Computer Science

Fredric Hellberg, Daniel Westerberg

Operating System for a MC68000 Based Microcomputer

Bachelor's Project

2004:21

Operating System for a MC68000 Based Microcomputer

Fredric Hellberg, Daniel Westerberg

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Daniel Westerberg

Fredric Hellberg

Approved, 2004-06-03

Advisor: Thijs Holleboom

Examiner: Stefan Alfredsson

Abstract

This report describes the results after building a MC68000 based microcomputer and constructing an operating system for it. The function, implementation and usage of the various parts in hardware and software are described. The hardware is not described in detail as it was designed previously to this project. Focus is instead directed towards the operating system written. Usage as well as design and implementation of the various parts of the OS is covered.

Contents

List of Figures

List of Tables

1 Introduction

This report describes the construction of an operating system for a microcomputer based on an MC68000 processor. In section 2 the background of the project is described and in section 3 the purpose and goal of this work is covered. Section 4 describes the hardware and its various parts. Section 5 gives an overview of the operating system, followed by a more detailed view of the parts that makes up the OS in the following sections. Appendix contains guiding information, such as structures, system calls and abbreviations, that is referred to in various places in the report.

2 Background

This project started as a hobby project by Daniel in 1998. He got the interest to build a computer after a course in micro computing at the swedish semi-equivalent to high school, gymnasiet, where a Z80 based computer was constructed.

When we started to look for a bachelors project Fredric had a contact with an external company. In the last minute, however, they realized they would not have enough time to coach us through their project. At this time Daniel had the idea to maybe take up this old hobby project as a reserve plan.

We started to evaluate and elaborate the idea to construct a microcomputer. As the design of the actual computer was already made, the main part of this project was to write the operating system for it. After evaluation we both felt that this could be a very interesting project. We were also running out of time and needed to settle with a project immediately. We were told that there are already a lot of this kind of microcomputers out there and that we would save a lot of time and effort by buying one instead of building one, but where is the fun in that?!

3 Purpose and goal

The goal is to build an expandable and easily debuggable MC68000 based microcomputer with an RS232 serial interface. For this computer an operating system will be designed. The goal is to have a small but usable OS with basic input and output functions through a serial interface. It will also have a memory manager from which memory can be arbitrary allocated and freed. The goal is however not strict and may be altered, extended or cut. Time will decide.

The purpose is for us to learn through the experience what problems that may occur and what their solutions might be. The construction of the operating system is the main part of this project.

We intend to start by making the hardware. This will have to be made in less than 30% of the total time of the project to be able to call it a computer science project and not an electric engineering project. When the hardware has been finalized the software will be written starting with startup code and serial routines to be able to communicate with the computer as quickly as possible. Later the other parts, such as a memory manager, will be written.

4 The hardware

This section describes all significant parts of the computer which mostly consists of the main board, often also called the mother board. The electronic schematic design of the computer was made several years ago, which left only the layout, etching and soldering of the circuit boards as part of this project. Because of this, no detailed description of the electronic design process is made in this report.

4.1 Overview

The term ”microcomputer” is nowadays used for these kinds of, usually embedded, computers which typically are quite small and have no, or only a small, graphical display. But actually almost all computer used today are microcomputers. In the old days, the 1960’s and 1970’s, there were also other types of computers, namely mainframes and minicomputers. A mainframe usually occupied several rooms, or even floors, in a building. They were used as central servers for storing and processing large (by that time) quantities of data. Then there were minicomputers. These only filled one small room or was not bigger than a refrigerator. They were mostly used as local central servers at companies. Later, the computers shrunk and were available in sizes small enough to keep on a desk. These were called microcomputers. PC’s is one typical example of such an original microcomputer.

This computer consists of a central processing unit, two kinds of memory; RAM and ROM, address decoding logic, interrupt logic, timer logic, expansion connectors and a serial circuit. Figure 4.1 shows an overview over the computer. The components in this figure are placed approximately as they are placed on the actual computer. The computer was named *dBOX* selfishly decided by Daniel, severaly years before the project, as the designer of the actual microcomputer.

4.2 CPU

The core of the microcomputer is the CPU which is a Motorola MC68000. It is a so called 16/32-bit CPU because it has a 16-bit data bus but is fully 32-bit internally. The address bus is 24 bits wide providing an address range with a total of 16MB. The choice to use this CPU was quite simple as Daniel, the designer of the computer, already had a lot of experience with this CPU from the Amiga range of computers. It was also a good choice because it is easy to program with, as it has sixteen 32-bit general purpose registers, but at the same time relatively simple to build as it only has a 16-bit data bus and a 24-bit

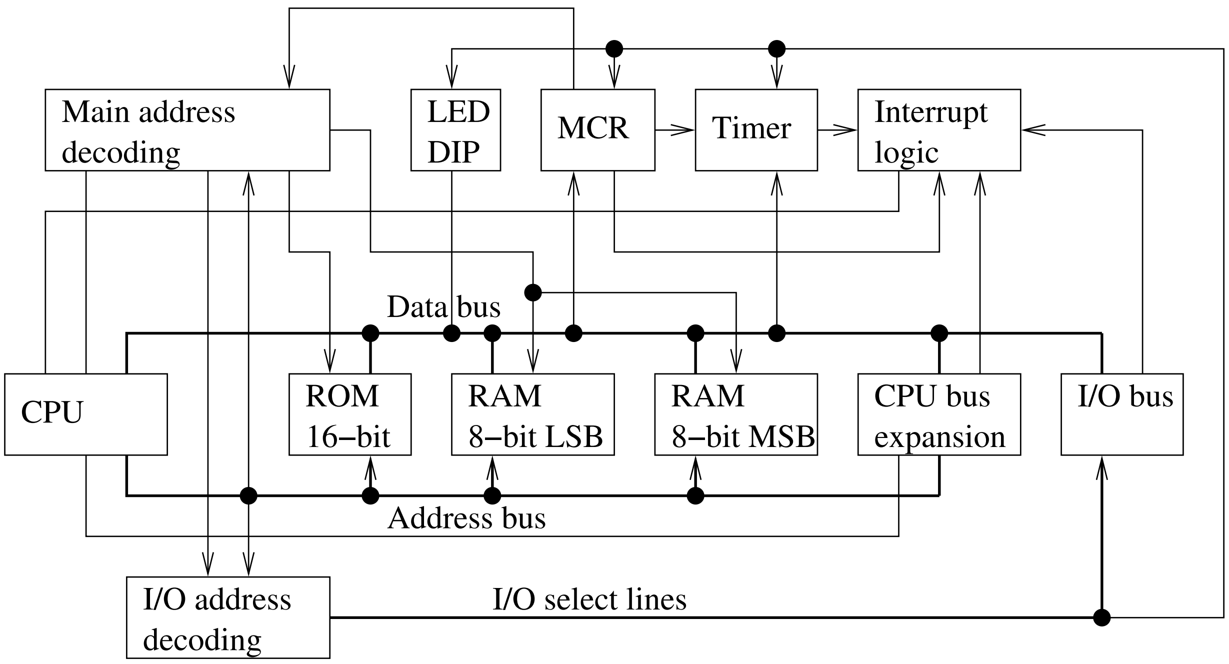

Figure 4.1: Overview of dBOX (Connections without arrows are bidirectional)

address bus.

4.3 Address decoding

The CPU does not distinguish between RAM, ROM and I/O addresses. It has one unified address space for everything. A few other CPUs distinguish between memory and I/O adresses but this CPU does not. For the CPU to be able to address any such specific addresses the main board must have external logic to be able to make this differentiation. This logic is generally called address decoding logic.

The main address decoding is done by a 74HC138 IC which divides the lower 8MB of the total address range into blocks of 1MB each. The reason to divide the memory into one lower and one upper half of the memory space was thought to be a good idea because then dBOX could have one large block of memory space, to be used by for example a large memory module, and a few smaller blocks to be used by for example smaller memory modules or other expansion devices. ROM must begin at address 0 and to keep the area for ROM small the lower 8MB were chosen to be divided into the 1MB blocks previously mentioned. The lowest megabyte is dedicated to ROM. The second is dedicated to RAM. The third is dedicated to I/O. The rest is dedicated to expansion devices. See figure 4.2. A size of 1MB per block was a reasonable size because the demultiplexers on the market is either 3 to 8- or 4 to 16-ways[1] and 8MB divided by 8 makes the size 1MB which seemed like a nice even number. The upper 8MB of the total address range is considered as one block of addresses dedicated to an expansion device, preferably a memory module.

4.4 Memory

There are two types of memory on the main board, a read-only memory of type EPROM and a read/write memory of type SRAM. The main board features a function to logically swap these areas of memory. The reason for this is that when the computer starts the ROM needs to be at address 0 as the CPU will start to read at this address. Later however a program might need to be able to change an interrupt vector, which are all located between address 0 and 1023, then RAM needs to be there. In this case a copy of at least the lowest 1024 bytes of data and the actual swap code in the ROM needs to be copied into the equivalent addresses of the RAM before the swap.

