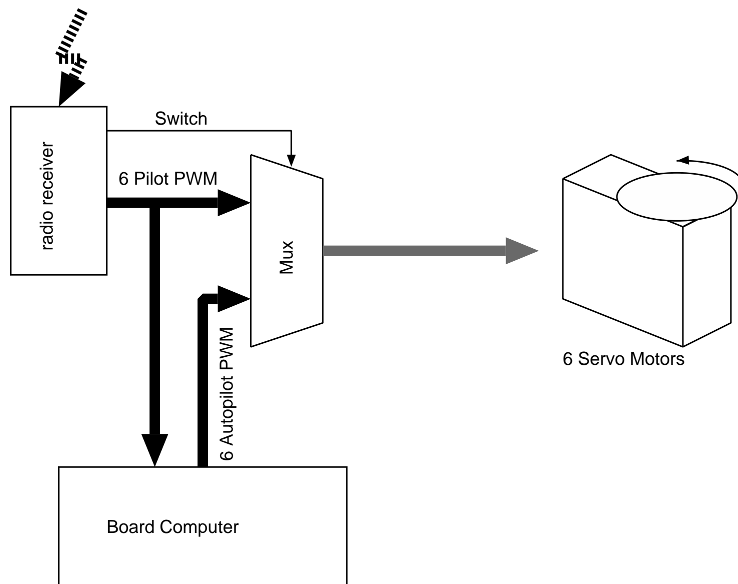


Figure 4.1: Multiplexer

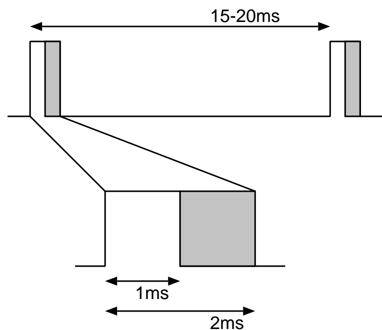


Figure 4.2: Pulse Width Modulation Signal

about 1 second we need at least 18 bits. At full speed the rotor turns at 1200 RPM. Since the sensor generates 3 pulses per rotation the minimal value is be about 17 ms. That means that 12 bits are used under normal flight conditions. Even in this case the number of cells used is very small: 358.

The procedure *PWMIn* reads the width of the specified channel and transforms it into microseconds. Similarly the procedure *Rotor* returns a third of the rotation period. The procedure *PWMOut* sets the length of the generated PWM pulse. The last two procedures return the position of 2 switches used for the switching of the PWM signals on the multiplexer board. These two procedures reside in this module since they are actually PWM signals sent by

the pilot transmitter and decoded by a multi-switch decoder.

```
DEFINITION Servo;

   PROCEDURE PWMOut(channel, pwm: INTEGER); (* us *)
   PROCEDURE PWMIn(channel: INTEGER; VAR pwm: INTEGER); (* us *)
   PROCEDURE Rotor(VAR period: INTEGER); (* us *)

   PROCEDURE Switch1(): BOOLEAN;
   PROCEDURE Switch2(): BOOLEAN;

END Servo.
```

The FPGA also holds the circuitry necessary for the generation of a 200 Hz interrupt. This circuit is a simple counter operating at a frequency of 230.4 kHz, therefore the counter counts up to 1152 (11 bits).

The complete implementation on the FPGA takes 699 cells, which is about 17% of the available space. We plan to port more functionality to the available space in the future. Possible functionality could be the UART implementation, digital filters or even some control function. Some feasibility tests have already been done with a UART implementation.

4.4 Serial Communication

The UART used is a Philips SC28L194. This complex, low-power, and powerful chip provides 4 parallel UART channels. The module HKernel implements its driver since it needs the serial communication to down-load modules after startup.

To access a UART channel it has to be initialized with the procedure *Start*. This initializes an input buffer of 256 characters and configures the channel with the parameters MR0, MR1 and MR2, CSR – for the meaning of these values we refer to [10]. Moreover, the interrupt on receive is activated. On the other side, the procedure Stop breaks the communication and disables the interrupt for that channel. However, the software buffer is accessible and contains the characters received until the Stop routine was called.

When a channel is activated with the routine *Start*, the UART is ready to receive and transmit bytes via the procedures *Send* and *Receive*. Furthermore, the procedure *Available* returns the number of characters deposited in the channel input buffer.