4.4.1 EPROM

This is the read-only memory, ROM, in which the bootstrap program, console, memory manager, interrupt handlers and serial routines reside. It consists of one 16-bit 128kB IC exchangable to 256kB. It is located at address 0 when it is not swapped with the RAM, see subsection 4.4.

[1]A 3- or 4-bit binary number decides which one of 8 or 16 pins should go active.

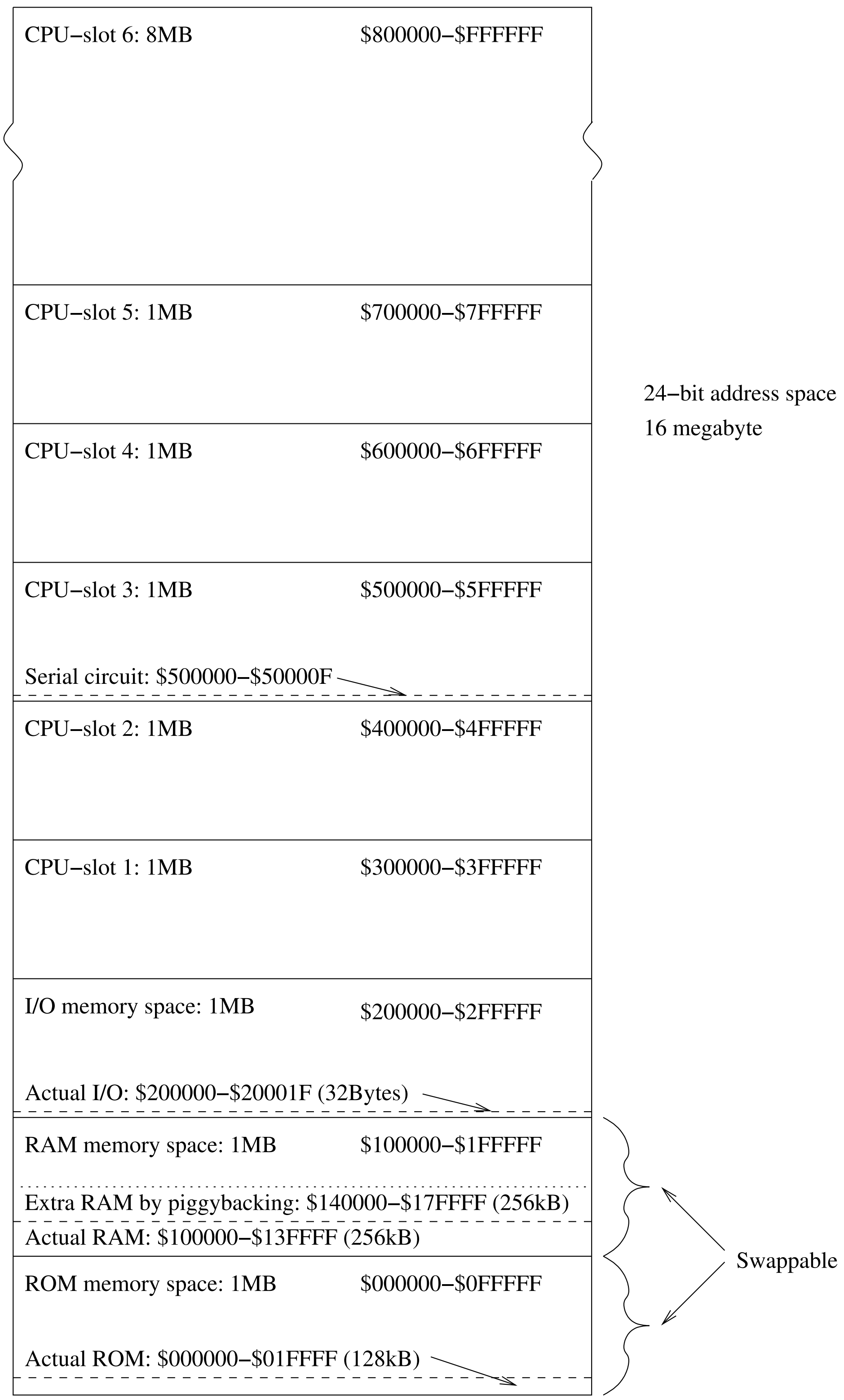

Figure 4.2: The address space in dBOX

4.4.2 RAM

This is the read/write memory, RWM or RAM. It is empty when the power is turned on but keeps its memory during a reset of dBOX. It consists of two 8-bit 128kB ICs which each handles one half of the 16-bit data bus, even and odd addresses, making a total of 256kB. It is easy expandable to 512kB by piggybacking[2] two equal ICs on top of the existing ones. See figure 4.2. This is done by connecting one chip-select pin of the new memory ICs to the X-Ram pin on the main board.

In case 512kB of memory would not be enough, another 13MB of memory can be connected via the expansion busses.

4.5 Input and output ports

The main board decodes 32 bytes for use by simple I/O-ports. Four bytes are allocated on the main board for a low frequency interrupt timer, a master control register, an 8-bit DIP-switch for input and 8 LEDs for output. The remaining 28 I/O bytes are unused on the main board. 24 of the remaining 28 I/O-ports are dedicated to the I/O bus expansion. The I/O bus expansion is a 34-pin ribbon cable connector, typically a standard non-twisted floppy cable. It has all 16 bits of the data bus and 12 select bits able to address one word each. It also has one interrupt line connected to it. For detailed pinout of this connector, see appendix C.

4.5.1 Main board I/O-ports

Following is a description of the bits in the four I/O-ports on the main board. I/O-ports are not an entirely correct description for some of these ports as they do not all communicate with the surrounding world, which is the usual concept of I/O-ports. Control registers might be a more suitable word but as they are in the same address range as the other

[2]A way of connecting a device directly on top of another.

I/O-port addresses they are still generally referred to as such. The three output ports have latches to keep their value after writing to them. They keep their value until written to again or power is turned off. Table 4.1 contains a brief description of the ports.

I/O-port / Register	Address	Description
Low-frequency clock (LFC)	$20001C	Interrupt timer settable between 0.15-38Hz
Master Control Register (MCR)	$20001D	Swap, Interrupt inhibit, CPU clock
Input DIP-switch (DIP)	$20001E	8-bit user input switch
The 8 LEDs (LED)	$20001F	8-bit user output LED indicator

Table 4.1: I/O-ports on the main board

LFC This register cannot be read. A copy of its value should therefore always be present in the environment variables. See appendix D. If read, an undefined value is returned but the register stays intact.

Bit 7 in this register inhibits the LF-clock. The other seven bits are used to set the speed of the clock. Setting the register to 0 places the clock in a constant timeout situation and IRQ 3 will always be set causing the computer to hang unless handled by the interrupt routine. Values from 1 to 127 give time intervals in seconds according to the following formula:

$$ti = \frac{1}{\frac{5000000}{2^{18}}} * value - \frac{1}{\frac{5000000}{2^{17}}}$$

where *value* is the value written to the LF-clock and *ti* is the time interval in seconds. Simply put:

$$mti = value * 52.6 - 26.3$$

where *mti* is the time interval in milliseconds and *value* is the value written to the LF-clock. The highest frequency is therefore ≈38Hz. The counter was first intended to be run at 20MHz but during construction it was revealed that it could not run faster than about 5MHz. This is why the highest frequency that can come out of it is not more than about 38Hz.

MCR This register cannot be read. A copy of its value should therefore always be present in the environment variables. See appendix D. If read, an undefined value is returned but the register stays intact.

Bit 0 controls the RAM<–>ROM swap. Setting bit 0 to 1 instantly causes RAM to be placed at address $0 and ROM at address $100000. Clearing bit 0 to 0 instantly causes RAM to be placed at address $100000 and ROM at address $0.

Bit 1 is interrupt inhibit. Setting this bit to 1 blocks all interrupts to the CPU, including the non-maskable IRQ 7. Clearing this bit enables interrupts. Interrupts that occured during the inhibit and that were latched will immediately be signalled to the CPU.

Bit 2 is LF-clock interrupt inhibit. Setting this bit to 1 blocks the interrupts from the LF-clock to the CPU. The clock will not stop counting. Clearing this bit enables LF-clock interrupts to the CPU providing that bit 1 is not set.

Bit 3 and 4 are unused.

Bit 5-7 are used to set the CPU clock speed. Table 4.2 shows the speed with different values. If using an external CPU clock source it must not run slower than about 20-50kHz as the CPU gets unstable at lower clock frequencies. If debugging requires a lower speed it is recommended to set the CPU clock to 1.25MHz and gate off the data acknowledge (DTACK) signal from the address decoding logic instead to make the CPU work slower. A device was designed for this purpose when debugging the hardware and software. See section 14 on page 41 for more information.

DIP This is a physical 8-bit switch on the main board that can be used to input user data. If the switch is removed, an external signal source may be fitted to supply input. This input port cannot be written to. If written to, nothing will happen.

Bit 7,6,5 (dec)	Speed
000 (0)	1.25 MHz
001 (1)	2.5 MHz
010 (2)	5.0 MHz
011 (3)	10 MHz
100 (4)	1.25 MHz
101 (5)	External
110 (6)	20 MHz
111 (7)	1.25 MHz

Table 4.2: CPU speed values in MCR

LED This is an output port connected to 8 LEDs to form a simple data output to the user. This output port cannot be read. An undefined value is returned if read, but the port's value stays intact.

4.6 Expansion busses

The main board decodes six blocks of addresses which can be used by six different expansion devices. Expansion devices are connected to the main board through the expansion slot which is a 50-pin ribbon cable, typically a standard SCSI cable. It has all 16 bits of the data bus, 19 bits of the address bus capable of addressing 512k-word which is equal to 1MB, 3 interrupt lines and full control bus to make it possible for expansion devices to master the bus for DMA purposes. To see detailed pinout of this connector, see appendix C. In this project we have only one expansion connected to this bus and that is the serial port. To be able to address the complete 8MB block, the address lines A20-A22 can be added separately by an additional connector.

4.7 Interrupt handling

To be able to keep programs running in parallel to external hardware such as the serial port and the timer without the need to poll these devices, some kind of interrupt system must be used.

The CPU has seven levels of prioritized interrupts where interrupt request 7 (IRQ 7) has the highest priority and IRQ 1 has the lowest priority. An interrupt value of 0 indicates that there is no pending interrupt request to the CPU. The CPU has three lines to represent one of the seven levels. To convert the seven lines into 3 lines a small card, that is connected to the main board, was designed to do this job. This card also has SR flip-flops on it to latch short interrupts long enough for the CPU to catch them. When the CPU responds to an interrupt request, logic on the interrupt card resets the corresponding SR flip-flop.

The seven interrupt lines are hard-wired to various things on the main board. IRQ 1 and 7 are connected to two switches directly on the main board which are used together with the DIP-switch to give simple command to the microcomputer during debug when no other I/O unit, like the serial circuit, is connected to dBOX. IRQ 2, 4 and 6 are connected to the expansion bus for use by external devices. IRQ 3 is connected to the low frequency timer. IRQ 5 is connected to the I/O bus.

5 Operating system overview

A computer is never useful without software to execute on it. In order to be able to execute software on a computer there must be a way of getting it into the computer. In the simplest case a single program is put into a ROM. This program starts to execute as soon as the computer is powered on. Digital watches, microwave owens and pocket calculators are made this way. If the usage of a computer is not completely specified when it is constructed it is required that it can change program after it is constructed and while it is running. To make this possible a general piece of software needs to be in the computer from the beginning that can be used to load other programs into it to execute. The simplest case is a plain bootloader that can do nothing except load a program into memory and execute this program. However; to be able to have more than one program in memory at the same time, to be able to execute more than one program at the same time and to provide a layer of abstraction from the actual hardware that the programs can benefit from, an operating system is needed.

This OS was decided to be named *ExOS* which derives from the swedish short term for Bachelors Project: ”Exjobb”.

5.1 ExOS

dBOX features an operating system called ExOS. ExOS is a fully 32-bit operating system written mostly in the assembly language. It can maintain up to 4GB of memory, although the CPU currently used in dBOX can not. All pointers and addresses maintained are 32-bit. The OS has the ability to maintain any number of programs running at the same time. It has hardware abstraction for the memory, serial circuit and the timer. Most of the OS was written using assembly language.

As the OS is located in a read-only memory it would be difficult for the various parts of the OS to store variables needed to maintain the state of the parts as they all would need to keep their own static memory address for these variables. This violates the concept of a dynamic unified managed memory. Therefore ExOS has one unified place in RAM where all different parts can store and get information about the state of the OS. This place is called the environment variables, also known as *envvars*. A function is used to retrieve a pointer to envvars. Envvars contains, among other things, base pointers to the timer device, program manager, kernel and memory manager. It also has snap-in function pointers that can be used to connect a user function to any interrupt without the need to swap ROM and RAM and change the actual interrupt vectors. See appendix D for more information about the environment variables.

When power is turned on, ExOS will: Initiate the environment variables by copying them to RAM, initiate the serial circuit and set it to 9600bps 8N1, initiate the memory manager and initiate the kernel to allow the concept of processes to exist. When this has been done a welcome message is printed to the serial and the primitive console starts.

6 System calls

ExOS features two types of system calls. The first type uses the TRAP instruction of the CPU. A TRAP instruction causes an exception, also known as a software interrupt. There are 16 TRAP instructions of which a few have been assigned to critical OS functions such as the memory manager, the timer and rescheduling functions. Functions that need to inhibit interrupts or perform other priviledged instructions must be called using a TRAP instruction as this is the only way a user program can execute priviledged code.

The second type uses plain subroutine calls. They are used for non-critical OS calls and support functions such as the built-in memory checker, serial functions and support string handling functions. The number of non-critical OS calls is only limited by storage in the micro computer system, unlike the TRAP kind of call which there are only 16 of. The idea to use one universal TRAP call for all functions where one of the CPU registers decides what function should actually be called was disregarded for three reasons: First; one register must always be kept clean to be usable for selecting the proper system function in case of a call which might not be desirable. Secondly; there will be an unnecessary overhead before each call to place the correct function selector in the register, plus the overhead for the universal TRAP to select the proper subroutine to be called. Thirdly; it is not such a good idea to execute in supervisor mode more than necessary as rescheduling can not occur. See section 7 on page 21 for more information about rescheduling.

The naming of the two types of system calls differ to be able to distinguish between them. TRAP calls are all in lower case. The second kind have the first letter in every word in the name capitalized (e.g. StrCopy).

In the report, however, when refering to a system call of any kind, both types of calls have the first letter capitalized to make it obvious that a function call of some kind is referred to.

Registers are used to pass arguments to system functions. The stack is not involved in parameter passing or for return values. Every function defines the registers that it use for input parameters, if there are any. Input parameters signifying values or data generally uses data registers. Input parameters signifying an address or a pointer generally uses address registers. The first data parameter generally uses D0, the second uses D1, etc. The first address or pointer parameter generally uses A0, the second uses A1, etc. There may be exceptions though.

If a function returns a value, a register is used also here. The data register D0 is used for functions returning values or data. Address register A0 is used when an address or pointer is returned. Functions may return more than one value. Each function defines which registers that are used. All system functions preserve all registers. The caller of a system function can always be sure that no register has been altered after a call to a system function, except for registers carrying possible return values. If a return value is specified to return a word smaller than the size of the register, which is 32 bits, only the specified size of the register will be valid. Example: D0.B specifies that only the eight least significant bits are valid as data. However, the rest of the register may have been altered and not restored by the function and must be considered undefined. If some part of the OS is set to call a user function, the user must preserve all registers used unless otherwise clearly noted.

6.1 TRAP calls

This subsection summarizes all system calls that uses the TRAP instruction. Only a brief description is given. Refer to the corresponding section mentioned, if any, for a more detailed description of how a particular function is implemented. See appendix E for the complete usage with parameters and return values.

allocmem This function is used to allocate memory for use by a user program or some other part of the OS. Any size may be requested but may be rejected if the amount of free memory is not available. See section 8 on page 27 for more information.

freemem This function is used to free memory allocated by AllocMem. See section 8 on page 27 for more information.

timer This function is used for all calls to the timer device. A parameter is used to define what timer function should be called. See section 9 on page 30 for more information.

reschedule This is a private OS function that may not be called by user programs. It is used by Block and Signal to do the actual rescheduling of processes. It is required that the stack is pure when called. See section 7 on page 21 for more information.

stop This call executes the STOP instruction which halts the CPU and makes it cease instruction execution and wait for an interrupt. User programs should use the syslist call Block if they have nothing to do to let other processes execute while waiting for an interrupt. It is however safe for user programs to call this function. It is mainly used by Block if the kernel has not been initiated.

supervisor This function is used to check if the calling code is executing in supervisor mode or not. Mainly used by the OS to determain if a rescheduling can be done or not.

swapromram This function sets up everything necessary to be able to safely swap the logical addresses of RAM and ROM in the computer which is necessary to be able to change the interrupt vectors in the low address range from address 0 to 1023.

debug This function can be used while debugging. It simply prints the current program counter position, the current status register, user- and supervisor stack pointer to the serial port.

usertrap This function takes a function pointer as an address that it immediately jumps to. As TRAPs execute in supervisor mode, a user program can through this function make a piece of its own code execute in supervisor mode. Regular programs may have little use for this but maintenance software may need this functionality.

reset This function emulates a reset on the computer. As the CPU has no instruction to perform a real reset on itself, this function provides an emulation of a real reset. It reads the reset vector from the memory and jumps to the start address just as the CPU does after a hardware reset. The startup code will not know the difference between an emulated or real reset. The RESET instruction is executed to reset external hardware that may use the reset pin on the CPU.