Since the receivers of the UART are asynchronous we need to implement an interrupt handler. The UART interrupt request generates a fast interrupt on the StrongARM which allows a very fast response to the request. The current implementation needs less than 2.5 μs. To achieve such a speed the handler has to be as simple as possible, but it also must be able to handle every kind of interrupt request. The UART can generate different kinds of interrupt requests but we allow only two types of interrupts: *receive with errors* and *receive without errors* (see [10]). Furthermore, when an interrupt request arrives, the handler must acknowledge the request and find the request source (the channel that generated it). Then it has to check if the requester is really ready to deliver the information in case of a *receive without error* interrupt. If an interrupt of the

type *receive with errors* is detected, the handler not only has to acknowledge the interrupt as for the errorless case but also must eliminate the source of the error. Usually reading the incorrect byte from the UART is enough. The handler is installed during initialization.

```
DEFINITION HKernel;

   PROCEDURE Start(channel: INTEGER; MR0, MR1, MR2, CSR: CHAR);
   PROCEDURE Stop(channel: INTEGER);

   PROCEDURE Send(channel: INTEGER; ch: CHAR);
   PROCEDURE Receive(channel: INTEGER; VAR ch: CHAR);
   PROCEDURE SendInt(channel: INTEGER; int: INTEGER);
   PROCEDURE ReceiveInt(channel: INTEGER; VAR int: INTEGER);
   PROCEDURE SendString(channel: INTEGER; VAR s: ARRAY OF CHAR);
   PROCEDURE ReceiveString(channel: INTEGER; VAR s: ARRAY OF CHAR);

   PROCEDURE Available(channel: INTEGER): INTEGER;

END HKernel.
```

4.5 Digital I/O

The helicopter needs some digital I/O for the control of external devices, for example a grabber. The UART described in the previous section provides four general digital I/O per channel. At present four of them are connected to LEDs. These are used for the visulazation of system states. The other 12 are free for future use.

The procedure *IOPC* configuresthe I/O pin of the four channels for input or output mode. *OP* writes the four output pins and *IP* reads the value of the four pins. Furthermore, the FPGA has a number of connected but unused pins that could easily be adapted to implement digital I/O.

```
DEFINITION HKernel;

   PROCEDURE IOPC(channel: INTEGER; ch: CHAR);
   PROCEDURE OP(channel: INTEGER; x: CHAR);
   PROCEDURE IP(channel: INTEGER; VAR x: CHAR);

END HKernel.
```

4.6 FPGA initialization

Since the FPGA used is a SRAM-based device, it has to be configured during the startup phase. We chose not to use a serial boot ROM for the FPGA, since this chip would be used only at startup and for nothing else. Our solution, instead is to configure the FPGA explicitly from the StrongARM.

We implemented two modes in which the FPGA can be configured. The first mode is similar to the down-loading of a module. The main difference is that the configuration bitstream is then copied into the SRAM of the device rather than that of the RAM memory. Such a down-load takes approximatively

8 seconds since the bitstream size is about 110 Kbytes and the serial link speed is 115 Kbauds. This solution is acceptable for the tests run in the lab, but not in the final implementation.

The second mode is a more efficient solution. The bitstream file is stored in the FlashROM. Right after startup the HKernel calls the procedure FPGAInit that reads the bitstream file and copies it to the FPGA configuration memory. This takes only about 36 ms.

Chapter 5

The Linker/Loader

The Linker/Loader strategy adopted is the same as in the original Oberon operating system [15]. The modules needed are loaded and linked dynamically on the target system. The helicopter computer does not have any user interface devices like keyboard, display or mouse. Therefore all the development has to be done on a host cross-platform. The host holds the development environment (a PC running the Oberon System) while the cross-compiled programs have to be down-loaded via serial link to the target system. The host-target protocol is discussed in the next chapter, at first we will focus on the method in which the object file created on the host platform and transfered to the system is linked, loaded and consequently made accessible to the rest of the system.

The HKernel holds a list of the modules loaded, the module data structure is as follows:

```
Module = POINTER TO RECORD
   size, key: INTEGER;
   name: ARRAY 32 OF CHAR; (* module identification *)
   code, entrytab, ptrtab, cmdtab, importab: INTEGER; (* address *)
   refcnt: INTEGER; (* reference count *)
   link: Module (* linked list *)
END;
```

We adopted the same data structure as that used in the *Oberon0* [1] system for the IT computer, i.e. a prototype of a network computer developed by Digital with a StrongARM processor. When a module is loaded, it is placed in the heap memory. After the linking process, the module structure looks like the one shown in Figure 5.1. The command table holds the links to the commands (parameterless procedures) declared within the module. The pointer table is always empty, since no garbage collector is implemented. The entry table holds the addresses of the exported procedures. Note that position 0 of the table holds the entry address of the module body.