6.2 System list calls

This subsection summarizes all system calls that uses the system list functions, also known as the *syslist*. Only a brief description is given for each function. Refer to the corresponding section, if there is any, for a more detailed description of how a particular function is implemented. See appendix E for the complete usage with parameters and return values.

Unlike the TRAP calls, syslist functions do not have a given location where they can be called. They can be placed virtually anywhere in the system memory. To make it as simple as possible, a decision was made to use the same address as the startup code which pointer is always located at address $4 in the memory space. To not collide with the actual startup code the syslist is located at decreasing addresses from the startup address while the startup code is located at increasing addresses. To save one line of dereferencing, a jump table is used instead of a plain array of function pointers. This way a calling program can simply jump directly into the table which in turn jumps to the correct location, instead of first getting the address to an operation from the array and then perform the jump to that address. The idea for this was borrowed from the Amiga Operating System which uses this very technique for all kinds of operating system and third party library function calls.

If the RAM and ROM has been swapped, the pointer to the jump table, and possibly even the table itself, would be located in RAM. In this case it would be very easy to update a particular function simply by altering a specific location in the jump table, or constructing a completely new jump table for the system to use.

Block This function is called by user programs that do not have anything to do for the moment. Block returns when the calling process gets a signal matching one of the signals sent in to Block to wait for. See section 7 on page 21 for more information.

Signal This function is used to send a signal to a process. If the other process is currently waiting for this signal in a call to Block, it immediately starts executing. See section 7 on page 21 for more information.

Connect This is used to connect a device, for example the timer device, to an event handler. See section 7 on page 21 for more information.

Event This function is used to signal that an event has occurred. An event can either call a connected handler or send a signal to the process requesting knowledge about the event. See section 7 on page 21 for more information.

AllocSignal Allocate a signal for use by a user program to associate with a system event, such as a timer so that the user program can go to sleep when it has nothing to do and then get woken up by this signal when an event occurs. There are 24 signals available for allocation for each process. See section 7 on page 21 for more information.

FreeSignal Return a signal previously allocated. See section7 on page 21 for more information.

SingleTask With this function user programs can request to not be scheduled out of execution. If a user program is manipulating sensitive shared memory or is executing time sensitive code for a short period of time this function ensures that no other process will execute until multitasking is turned on again. Interrupts however, continues to arrive. If a call to Block is made with multitasking prohibited it will immediately be enabled again.

Delay This function is used for making delays without using a hardware timer. It checks the current speed of the CPU and adapts a busy loop to take the requested amount of time. It takes two parameters, seconds and microseconds. The delay is not accurate down to micro seconds and should mainly be used for debugging purposes. If multitasking is enabled the delay is never shorter than the given time but may be significantly longer.

InitSerial This function is used mainly by the startup code to initiate the serial circuit. See section 10 on page 33 for more information.

SendS This function is used to send data synchronously to the serial port. See section 10 on page 33 for more information.

SendA This function is used to send data asynchronously to the serial port. See section 10 on page 33 for more information.

GetS This function is used to get one byte of data synchronously from the serial port. See section 10 on page 33 for more information.

GetA This function is used to get one byte of data asynchronously from the serial port. See section 10 on page 33 for more information.

PutS This function is used to send one byte of data synchronously to the serial port. See section 10 on page 33 for more information.

ReadS This function is used to read data synchronously from the serial port. See section 10 on page 33 for more information.

FlushTx This function waits until the transmit FIFO in both RAM and the serial circuit has been emptied. See section 10 on page 33 for more information.

FlushRx This function resets the FIFO in both RAM and the serial circuit. See section 10 on page 33 for more information.

PutStr Send a null-terminated string over the serial port. See section 10 on page 33 for more information.

SetSerSpeed This function is used to set the speed of the serial circuit. See section 10 on page 33 for more information.

GetSerInfo This function is used to get information about the serial status. See section 10 on page 33 for more information.

StrCmp This function compares two null-terminated strings. It works like the strcmp() function in C. This function can thus be used in sort functions.

StrCmpNC This function compares two null-terminated strings case insensitive and return -1 (true) if they are the same and 0 (false) if they are not.

StrLen This function returns the length in bytes of a null-terminated string. The null-character at the end is not included. It works like strlen() in C.

StrCopy This function copies one null-terminated string into a buffer and put a null-character at the end. It works like strcpy() in C.

Str2Int This function takes a pointer to a buffer containing a number that is to be converted into a long value. Preceding white spaces are ignored. Numbers can be either decimal, hexadecimal if preceded by a $-sign or in binary if preceded by a %-sign.

Int2Dec This function takes a long value and converts it into a string of up to ten decimal numbers. If a buffer is passed in, the characters are written to this buffer with a trailing null-character. The buffer must be able to store up to 12 bytes unless the caller is sure less digits will be written. No preceding zeroes are written. If the buffer is a null-pointer the characters are sent directly to the serial port to ease debugging of software.

Int2Hex This function takes a long value and converts it into a string of eight hexadecimal numbers. If a buffer is passed in, the 8 hex-characters are written to this buffer without a trailing null-character. The buffer must be able to store 8 bytes. If the value does not fill out 8 characters, preceding zeroes are padded. If the buffer is a null-pointer, the characters are sent directly to the serial port to ease debugging of software.

CompFreeList This function is used to reduce the number of adjacent free blocks in the memory list. Compressing the free list speeds up further memory allocations and deallocations. See section 8 on page 27 for more information.

MemInfo This function is used to retrieve information about the state of the memory manager. See section 8 on page 27 for more information.

MemCheck This function checks memory for faults. It makes a read-write-read-write access to every byte in a specified memory area. See section 8 on page 27 for more information.

StoreProg This function is used to decode and store a downloaded program from the S19 format into binary in memory. See section 12 on page 37 for more information.

GetEnvToA6 This function is used to obtain a pointer to the global environment variables, envvars, ExOS uses to store various information. This function checks to see if the memory has been swapped and always returns the correct address to the environment variables. See appendix D for more information.

IncLed This function is useful when debugging. Every call to IncLed increases the value of the 8 LEDs on the main board and outputs the new value to the LEDs.

7 Kernel

This section describes the inner most core of the operating system which is how to handle processes. Memory management and devices, such as the timer device, can be considered core material but these things are quite comprehensive and covered in their own sections.

7.1 Introduction

To be able to make ExOS useful, making a console with a number of serial functions and a memory manager was not enough. dBOX has relatively much memory for one single program to use. This lead to the decision to introduce the concept of multitasking into ExOS. The type of multitasking is a non-prioritized, pre-emptive multitasking using the Round-Robin algorithm.

7.2 Stacks

As stacks play an important role in the design of a multitasking kernel this subsection tries to clarify which stacks that exists and what they are used for. But first a short introduction to stacks in general.

Whenever a subroutine is called, the return address to the calling code is put on the current stack. If the subroutine intend to use any registers, which is usual, it must first store also these registers on the stack before using them so that they can be retrieved and restored when the subroutine has finished. In ExOS all registers (except the return register of course) must be preserved in all functions or subroutines. When a subroutine is entered that causes the CPU to enter the supervisor mode, whether it already was in supervisor mode or not, also the status register is put on the stack along with the return address. TRAP calls, exceptions and interrupts causes the CPU to enter supervisor mode. The return from these routines uses a slightly different instruction, which also reads the stored status register from the stack and puts it in the CPU, causing it to go back to the previous mode before the call. Often back to user mode.

The CPU has two stack pointers; one user stack pointer and one supervisor stack pointer. These two stacks are used when the CPU is in the two modes, user mode or supervisor mode respectively. All user programs execute using a user stack which is local to each process. It is local because ExOS scheduler swaps also the stacks when a rescheduling is made. This has to be done to keep different processes from trashing each others stacks which may contain local variables and such. Interrupts, exceptions and TRAPs execute in supervisor mode and thus use the supervisor stack. ExOS maintains only one supervisor stack. This way it is easy to swap between user programs – processes – but makes rescheduling impossible if a user program would execute in supervisor mode, like for example while allocating memory. Maintaining multiple supervisor stacks would be very hard as interrupts can occur at any time, can be nested and are not executing as part of any process but asynchronously to everything else.

7.3 Multitasking

The idea of how to switch between processes is fairly simple. A low frequency timer causes an interrupt, which stops the currently running user program and stores the processor status on the stack. The CPU enters supervisor mode, which allows manipulation of all of the status registers of the CPU. A list of processes is searched for the occurrence of another process that has been interrupted before but still wants to run. If such a process is found, the scheduler stores all CPU registers and the processor status for the current user program. The processor status for the new process is placed on the stack and the registers are loaded. The interrupt routine returns and the new process continues its execution. If no other process wants to run, the scheduler does nothing but let the current program continue for another time quantum.