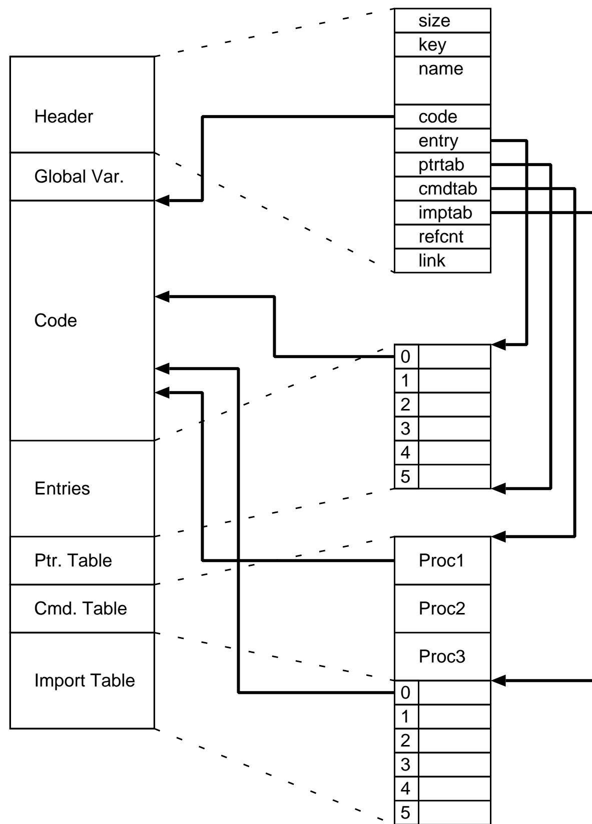


Figure 5.1: Module Structure

The main procedures used for loading and linking are:

```
DEFINITION HModules;

   PROCEDURE FindModule(VAR name: ARRAY OF CHAR; VAR m: Module);
   PROCEDURE ReceiveMod(VAR name: ARRAY OF CHAR; VAR m: Module);
   PROCEDURE CallP(m : Module; VAR pname: ARRAY OF CHAR);

END HModules.
```

FindModule searches in the list of modules loaded for the module with the specified name. If it finds it, it returns its pointer, otherwise NIL. *CallP* calls the command named pname of the module m. The procedure *ReceiveMod* is responsible for the recursive down-loading process. The *FindModule* is used to test whether the requested module is already linked. If it is not, then the host is requested to send the module. Then the Module is linked and its body executed.

5.1 Linking Process

When the module is completely loaded and placed in memory the external calls have to be replaced by real relative addresses. This is done with fixup chains,

more precisely, one chain per imported module. The fixup chain is simply a linked list of all external calls of a module.

The linking process traverses these fixup chains and replaces the offset of the next external call with the relative jump offset for the called procedure. Since the system does not support the import of variables, there are only external procedure calls that have to be fixed. Furthermore, the reference counter is initialized with zero and incremented by one each time the module is imported by another one. This is necessary in order to allow a safe unloading of modules.

Chapter 6

Host Protocol

This chapter describes the protocol used for the host to target and target to host communications. This protocol is needed during the startup phase to establish a means of communication. After the startup process and after the HKernel module has initialized everything, the main module HelyOS enters the endless procedure *Loop*. This procedure waits for messages coming from the serial port and interprets them. On the other hand the host has an Oberon task installed (see [15]) that polls the serial port and waits for commands arriving from the target. When the auto pilot takes control of the helicopter the serial cable is disconnected and the auto pilot computer communicates via the radio communication protocol, described in another technical report [6].

6.1 Host to Target

The target accepts two type of message. The command format is:

CallProc = sync:3 CP:1 name len:4 {char:4} parameter:32.

LoadMod = sync:3 MD:1 name.

The first command tells the target to execute the command specified with the string name. The name string has to have the structure $x.y$ where x is the anme of the module and y the parameterless procedure name of the command to be executed. Furthermore it is possible to give a parameter string to the procedure called via the field *parameter*. This string can be accessed on the target via the *GetParameter* procedure. If the module is not yet loaded the target automatically requests the module from the host (see next section). The second command tells the system to load a module and as in the case above the module is down-loaded only if it is not yet in the target. The *sync* value is a synchronization sequence that allows the target to synchronize with the byte stream, eliminating possible transmission errors.

6.2 Target to Host

The host accepts the following messages from the target system. The underlined words are the responses from the host. The *sync* sequence, as in the host to

target case, is a synchronization sequence that allows the host to synchronize with the byte stream.

FileRequest = sync:3 FR:1 name [len:4 {char:1} checksum:4 | -1].

FileSend = sync:3 FS:1 name len:4 {char:1}.

FileSendPacked = sync:3 FSP:1 name len:4 {pckn:1 {char:1} chk:1} (ACK:1 | NACK:1).

Log = sync:3 LOG:1 (LN:1 | STR:1 string | CH:1 ch:1 | INT:1 int:4 | HEX:1 int:4 | FLOAT:1 real:4 | CLEAR:1).

ModuleStatus = sync:3 MS:1 modname key:4 adr:4 {class:1 form:1 adr:1 x:4} 0FFX.

With the first two messages the target can load a file from the host or write a file on the host. The HFiles module uses this protocol to access the remote file system. The file request message includes a checksum. This is necessary since a transmission error during the transfer of an object file could compromise the integrity of the whole system. On the other hand, the file transfer from target to host is more critical, since the host system is not a real-time system and cannot guarantee a real-time response. Therefore there are two ways for sending a file, the first being a straight-forward sending of data via the serial link. This works very well for small files and is very fast. If the files are larger, the host system cannot sustain the data transfer and may lose some of the incoming bytes. To overcome this problem we implemented a handshake protocol that slows down the effective transfer rate, but ensures a correct transfer of large files. The file is subdivided in packets with a checksum and a packet number. In this way the host system can check errors in the packets received and, in case it detects an error, it can request the re-sending of a packet.

The Log message is used by the HLog module to display information in the *Target Log Viewer* – a special Oberon Viewer [15] opened at startup on the host system.