There are however certain problems that can occur. If the CPU is in supervisor mode already, for example because a TRAP call was made by the user program or a lower priority interrupt has occurred but not yet finished when a scheduling interrupt occurs, it is not possible to switch processes as both the currently running process and the interrupt routine are using the same stack, the supervisor stack. The question that arises is how to handle this situation. One solution would be to simply not reschedule processes but instead wait for the next schedule interrupt to occur and hope that the current program is not in supervisor mode at this time. If the current program however makes heavy use of TRAP calls or a lower priority interrupt is used a lot, chances are that many scheduling interrupts may occur before it interrupts the user program in user mode and can reschedule. The end user would experience the computer as less responsive or slow in such a case. The other way to solve this problem, and also the way used in ExOS, is to manipulate the bottom of the supervisor stack to intercept the last return to user mode so that the rescheduler is called instead when all supervisor calls are finished. This is called delayed rescheduling.

7.4 Processes

A linked list of all processes is managed by the program manager which is the terminal. Every process has a process control block, PCB, attached to it containing information and state for each process. Infact, not only processes have a PCB but everything loaded into RAM using the terminal gets a PCB. This means that every program, script or data block has a PCB. There are three types of PCBs: *Code*, *data* and *script*. PCBs of type code are used for programs which can be executed and are then turned into processes. Envvars maintains a pointer to the list of PCBs. Processes can have seven different states: *New*, *ready*, *running*, *waiting*, *suspended*, *finished* or *crashed*. The ready, running and waiting states are the active states. The scheduler does not care about processes in any other state. It is very important that only PCBs of type code are in the state ready, running or waiting. Following is a short description of the seven states:

New A process in state new has not yet been executed or has been restored to its original state after execution. The terminal can restore a crashed, finished or suspended process into the new state again.

Ready A process in the ready state wants to run but has been scheduled not to run at this particular moment.

Running This process is currently running on the CPU. There can only be one running process at a time unless the computer has more than one CPU. dBOX has only one CPU and ExOS can not handle more than one CPU.

Waiting This process has nothing to do for the moment and has called the function Block to wait for signals and to let other processes use the CPU in the meantime. It will be set to running once it receives a signal from either the OS or another process.

Suspended This process has been stopped by the user. A suspended process can be set to ready to make it continue execution.

Finished This process has finished executing and has exited. A finished process can be run again after resetting its program counter and by setting it to ready.

Crashed This process has executed an illegal instruction or accessed an illegal address and caused an exception. It is similar to the suspended state, but not caused by user intervention. An error code is also attached to this state.

See appendix D.6 for detailed information about the data stored for each process.

7.5 Functions

To make the CPU utilization and power consumption optimal, ExOS has four key functions that processes can use when they either have nothing to do or want to notify another process about an event of some kind. These functions are **Block**, **Signal**, **Event** and **Connect**.

Block should be called by a process whenever it has finished its current task and awaits more data to work on from e.g. another process or the serial port. A timer can also be set to signal a process when it times out. Block is then used to wait for this signal. Block takes a 32-bit mask of signal bits as a parameter that Block is instructed to listen for. If any of these signals have already arrived to the process before the call to Block was made, Block immediately returns with these signals as a return value. If none of the bits Block was instructed to wait for has arrived, Block puts the calling process in the waiting state and schedule another process that is ready to run. If no other process are in the ready state, Block executes the STOP instruction which halts the CPU until an interrupt occurs. Stopping the CPU causes all bus activity to cease, which means that all memory will enter the low power standby mode. This will reduce the power consumption of the computer and minimize the electronic noise around it. Block may only be called by user programs in user mode. It may be called if the kernel has not yet been initiated. The STOP instruction is in this case called immediately. If Block is called without any bits to wait for it will not recognize any signals and will wait forever. Block is the only function able to detect and receive signals.

Signal can be used by both user programs and ExOS to wake up a specific process from the waiting state and put it in the running state. A mask of signals is then sent to the specified process. If this process is currently in a Block call waiting for any of these specific signals it is immediately scheduled to run. Otherwise, if the signalled process is not in a call to Block or in a call to Block but not waiting for any of these specific signals, the bits are set in the process control block of the process that is being signalled and the process calling Signal continues to run.

Event is used to signal to a, for the caller, unknown destination that an event of some kind has occurred. To use Event, a call to Connect has to be made first to install an event handler that Event can use. The timer device uses Event to signal to processes that a timer has timed out. What distinguishes Event from Signal mostly is that Event can not only send a signal but can also call a handler asynchronously from the process point of view. A handler is basically a function somewhere, usually belonging to the process that installed the handler using Connect. This method is usually more efficient than to switch processes, which happens when a signal is sent, if something less complex has to be done that does not need the other process "awareness" of the event. An example would be high speed data sampling.

Connect is used to connect events to devices, e.g. a timer that is registered by a user program. In the call to Connect the user must specify what type of device to connect and to which unit number. The unit number can be the same as another as long as there is a difference between the type of the two devices. When e.g. a timer times out the user program may be aware of the event in two ways, via a signal or via a call to a handler. When a call to Connect is made, the user program must specify which one of the two that is to be used. If a signal is to be used, a signal number and a process control block must be specified. The signal number is the number in the 32-bit mask of signal bits that was mentioned above. The process control block is used by ExOS to know what process to send the signal to. If a handler is to be used instead, an address to that handler must be specified and an address to a place in memory where user data is stored. This user data can be used by the user program for various things. The code executing in the handler must preserve all registers it uses and restore them before returning. A handler executes in supervisor mode and is thus priviledged but also limited in some ways. Memory functions for example are forbidden to be used in supervisor mode.

7.6 Exceptions

Whenever a user program executes an instruction that causes an exception of some kind, e.g. a division by zero, a word or long memory access at an odd address or an illegal instruction, its state is set to crashed and rescheduling occurs. This can however be prevented if a subroutine has been connected to the exception handler. See appendix D. In this case this routine, or function, is called instead which gives a process a chance to catch exceptions. This is for example necessary if tracing is to be used as ExOS would otherwise crash the process requesting tracing if it doesn't catch the trace exception itself.

8 Memory management

In this section the memory management in ExOS is discussed. Memory management includes allocation, deallocation and compression of memory. Further, memory check and information about free memory is discussed. The memory management section in ExOS is separated from the rest of the OS. There are no other functions involved in the management of memory.

8.1 How to store memory information

One question that arised when the planning of the memory management started was "How can the OS know which segments of memory are free and which are occupied?" First we thought about using two linked lists, one for free memory and one for occupied memory. Then a decision to use one single array instead was made. With an array all memory information could be kept in one place. The reason to use one singel array was that if there were two places to keep information, and one contained corrupted information it would be difficult to know which one that contained the corrupted information and which contained the correct information. To keep the memory information in an array separated from the actual allocated memory segments, it is more unlikely that a faulty program would accidently overwrite it. This compared with having the information in a linked list where each node would be a part of the memory segment making it vulnerably to overflows in the allocated memory segment. Why corruption can occur in the first place is because the CPU has no memory management unit (MMU) and thus can not protect memory. This in effect means that the hardware allows all programs to read and write everywhere in memory, include overwriting themselves and other programs.

The array that ExOS uses is called the free list and contains blocks. Each block keeps information about one memory segment which has a start address, an end address and a size. Right after a hardware reset the free list contains one block that represents that the whole memory is free. For more specific information see appendix D.

8.2 Allocating memory

When a call is made to the AllocMem system function, the OS searches its free list for a segment where the requested size of memory will fit. When it finds one, it splits that segment into two segments. The first segment is set to be occupied and the second segment is set to be free. To set a segment to be occupied means to set that segment's size to zero. By this the operating system knows how to determine which segment that is free and which one that is occupied. Before doing the split the OS has to move the blocks that lies after the chosen one downwards. So after this the free list contains one more block. Memory is always allocated from the first segment in which the requested memory size fits.

8.3 Deallocating memory

When an allocated memory segment is to be returned by a call to the FreeMem system function, the OS searches its free list for the start address to that memory segment. When the address is found, the size field in that block is set to the difference between the end and start address of that memory segment. By this the block that was occupied before is now free for allocation. If an address was given to the deallocator that was not present in any allocation block, i.e. a faulty pointer, nothing is done.

8.4 Fragmentation of memory

When memory is first allocated and then deallocated the free list becomes fragmentated and may consist of many adjacent free segments in memory and also many blocks marked as free in the free list. AllocMem can not allocate a large segment in memory that is represented by more than one free block in the free list. This is why a fragmented free list is a problem. The reason why this is not done while deallocating is because of efficiency. The function CompFreeList, which defragments and compresses the free list, works in a way that makes it a lot more efficient when compressing the whole free list than FreeMem would do.