The last message is used for debugging. See the next section for details.

6.3 Debugging

The user can analyze the status of all global variables of a specified module, with the HelyOS command *ShowStatus*. When the HKernel wants to display such information it requests the host system to send the relative offsets, type and name of all global variables. The host system can access this information thanks to the symbol file extensions stored on the host system. For more details on the file structure see [14].

Chapter 7

Real-Time Scheduling

In general real-time operating systems the scheduling of the different system tasks has to be versatile, robust and must run transparently with as little overhead as possible. In our case, however, we have an undeniable advantage: we know the kinds of jobs needed by the auto pilot application. Therefore we can focus and optimize our strategy for this application. We are convinced also that the implementation of a simple and clear strategy, has the side effects of a more robust, faster and smaller code size implementation.

The stategy usually adopted is the use of coroutines as the multitasking entity. Our approach (see [12]),uses subroutines as the multitasking entity instead. The scheduler starts the subroutine tasks in a fixed and predefined order, according to their priority. The task may be preempted by other tasks, i.e. suspended but, in opposition to the coroutine approach, they run to completion.

7.1 Tasks and Their Priority

As mentioned above an undeniable advantage is that we can assign priorities off-line and choose the optimal strategy. We adopt a rate-monotonic priority assignment strategy [7], i.e. the priority of the task is proportional to its request rate. This fixed priority strategy simplifies the scheduling algorithms, and since the tasks are started in a deterministic way, the behavior of the system is easy to predict. Moreover, this strategy has been proven to be the optimal solution for fixed priority assignment (see [7]).

We have 3 types of tasks running in the system: tasks with 200 Hz rate, tasks with 50 Hz rate and background tasks. Tasks with 200 Hz rate are denoted as *Synchronous High Tasks* and have a high priority, tasks with 50 Hz rate are called *Synchronous Low Tasks* and have a medium priority, while background tasks are just *Tasks* and have a low priority.

The system implements four phases (0, 1, 2, 3) of 5 ms each. After phase 3 the system starts again from phase 0. High synchronous tasks are started at the beginning of each phase and preempt any tasks running with a lower priority. Low synchronous tasks, on the other hand, are started every 20 ms when all the high priority tasks are completed and preempt all background tasks. The phase in which the task is started is given by the parameter *startphase* in *InstallLowSync*.

A synchronous task can implement any complex computation. The only limitation is that it has to terminate before another synchronous task of the same priority can be started. This means that to guarantee the integrity of the system the computation time of all high sync tasks cannot be greater than 5 ms, and similarly the computation time of all the low sync tasks cannot be greater than 20 ms.

The background tasks are similar to the synchronous ones, the only difference being that they are started as soon the processor has terminated the handling of all synchronous tasks. The background tasks have the lowest priority, and can be preempted by every synchronous task.

```
DEFINITION HelyOS;

TYPE
   TaskCode =  PROCEDURE (me: INTEGER!);
   Task = POINTER TO RECORD
      proc: TaskCode;
      name: ARRAY 32 OF CHAR
   END;
   SyncCode = PROCEDURE (phase: INTEGER);
   SyncTask= POINTER TO RECORD
      proc: SyncCode;
      name: ARRAY 32 OF CHAR
   END;

   PROCEDURE InstallHighSync(s: SyncTask; VAR name: ARRAY OF CHAR);
   PROCEDURE RemoveHighSync(s: SyncTask);
   PROCEDURE InstallLowSync(s: SyncTask; VAR name: ARRAY OF CHAR;
                         startphase: INTEGER);
   PROCEDURE RemoveLowSync(s: SyncTask);
   PROCEDURE Install(t: Task; VAR name: ARRAY OF CHAR);
   PROCEDURE Remove(t: Task);

   PROCEDURE StartSync;
   PROCEDURE StopSync;
   PROCEDURE ExecTask;

END HelyOS.
```

7.2 Implementation and Performance

The scheduler implementation uses re-entrant interrupts. An interrupt signal is generated every 5 ms, which starts the interrupt handler *Scheduler*. This handler is the actual scheduler. It saves the processor status register (SPSR), the general purpose registers (R0..R12), the FP registers (FP0..FP7) and increments the phase counter (modulo 4). The other registers R13, R14 and R15 are special registers that are saved by the StrongARM processor automatically (see [3]). This is the prolog phase. Thereafter the scheduler has to enable the interrupt, since the processor disables it automatically during the interrupt call. The scheduler then starts the synchronous high and low priority tasks. When all the started tasks are completed, the interrupt handler stops the re-entrant interrupt to protect the epilog. In the epilog phase, it restores the registers,

the processor status and returns to the interrupted process, a background task or a synchronous low task. With the return the interrupts are automatically re-enabled.

The interrupt overhead is less then 8 μs, and is mainly due to the memory access time needed to store all the registers. Therefore one of the optimizations is to reduce this overhead by reducing the number of registers neededing to be saved. To do so the emulation strategy was changed. Normal registers are used for the emulation of the floating-point registers, reducing the number of register to be saved to the 12 general purpose registers and the processor status.

```
MODULE HelyOS;

   PROCEDURE [4] Scheduler;
   VAR
       fpe: ARRAY 16 OF INTEGER; (* local FP Register *)
       lr, spsr, localphase: INTEGER; list: SyncTask;
   BEGIN
      (* --------------- Prolog ------------------- *)
      STPSR(4, spsr);  (*store SPSR *)
      FPE.SaveFPR(fpe); (* save FP register *)
      PUT(HKernel.FPGAINTACK, 0); (* clear the fpga's int request *)
      phase := (phase + 1) MOD 4;
      localphase := phase;

      (* --------------- Scheduler ---------------- *)

      SetIRQ; (* enable re-entrant IRQ *)

      (* start all sync high priority task *)
      list:= synchigh;
      WHILE list # syncguard DO
         IF list.startphase = Always THEN list.proc(localphase) END;
         list := list.next
      END;

      (* start low priority tasks *)
      list:= synclow;
      WHILE list # syncguard DO
         IF list.startphase = localphase THEN list.proc(localphase) END;
         list := list.next
      END;

      ResetIRQ; (* disable re-entrant IRQ *)

      (* ---------------- Epilog ------------------ *)
      FPE.RestoreFPR(fpe);
      LDPSR(69H, spsr) (* SPSR := spsr *)
   END Scheduler;

END HelyOS.
```

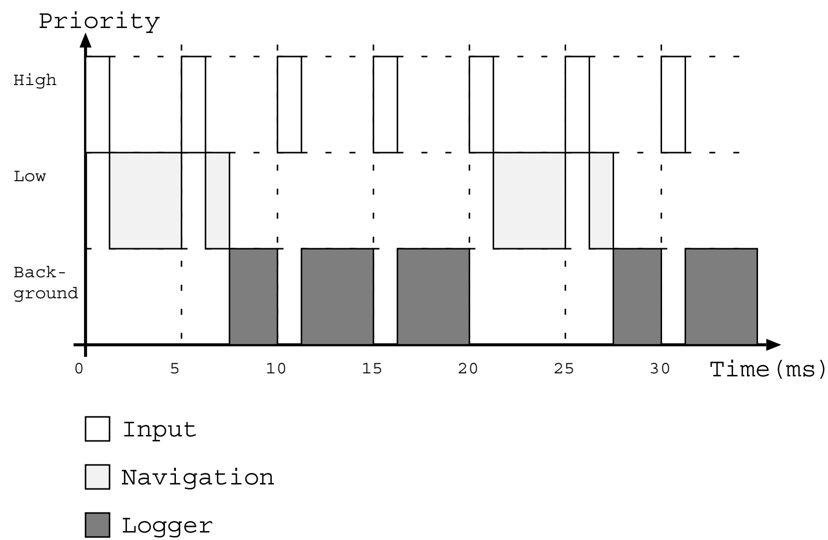


Figure 7.1: Scheduling Example

7.3 Example

Figure 7.1 shows a possible configuration of the system and its timing. The *Input* high sync task is started in each phase (every 5 ms) and has a high priority. The *Navigation* low sync task is started only in phase 0 and has a lower priority. Thus this task can be preempted by the *Input* task. The *Logger* is a background task that runs with the lowest priority. It is preempted by every synchronous task.

Chapter 8

Floating-point Emulation

The emulator[1] is completely integrated into the system, and its execution is transparent to the applications. The StrongARM processor generates an undefined instruction trap whenever its decoder decodes an unknown instruction. The undefined instruction trap handler decodes the unknown instruction and, if it is a floating-point instruction, executes it.

We chose to be compatible with the existing ARM floating-point instruction set and adopted the IEEE 754 floating-point format with 32 bits. Internally the emulator emulates 8 floating-points registers. Each register is represented with 8 bytes, 4 for the mantissa and 4 for the exponent.

8.1 Look-ahead Optimization

Since a floating-point instruction is usually followed by more FP instructions, the trap handler can reduce the overhead of the undefined instruction trap, if instead of a return it directly tries to decode the following instruction. Such a trap handler is slightly more complex (15 assembler instructions more) but 26% faster than the normal handler without this look-ahead feature. See appendix B for more details on the number of floating-point instructions needed and the number of sequential floating-point instructions.

8.2 Performance

Since the control of a helicopter uses floating-point operations very frequently, performance of such an emulation is a central concern in our project. The performance achieved with the current look-ahead solution is about 0.5 MFLOPS.

[1] The emulator was written by N. Wirth

8.3 Floating-point Library

To completely eliminate the trap overhead, the compiler was changed to call the floating-point library directly, instead of generating a floating-point instruction. This reduces the overhead of a context switch for each instruction. Moreover, the floating-point registers are normal integer registers, thus reducing memory accesses. This method is about 3 times faster than the emulation with traps. Its performance is 1.5 MFLOPS.