Whenever AllocMem failes to allocate memory, it compresses the free list to make all small, adjacent free segments into one large segment. AllocMem then attempts to allocate the requested amount of memory again. The compression is done from the end of the list to the beginning. Since the elements in the list gets fewer for every merge there is, the workload may decrease and the process speeds up as a result of the fewer blocks there is to move upwards in the free list array. If all memory is deallocated, the free list only contains one block after a compress is done.

8.5 Memory information

There is a possibility to aqcuire information about the memory usage in dBOX. The function MemInfo makes it possible to retrieve information about the amount of free and occupied memory segments in the RAM. It is also possible to retrieve information about the largest free segment size and total amount of free memory. The function MemInfo may be called with one parameter that tells it which information it should return. For more information see appendix E.

8.6 Memory check

The function MemCheck checks the memory for faults. It may be called with the start address to the memory that should be checked. If the start address is even, the function checks all even addresses and if it is odd then odd addresses is checked. The amount of checked bytes is also a parameter to the function. The return value indicates if the memory has faults or not, but not where in the memory the faults resides.

8.7 Problems with memory management

Problems may occur when an interrupt arrives and the memory manager is busy if the interrupt also decides to allocate or deallocate memory. If the interrupt happens to be the scheduler, problems also occurs if a process gets interrupted when it executes in the middle of the memory manager code, i.e. is allocating or deallocating memory. To solve this latter issue, AllocMem and FreeMem are TRAP calls which makes them execute in supervisor mode and the scheduler is forbidden to reschedule when the CPU was executing in supervisor mode. This solves the scheduling issue. However, allocating and deallocting memory in supervisor mode is otherwise forbidden.

The function CompFreeList is also sensitive but it is a syslist call and as such can be interrupted at any time. To prevent this, CompFreeList is also forbidden to call from supervisor mode and also must be surrounded by a SingleTask(TRUE) / SingleTask(FALSE) pair.

9 Timer device

ExOS offers the opportunity for a user program to have a timer. A timer can be used in many different ways. It can be used to measure elapsed time between two states or as a software interrupt. In this section the timer device that resides in ExOS is explained.

9.1 Description

ExOS' timer device has an amount of 128 timers and a resolution of 26ms. Each timer can be registered, unregistered, set to a start value, reset to the start value, started, stopped, read and connected to cause an event. One TRAP call is used for all functions of the timer device and one of the in-parameters specifies which timer function that should be called.

9.2 Register a timer

When a user program needs a timer it has to register a timer. The probability is very small that there is no one free, but if there is no one free they will have to wait until there is one timer free. When a timer is registered it can not be used by someone else. When the registering of a timer is done the user gets a reference number that must always be used to refer to the timer.

9.3 Unregister a timer

When a user program does not need a timer anymore, it can be unregistered and put back into the list of free timers.

9.4 Set a timer

The timer can be set to a start value. The direction, up or down, of the timer can also be set as well as if the timer should be repetitive or a "one-shot" timer. The start value is set in milliseconds. When setting the timer the user program should set the value in multiples of the resolution for the timer to be as correct as possible as the timer does not actually count in milliseconds. If the user does not do this, the stored value gets truncated. The timer will still work, but the result may not be as expected from the user's point of view. If a value less than the resolution is set, e.g. 20ms when the resolution is 26ms, the timer will wait for about half an hour! ($26ms * 65536$) This is because it always rounds the value down, which in this case makes it zero. The first timertick makes the internal counter, which is a word (16 bits), overflow creating the value 65535. Now another 65535 ticks, each 26ms long, will have to pass before the timer times out.

9.5 Reset a timer

The timer can be reset to its start value at any time. This can be useful if for example a user program has a task to do that must be done within a given time but can be finished before the limit is reached. If the task is finished before the time limit exceeds, the program can reset the timer and start over with the task. A good example of use for a reset function is for a screen saver that is reset everytime the mouse moves or the keyboard receives input.

9.6 Start and stop a timer

Of course the user program can start the timer and also stop the timer. When the user program sets a timer it is not started. Therefore the user program must start the timer after setting the values of it.

9.7 Read a timer

The user can get the current value of a timer from the timer device. This is good if the timer is used as a stop watch. The timer can count both up and down. When reading the timer it is important to know that the timer is not able to return every millisecond but only multiples of the resolution.

9.8 Connect a timer

When a timer has been registered, it can be connected to a handler or a signal. A handler is a pointer to a function in a user program that will be called when the timer times out. For more information about signals and handlers see section 7 on page 21.

10 Serial communication

The serial communication functions are completely isolated from the rest of the OS components. No other part of the OS touches the serial port or structures except these routines described in this section. Of course this is also true for user programs.

The UART circuit used is a 16C550. It is connected to IRQ 4 through the expansion card it is mounted on. It supports one full duplex serial channel with one FIFO buffer of 16 bytes each for receive and transmit respectively. It can be programmed to speeds up to 1.5Mbps. It is connected to use the RS232 standard. The serial routines use the settings 8N1 without any handshaking, although the hardware is wired with all signals and thus support all kinds of handshaking available for RS232 and modems.

10.1 How it works

All serial communication is interrupt driven. This means that the rest of the OS and user programs can take full advantage of the CPU at the same time as the serial port is fully active.

Apart from the UART's own 16 byte FIFO buffers there are also two 1kB FIFO buffers in memory, one for receive and one for transmit. These are used for asynchronous reads and writes by the other parts of the OS and user programs and being able to accept input data even when no process is currently listening. This means that ExOS always uses buffered I/O.

Each FIFO has three fields: a *base pointer*, a *beginning* and an *end*. The base pointer points to the actual memfory location where the FIFO buffer is. The beginning and the end are counters. Both are initiated to zero in the function InitSerial that is executed at startup time. They will then count transmitted or received bytes indefinitely. When they are equal there is no data in the FIFO. If end is higher than beginning it means that data has been put in the FIFO that has not yet been read or transmitted. The beginning and the end are thus not counting within the 1kB of the FIFO. The reason for this is that it would be hard to determine the amount of data in the FIFO if the end would have wrapped around the buffer and is then lower than the beginning which might not has wrapped around yet. This way we also get a counter of the number of received and transmitted bytes for free. The pointers are stored in envvars.

10.2 User functions

The serial routines supply 12 functions for sending and receiving data over the serial port and to control it. Seven of them are used for the actual communication. The other five are support functions. All functions for receiving and transmitting are safe to call in the absense of the serial circuit. See appendix E for a complete description and usage of these functions.

InitSerial This function initiates the serial circuit. If the system detects that the serial circuit has already been initiated once, after a restart of the computer for example, it will not be reset, only set back to its default values, except for the speed which is only set during the first initiation. During the first initiation the 16C550 is reset externally and speed is set to 9600 bits per second. In both the first and all subsequent calls, this function sets the mode to 8 bits, no parity, 1 start bit. It also initiates the FIFO pointers in RAM and the 16C550.

SetSerSpeed This function is used to set the speed for the communication.

GetSerInfo This function takes a pointer to a taglist as an argument. This taglist is interpreted and filled with values for every request found. Information can be requested about current speed, status, size of FIFOs, location of FIFOs, the current fill level of the FIFOs and how much data that has been transferred to the FIFOs.

GetS This function is used to get one byte of data from the serial port. If there is no data present, it blocks the caller.

GetA This function is used to get one byte of data from the serial port. If there is no data present, it returns immediately with an error.

PutS This function is used to send one byte of data to the serial port. If the transmit FIFO is full, this function blocks the caller until it can put the byte into the FIFO.

ReadS This function is used to read a specified number of bytes from the serial port. If the receive FIFO does not contain enough data this function reads all that is available and then blocks and waits for more data until it can satisfy the caller.

SendS This function is used to send a specified number of bytes to the serial port. If the transmit FIFO becomes full, this function blocks the caller until it can put all data into the FIFO.

SendA This function is used to send a specified number of bytes to the serial port. If the transmit FIFO becomes full, this function returns immediately with the number of bytes it actually managed to send.

PutStr This is a stub function for SendS to make it easier to send a null-terminated string to the serial port without the need to know its length.

FlushTx This function blocks until the transmit FIFO has been emptied in both RAM and the UART. Should be used before the speed is changed to be sure that all previously written data has been transferred using the old speed before it is changed.

FlushRx This function resets the receive FIFO buffers in both RAM and the UART.

11 Console

To be able to communicate with the surrounding world it is not enough to only provide serial functions. Something that can interpret what is sent over the serial port and respond with actions is also needed.

This section describes how ExOS interacts over the serial port. There are two consoles which can not be used at the same time. You may want to call the consoles *shell*s instead. One of them is very primitive and accepts only a few single-character commands. The other is more advanced and accepts multi-character commands possibly with parameters. The decision to make two consoles was made because it would be too much work to make a full featured terminal and put it in ROM without the possibility to be able to download programs into the microcomputer at all until it was all finished. Therefore a simple console with only the most basic features, like starting the memory manager and loading one single program into RAM and run it, was made first. The terminal could then be downloaded and tested from this primitive console. When the computer starts up it initially starts the primitive console.