Chapter 9

Numerical Support

9.1 The Math Module

The Math module implements the following standard functions. The implementation takes advantage of certain features of the floating-point emulation routines.

```
DEFINITION Math;

   PROCEDURE arctan (x: REAL): REAL;
   PROCEDURE cos (x :REAL): REAL;
   PROCEDURE exp (x: REAL): REAL;
   PROCEDURE ln (x: REAL): REAL;
   PROCEDURE sin (x: REAL): REAL;
   PROCEDURE sqrt (x: REAL): REAL;

END Math.
```

9.2 The MatLib Module

This module implements matrix operations. The operations take advantage of the RIDER concept of the OberonSA compiler [14].

```
DEFINITION MatLib;

   PROCEDURE AddMat (VAR a, b, c: ARRAY OF REAL; n: INTEGER);
   PROCEDURE InvertMat (VAR a: ARRAY OF REAL; n: INTEGER);
   PROCEDURE MultMatMN1 (VAR a, b, c: ARRAY OF REAL; m, n: INTEGER);
   PROCEDURE MultMatMN3 (VAR a, b, c: ARRAY OF REAL; m, n: INTEGER);
   PROCEDURE MultMatMN5 (VAR a, b, c: ARRAY OF REAL; m, n: INTEGER);
   PROCEDURE ScalarMult (VAR a: ARRAY OF REAL; x: REAL;
                         VAR b: ARRAY OF REAL; n: INTEGER);
   PROCEDURE ScalarProd (VAR a, b: ARRAY OF REAL; n: INTEGER): REAL;
   PROCEDURE SubtractMat (VAR a, b, c: ARRAY OF REAL; n: INTEGER);
   PROCEDURE TransposeMatrix (VAR a, b: ARRAY OF REAL;
                              rows, cols: INTEGER)

END MatLib.
```

Chapter 10

The Startup Process

The first problem that has to be solved in a new system is how to start it. This may seem to be a small detail, but if it is not solved in a clean manner from the beginning it can turn into be a painful problem later. In this project we adopted the same strategy as that used in the Ceres System [15]: a small piece of code (less than 100 instructions) resident in ROM is executed after each reset or startup. It is responsible for loading the system core via the serial link or from the ROM and jumps to its module entry point. After some initialization, the bootloader tests a pin of the UART chip to decide if it will download the core via the link or from the ROM. If the boot pin is high then the ROM is selected, otherwise the serial link is used as source for the core. Once the core is loaded and started, it performs the initialization of memory and devices, then it starts the procedure *HelyOS.Loop* which waits for commands sent via the serial link. The protocol is the same for both types of download. In EBNF notation:

Boot = {size:4 adr:4 {word:4}} 0:4 startadr:4.

In case of a serial download, the host computer connected via the serial link is responsible for this unidirectional protocol. It sends a sequence of blocks of the size *size* that is then copied to the memory starting from the absolute address *adr*. The last block is characterized by the *size = 0* and terminates the protocol with *startadr* as the absolute entry address of the code to be executed.

The time needed to download the full system via serial link, i.e. downloading the software core, initializing the FPGA and all the devices, down-loading all the base modules and starting the control application, is less than 10 seconds.

Chapter 11

HelyOS Commands

The module HelyOS exports the following commands. As mentioned above, some of these commands accept a parameter string sent from the host system to the target via the call message.

```
DEFINITION HelyOS;

   PROCEDURE ShowModules;
   PROCEDURE ShowHeap;
   PROCEDURE ShowCommands;
   PROCEDURE ShowStatus;
   PROCEDURE Free;

   PROCEDURE Directory;
   PROCEDURE RemoteRead;
   PROCEDURE RemoteWrite;
   PROCEDURE MakePersistent;
   PROCEDURE ResetROMDisk;

   PROCEDURE ShowTasks;
   PROCEDURE StartSync;
   PROCEDURE StopSync;

   PROCEDURE RESET;

END HelyOS.
```

The command *ShowModules* lists all loaded modules, their size and their reference count. *ShowHeap* lists all free heap blocks and their sizes. *ShowCommands* lists the names of all commands implemented by a module. *ShowStatus* shows the value of all global variables of a module. Last but not least, *Free* unloads a module.

The other commands are used to interact with the file system. *RemoteRead* and *RemoteWrite* allow the Olga computer to access the host file system. The *MakePersistent* command moves a file from the RAM disk to the ROM disk. Since the ROM disk is implemented in a FlashROM we cannot selectively delete files from the disk but can only delete the complete structure with *ResetROMDisk*.

The task system can be started or stopped with the *StartSync* and *StopSync* commands. A list of all installed tasks can be displayed with the command *ShowTasks*. The system can be restarted with the *RESET* command.

Chapter 12

HLog Module

This module allows the target system to write simple strings on the host system. The host system automatically opens an Oberon Viewer, where all the messages sent by the target are displayed.

```
DEFINITION HLog;

   PROCEDURE Ch (ch: CHAR);
   PROCEDURE Clear ();
   PROCEDURE Hex (h: LONGINT!);
   PROCEDURE Int (i: LONGINT);
   PROCEDURE Ln ();
   PROCEDURE Real (r: REAL);
   PROCEDURE Str (VAR s: ARRAY  OF CHAR);
   PROCEDURE Time ();

END HLog.
```

Appendix A

FPGA

This is the Lola [11] program that describes the hardware implemented in the FPGA. The FPGA was implemented using the Trianus [4] and Hades [8] tools. Figure A shows the placement of the varius circuits.

```
MODULE Servo; (* ms, 18 Jul 97 *)

  (* Note: SR not used since Trianus doesn't
     accept loop connections without labels
  *)

  (* GClk = 50MHz; G1 = 3.6864MHz; G2 = 230.4 KHz *)

  IN
    INTACK', GClk, G1, G2, RTF : BIT;
    PWMIn : [6] BIT;

  OUT
    INT', F200', F50', Clk : BIT;
    PWMOut : [6] BIT;

  VAR
    pscale, pscalec: [4] BIT;

    freq200, freq200c: [11] BIT; F200rst' : BIT;
    intsr: BIT;

    cntr20, cntr20c: [13] BIT; cntr20rst' : BIT;
    outcmp, oactive, syncoactive, pwmsr, pwmrs: [6] BIT;
    outreg, syncoutreg: [6][9] BIT;

    incntrst', incntsync: [6] BIT;
    incnt, incntc, inreg: [6][9] BIT;

    rtfrst', rtfsync: BIT;
    rtfcnt, rtfcntc, rtfreg: [18] BIT;

BEGIN
```

```
(* ------------- prescaler -----------------------------------------*)

  pscale.0 := REG(G1: ~pscale.0); pscalec.0 := pscale.0;
  pscale.1 := REG(G1: pscale.1 - pscalec.0);
  pscalec.1 := pscalec.0*pscale.1;
  pscale.2 := REG(G1: pscale.2 - pscalec.1);
  pscalec.2 := pscalec.1 * pscale.2;
  pscale.3 := REG(G1: pscale.3  - pscalec.2);
  pscalec.3 := pscalec.2 * pscale.3;
  Clk := pscale.3;


(* ------------- Interrupt generator  ------------------------------ *)

  freq200.0 := REG(G2: ~freq200.0 * F200');
  freq200c.0 := freq200.0;
  FOR i := 1 .. 10 DO
    freq200.i := REG(G2: (freq200.i - freq200c[i - 1]) * F200');
    freq200c.i := freq200.i * freq200c[i - 1];
  END;
  (* freq200=1151  200Hz Pulse *)
  F200rst' :=~freq200.0 + ~freq200.1 + ~freq200.2 + ~freq200.3 +
        ~freq200.4 + ~freq200.5 + ~freq200.6 + freq200.7 +
         freq200.8 + freq200.9 + ~freq200.10;

  F50' :=  cntr20rst' ;    (* 50 Hz Pulse *)
  F200' := F200rst';    (* 200Hz Pulse *)

  (*INT' := SR(~F200', ~INTACK'); *)
  INT' := ~(INTACK' * intsr);
  intsr := ~(F200' * INT');


(* ------------- pwm outputs --------------------------------------- *)

  (* main counter :
     20ms @ 230.5kHz = 4608 -> if cntr = 4607 restart cntr
  *)
  cntr20.0 := REG(G2: ~cntr20.0 * cntr20rst');
  cntr20c.0 := cntr20.0;
  FOR i := 1 .. 12 DO
    cntr20.i := REG(G2: (cntr20.i -  cntr20c[i-1]) * cntr20rst');
    cntr20c. i := cntr20.i * cntr20c[i-1]
  END;
  cntr20rst' := ~cntr20.0 + ~cntr20.1 + ~cntr20.2 + ~cntr20.3 +
        ~cntr20.4 + ~cntr20.5 + ~cntr20.6 + ~cntr20.7 +
        ~cntr20.8 + cntr20.9 + cntr20.10 + cntr20.11 + ~cntr20.12;

  (* 6 pwm outs *)
  FOR  i := 0 .. 5 DO
    oactive.i := REG(G2: MUX(cntr20rst':
                    REG(GClk: syncoactive.i), oactive.i));
    FOR j := 0 .. 8 DO
      outreg.i.j := REG(G2: MUX(cntr20rst':
```

```
                    REG(GClk: syncoutreg.i.j), outreg.i.j))
    END;
    outcmp.i := (outreg.i.0 - cntr20.0) + (outreg.i.1 - cntr20.1) +
          (outreg.i.2 - cntr20.2) + (outreg.i.3 - cntr20.3) +
          (outreg.i.4 - cntr20.4) + (outreg.i.5 - cntr20.5) +
          (outreg.i.6 - cntr20.6) + (outreg.i.7 - cntr20.7) +
          (outreg.i.8 - cntr20.8);
    (* PWMOut.i := oactive.i * SR(~cntr20rst', ~outcmp.i) *)
    pwmsr.i := ~(cntr20rst' * pwmrs.i); pwmrs.i := ~(outcmp.i * pwmsr.i);
    PWMOut.i := oactive.i * pwmsr.i;
  END;


(* ------------- pwm inputs ---------------------------------------- *)

  FOR i := 0 .. 5 DO
    incnt.i.0 := REG(G2: (incnt.i.0 - PWMIn.i) * incntrst'.i);
    incntc.i.0 :=  incnt.i.0 * PWMIn.i;
    inreg.i.0 := REG(G2: MUX(incntrst'.i: incnt.i.0, inreg.i.0));
    FOR j := 1 .. 8 DO
      incnt.i.j := REG(G2: (incnt.i.j - incntc.i[j-1]) * incntrst'.i);
      incntc.i.j := incnt.i.j * incntc.i[j-1];
      inreg.i.j := REG(G2: MUX(incntrst'.i: incnt.i.j, inreg.i.j))
    END;
    incntsync.i := REG(G2: PWMIn.i);
    incntrst'.i := PWMIn.i + ~incntsync.i;
  END;


(* ------------- rotor frequency ----------------------------------- *)

  rtfcnt.0 := REG(G2: ~rtfcnt.0 * rtfrst');
  rtfcntc.0 :=  rtfcnt.0;
  rtfreg.0 := REG(G2: MUX(rtfrst': rtfcnt.0, rtfreg.0));
  FOR j := 1 .. 17 DO
    rtfcnt.j := REG(G2: (rtfcnt.j - rtfcntc[j-1]) * rtfrst');
    rtfcntc.j := rtfcnt.j * rtfcntc[j-1];
    rtfreg.j := REG(G2: MUX(rtfrst': rtfcnt.j, rtfreg.j))
  END;
  rtfsync := REG(G2: RTF);
  rtfrst' := ~RTF + rtfsync;

END Servo.
```