11.1 The primitive console

This console can, and will, execute if the memory manager has not been initiated for whatever reason and if the serial port is present. It can take a few simple commands consisting of only one character. It can do things like change the serial speed and load a program into RAM the simple way (see section 12) and run it. It can also start the memory manager and the kernel if they, for e.g. debugging reasons, was not previously started. By typing "?" and then Enter a list of commands is printed.

11.2 The advanced console

The advanced console, also called the *terminal*, can take a lot more and complex commands than the primitive one. It needs the memory manager to be active. If the kernel has not been previously initiated it will initiate it. The terminal also serves as a program manager with functionality to store and manage a list of programs in memory. By typing "help" and then Enter at the prompt in the terminal a list of commands is listed. Following is a list of operations the terminal features.

- Load programs
- Start programs
- Stop programs
- List programs
- Delete programs
- Display memory manager information
- Read and modify memory areas

The terminal is the only part that was not written in assembly, but in the high-level language E on the Amiga.

12 Converting motorola hex format

This section describes how programs that the user downloads into dBOX are converted from hexadecimal format into raw binary data that can be executed by the CPU. The hexadecimal format used is Motorola's own hexadecimal format for transferring binary information across platforms. This hex format is later be referenced to as the ".s19 format" as ".s19" is the extension that files of this format usually have. This section will also describe how storage of these programs is done by ExOS.

12.1 Downloading and storing a program

The user can write programs and download them into dBOX. This can be done in two different ways, by the primitive console or by the terminal. When the primitive console is used, the memory manager may not yet have been activated. Because of this the console only takes one line at a time and converts it from the S19 format to binary. Since the memory manager may not have been activated, there can be no allocation of memory for the program. The console therefore puts the program just after the memory block in RAM that may be used later by the memory manager if initiated with default values. Only S19 records of type S1 or S2 are converted; other types will be rejected.

The other way to download programs is through the terminal. When a program is downloaded using the terminal the lines of that program are put in a linked list, where each line is an element in the list. The terminal does this job and then calls the function StoreProg to do the actual decoding and storing. StoreProg checks the size of the downloaded program. If the size is equal to zero an error code is returned and the storage is aborted, otherwise memory is allocated for the program. If the program does not fit into memory, an error code is returned and the storage is aborted. If no error occured, the lines of the program are converted from the S19 format into binary. If a bad checksum is discovered after decoding a line the storage is aborted, memory is deallocated and an error code is returned.

If all went well, the program size and the address where the program was stored in RAM are returned. The two values are then stored in a PCB by the terminal so that ExOS can control the execution of the program. See appendix E for a complete description of how to use the function StoreProg.

13 Problems

We have encountered a couple of problems during this project. They are described here in three sub sections.

13.1 Hardware problems

We have not had any serious problems with the hardware during the project but there have been a few minor errors.

- A circuit, the DIP-switch input driver, was placed in the wrong direction and made an output. This was quite easily fixed by simply turning it around and reroute a few signals. It was discovered before any initial tests had been made.
- A circuit was assumed to be 0.3" wide but was actually 0.6" wide which made it unfittable. Quite easily fixed.
- A capacitor was placed in the way of a resistor that had to be placed around it. Very small error.
- The main board had only one logical error. An enable signal to the main address decoding circuit was inverted and thus permanently disabled the address decoder which froze the whole computer.
- The original power supply unit was dropped on someones head (no names) and broke while he was crawling under the desk connecting a serial cable. As it was undetected that the power supply unit was defect, the serial circuit got 11V instead of 5V and broke too. A new serial circuit had to be ordered and a really old and big AT PSU had to be used.
- The low frequency clock on the main board could not be clocked faster than about 5MHz which left us with a timer that could not operate faster than 38Hz. A problem we had to live with.

13.2 Software problems

There have been a small variety of different software problems. Some bigger, some smaller, some hard to find.

- The first really hard to find software problem was that the program used to burn the 16-bit EPROM expected Intel byte-order on the bus and swapped the bytes which resulted in garbage for our Motorola CPU to execute. It took half a week and some extra debugging hardware before we found out what the error was. When it was found, however, it was easily fixed.

- As assembly is a time demanding language to program in we tried to use other high-level languages. C was one of the candidates but the one compiler we had access to that could generate 68k-code in S19-format could not generate relocateable, or position independant, code which is needed in our computer as it is impossible to predict where a program might be placed in RAM. The other language, Amiga-E, exists only for the AmigaOS and generates 68k-code but with startup code and a loadhunk to be loadable by the AmigaOS. A disassembler and a small program to interpret the disassembly output and turn it into S19 solved that problem.

- Switching between processes during multitasking turned out to be a big problem. As the rescheduler can be called from many places, in supervisor mode or user mode; it is almost impossible to know where the final return address to the user program is on the supervisor stack. The return address needs to be saved before a switch to another process can be made. The solution to this problem is to simply avoid being in supervisor mode as much as possible. In interrupts or other places where supervisor mode is forced or needed, the switch between processes is simply prohibited or delayed.

13.3 Other problems

The delivery of some of the components took a lot longer than expected which delayed the completion of the hardware. During that time however a lot of software was written and this report was started on so no time was actually lost.

14 Testing and debugging

This section describes how the testing and debugging of the project was made. It also describes how problems were discovered and their solutions. It is assumed that the reader has read section 13 *Problems* before continuing.

14.1 Hardware

Most of the debugging and discovery of bugs was done by visual examination, before the power was applied for the first time. The IC that did not fit and the capacitor in the way of the resistor for example was obviously discovered during assembly of the main board.

On the other circuit boards there were etching errors that were discovered by testing these cards separately. They were quite easily fixed. As the main board did not have that many bugs when we started to test it with power applied the debugging of it did not take very much time or effort. However, for the most part of testing and debugging a simple multimeter and a two channel digital sample-and-hold oscilloscope was used. With these instruments we could measure signals in the address decoding logic and see that the CPU was addressing the expected areas of RAM and ROM. This is how the inverted enable signal to the main address decoder was discovered.

When the enable signal was fixed the computer actually worked, but it executed none of the code in ROM as expected. At one point even the CPU was suspected to have failed so it got replaced. After two days of no progress in finding the cause of the malfunction, two pieces of debugging hardware were constructed. One card with a lot of LEDs that was connected directly to the CPU bus through an expansion slot. On this card the address bus and half the data bus could be read in binary. It turned out to be quite impossible to read anything useful as the speed of the CPU is so high. At this time another piece of hardware was made that could inhibit the bus cycles of the CPU by gating off the DTACK signal. With a switch connected to this circuit each bus cycle could be manually stepped one by one while the bus was read. This was how the byte swap of the ROM was discovered. It took four days to discover this error which actually was a software error.

14.2 The DIP-switches and LEDs

To be able to make simple communication possible with dBOX, the main board was fitted with an output port connected to 8 LEDs and an input port connected to an 8-bit DIP-switch. The DIP-switch together with IRQ 1 and IRQ 7 makes a simple input device able to give dBOX simple instructions such as setting the clock frequency of the CPU and output certain values of the environment variables to the LEDs.

14.3 Software

The software that was written has been tested in various ways. Some have been tested with a simulator which is included in the assembly package that was used during the programming of the software. Some have been tested on an Amiga running on an actual 68k CPU with wrapper functions in the high level language Amiga-E. E was chosen because it is essentially typeless which makes it easy to use, without nasty casts cluttering the code, when programming close to hardware. It also allows assembler instructions to be placed anywhere in the otherwise high level constructs. It also allows code outside of procedures, both assembly and E-code. Other pieces of code have been tested in real life directly on our micro computer.

Most of the code that is close to the hardware is meaningless to test on anything other than dBOX itself. Examples of this are the startup code and the serial code. Also time critical code with interrupts can only really be tested on the actual hardware.

To test the timer device together with the connect and event functionality a test program was written that is called the "Running Light". This program lets the user choose between a few "programs". Each program contains a byte stream that the program could run. During "execution" of the "program" each byte value is displayed on the LEDs on the motherboard. The user can select which program to run via the DIP-switches. There are three modes that each program can be executed in: *Forward*, *Backward* and *Bounce*. Forward and backward is what it sounds like and bounce is simply a modified forward and backward execution.

The code for Timer, Connect and Event were meant to reside in ROM but it was first tested by downloading it to RAM together with the running light code. This turned out to be more difficult than expected as the version of code in ROM was not up to date at that time. For example, the free list was not correctly initialized, so a new one had to be initialized via the program in RAM. Then there were something in the code that was not functioning properly. After a huge amount of time we discovered that when the microcomputer was reset, the pointer to the connect list was not reset. This caused the computer to do some execution with random results.