Figure A.1: FPGA Design

Appendix B

Helicopter System Modules

The following tables list the modules of the system, along with the code size (column Code) in KBytes, data size (column Data) in KBytes, the number of floating-point instructions (column Fp.) and the number of sequential floating-point instructions (column Seq. Fp.), i.e instructions following other floating-point instructions.

Module	**Size**	**Data**	**Fp.**	**Seq. Fp.**
ADC	451	35	203	157
FPE	556	79	0	0
HelyOS	1595	239	0	0
HFileDir	1391	15	68	0
HFiles	1345	7	1	0
HLog	126	7	1	0
HModules	672	43	7	0
HROM	296	71	0	0
Math	311	7	184	150
MatLib	727	15	143	54
Servo	320	15	3	0
Total	7790	533	610	361

Table B.1: Software Core Modules

Module	**Size**	**Data**	**Fp.**	**Seq. Fp**
BootBurner	436	7	0	0
CoreBurner	273	7	0	0
FPGABurner	474	11	0	0
Total	1183	25	0	0

Table B.2: Utility Modules

List of Figures

1.1 Software Core 2

2.1 Memory Map 3
2.2 Memory Organization 4
2.3 Interrupt Vector Table 6

4.1 Multiplexer 12
4.2 Pulse Width Modulation Signal 12

5.1 Module Structure 17

7.1 Scheduling Example 24

A.1 FPGA Design 35

Bibliography

[1] M. Aeschlimann. Oberon0 system for the it. Semester work, ETH Zurich, Institute for Computer Systems, June 1997.

[2] Digital Equipment Corporation. *Memory Management on the StrongARM SA-110: Application Note*, 1998.

[3] Digital Equipment Corporation. *SA-110 Microprocessor Technical Reference Manual*, 1998.

[4] Stephan W. Gehring. *An Integrated Framework for Structured Circuit Design with Field-Programmable Gate Arrays*. PhD thesis, ETH Zurich, 1997.

[5] M. Kottmann. A computer system for model helicopter flight control, technical memo nr. 4: The auto pilot software. Technical report, ETH Zurich, Institute for Computer Systems, February 1998.

[6] M. Kottmann and J. Chapuis. Specification of communications protocols. Internal Memo.

[7] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20:46–61, 1973.

[8] Stefan H.-M. Ludwig. *Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor*. PhD thesis, ETH Zurich, 1997.

[9] Maxim Integrated Products. *MAX196/Max198 Multirange, single +5V, 12-Bit DAS with 12-Bit Bus Interface*, 1995.

[10] Philips. *SC28L194 Quad UART with TTL Compatibility at 3.3V Supply Voltage*, 1995.

[11] N. Wirth. Lola system notes. Technical Report 236, ETH Zurich, Institute for Computer Systems, June 1995.

[12] N. Wirth. Tasks vs. threads: An alternative multiprocessing paradigm. *Software-Concepts and Tools*, 17:6–12, 1996.

[13] N. Wirth. A computer system for model helicopter flight control, technical memo nr. 1: The hardware core. Technical Report 284, ETH Zurich, Institute for Computer Systems, January 1998.

[14] N. Wirth. A computer system for model helicopter flight control, technical memo nr. 2: The programming language oberon sa. Technical Report 285, ETH Zurich, Institute for Computer Systems, January 1998.

[15] N. Wirth and J. Gutknecht. *Project Oberon*. Addison-Wesley, 1992.

[16] Xilinx. *XC6200 Field Programmable Gate Arrays*, 1996.