15 Conclusions

A computer was constructed based on a slightly modified version of the original design. An operating system was written based on our own thoughts and ideas. The hardware works flawless. The software works very good. The OS designed actually surpassed the goal of this project. It was originally not supposed to include a multitasking kernel. Also the timer device was not included in the initial goal planning but was decided at a later state to be included.

Time was always an issue during the project. The hardware could barely be finished within the timeframe of 3 weeks (6 calendar weeks) which was 30% of the time. dBOX did not turn out to be so very debuggable as we thought either, as extra hardware had to be built for this purpose. The planned software consisting of serial routines, a memory manager and a simple console were finished earlier than expected. Therefore a general timer device based on the LF-clock, a multitasking kernel and a more advanced terminal was also developed. Time did however not permit a full test of the kernel and the new terminal.

We have learned a lot while working on this project: What startup code actually does. Issues with general function calls when called in different CPU modes. How memory management can be implemented. And last but certainly not least; what makes a multitasking kernel and all the problems that comes with the design of it. It has been a very interesting and also fun project to work on.

15.1 Future

As this project could not meet all wishes covering the possibilities with the computer, this subsection is dedicated to future thought about the project, dBOX and ExOS.

The motherboard is not optimal in this first revision. The swapping of RAM and ROM was not a good idea. The next revision will have an additional 1kB or RAM that will be mapped of the first kB of ROM instead. This makes life a lot easier. The timer is much too slow to be really useful for anything but taskswitching and user delays. Two timers will be made for revision 2. One high frequency 16-bit timer reaching from a few kHz to a few hundred kHz. One low frequency 8-bit timer will be used that can run from a few hertz up to a few hundred hertz. The line thickness and distance between the lines on the circuit boards will be made a lot smaller. A wider cable for the expansion bus will be used so that no lines, like +/-12V and A20-A22, have to use their own wires on the side.

Other things that are planned for either revision 1 or 2 is an IDE harddisk controller, a PS/2 keyboard interface, a sound card and a graphics card. The harddisk controller is the easiest to build. Only three 8-bit buffer driver ICs are needed. The PS/2 interface is a little trickier as it uses its own type of serial communication. A sound card will most likely be based on the Amiga sound chip named Paula which supports four independent 8-bit, AM and FM modulatable, DMA driven D/A-converters, configurable as two 14-bit channels producing stereo sound from 4kHz up to 64kHz. This chip also has an RS232 serial interface built in and the ability to detect mouse movements from a mouse.

The hardest thing to make would be the graphics card as it requires very high data rates. If the decision to make a graphics card from scratch is made, a 16-bit 640*480 pixel VGA compatible card will be made. The problem with this is that it requires 617kB of memory and a constant data rate of 16MB/s only to keep the display on the VGA monitor. This is more than the CPU can handle so it has to be made using DMA. Another option would be to use an old graphics card designed for the ISA-bus in PCs. The problem could however be to aqcuire documentation for such a card. The future will tell what solution will be made. Perhaps just a simple LCD will be used. As static RAM, which is used on dBOX, is quite expensive the possibility to use old 72-pin SIMM moudles on dBOX will be investigated too. A bit more exotic thought about a PowerPC based computer using the CHRP standard has also been made which would probably run at about 100MHz.

About ExOS: As multitasking was not expected to be done in this project it would have been a thing for the future. What is left now are improvements like prioritized pre-emptive multitasking and garbage collection. Drivers for the new proposed hardware will of course also have to be written.

A Abbreviations

8N1 *8 data bits, No parity, 1 stop bit* A common configuration used for RS232 serial ports.

AM *Amplitude Modulate* A signal modifies, or modulate, the amplitude of another much higher frequency signal.

ASCII *American Standard Code for Information Interchange.* A standardized form of letting bytes represent the different characters on a western keyboard and a few additional control characters.

bps *bits per second.* Data transfer speed unit used mainly for serial transfers.

CHRP *Common Hardware Reference Platform* A standard for manufacturing main boards with PowerPC processors developed by IBM and Motorola.

CPU *Central Processing Unit.* Also known as a micro processor or simply a processor. This is the unit that executes the program instructions and answers to the interrupts given by other units.

D/A *Digital to Analog* Usually a device dedicated to convert digital value to analog voltage levels. Often used to convert digital sound to analog sound that humans can hear.

DIP *Dual Inline Package.* A method of packaging circuits and other electronical componens with many pins. In this document it is used to describe a set of switches grouped together in a 0.3" DIP.

DMA *Direct Memory Access.* A method circuits other than the CPU can deploy to read or write directly to memory by them selves while the CPU is doing other things, to save time. No DMA is occuring on our microcomputer (yet).

EPROM *Erasable Programmable Read Only Memory.* A kind of non-volatile memory that can only be written to using a special device. Ultra violet light is used to erase it before it can be re-written. This kind of memory stores most part of the operating system and startup code in our project.

FIFO *First In First Out.* A queuing mechanism used as an in between buffer in places where data can not be processes immediately but must be stored and available at the same time as new data can fill in asynchronously. In this project FIFO buffers are used in the actual serial circuit and in RAM to serve as buffered I/O.

FM *Frequency Modulate* A signal modifies, or modulate, the frequency of another much higher frequency signal.

I/O *Input / Output.* 1. I/O port: A name for a specific port in to or out from the computer. 2. I/O in general: A stream of data in to or out from a computer, typically through some kind of I/O port.

IC *Integrated Circuit.* A way of putting a lot of electronical components on a single chip and integrate them into a plastic or cheramic package. In this document interchangably used with the word "circuit".

IDE *Integrated Drive Electronics* A standard for connecting harddisks to computers.

IRQ *Interrupt ReQuest.* A signal that external hardware can issue to request the CPU to take an interrupt and serve the hardware's urges. The abbreviation "IRQ" is often interchangably used with the word "interrupt".

k *kilo.* A size operator generally meaning 1000. In computer relations though it often means 1024 when dealing with bytes and sizes but 1000 when dealing with bits and speeds.

kB *kiloBytes.* 1024 bytes. Not to be confused with "kb" which means "kilobit" or 1000 bits.

LED *Light Emitting Diode.* A semiconductor that emits light with a specific colour when subject to an electrical current of a few milliamperes.

LF *Low Frequency.* Abbreviation meaning low frequency. Used in this document together with the timer clock on the main board which is running with a very low frequency. The definition of LF varies depending on the context but generally reaches from 1 hertz to a few tenth of kilohertz.

MB *MegaByte.* 1048576 bytes or 1024kB. Used to measure amount of data or storage capacity.

Mbps *Megabit per second.* 1000000 bits per second. Data transfer speed. Is sometimes also written as Mbit.

ms *milli second(s).* 1/1000's of a second.

OS *Operating System.* A collection of code and functions needed to make a computer useful for user programs. It serves as an abstraction from the hardware. In this project it includes startup code, serial functions, memory handling, an abstraction to the LF-clock, and a user interface.

PCB *Process Control Block.* A structure containing information about a process.

PS/2 *Personal System/2* An invention by IBM to connect mostly keyboards and mice to computers.

PSU *Power Supply Unit.* A device used to convert the electricity from the net to voltages used by the computer and supply power to the computer.

RAM *Random Access Memory.* Volatile memory that can be both read and written to and addressed in any order.

ROM *Read Only Memory.* A memory that can only be read from. Usually these kinds of memories are programmed at the factory. See also EPROM.

RS232 *Recommended Standard 232* A computer serial interface standardised by IEEE. Usually written RS-232.

SCSI *Small Computer System Interface.* An interface often used to connect harddisks, scanners, tape streamers etc. to computers.

SIMM *Single Inline Memory Module* A memory module with the same connectors on both sides of the module, hence single. They exists as 8-bit 30-pin, or 32-bit 72-pin modules. They were used as main memory in computer mainly during the 1990's.

SR *Set Reset.* A kind of electronical flip-flop often used to eliminate contact bounce in physical switches. In this project SR-flip-flops are used to latch short interrupt pulses until the CPU has time to respond.

SRAM *Static Random Access Memory.* A kind of RAM that does not require refresh to keep its memory. Its information is stored in static flip-flops. See also RAM.

UART *Universal Asynchronous Receiver and Transmitter.* A circuit used to receive and transmit data to an other system asynchronously with the rest of the computer. A serial port is a good example of a UART.

VGA *Video Graphics Array* A standard to connect graphics cards to computers.

B Schematics

The design of dBOX was not part of this project but was already made several years ago. As this project had to include a minimum of hardware, a few redesigns had to be made to simplify the main board. It was also clear that everything would still not fit on the main board but needed to be moved out on small plug-in cards. Interrupt-, swap- and CPU-clock logic got placed on their own cards. During the layout and planning of the software a few other parts were found to originally not have been made in an optimal way and were changed.

For the interested reader it might be worth mentioning what program that was used to do this schematic. Personal Paint is the name of the program that was used. It is an 8-bit pixel based drawing program for the Amiga. If an old Amiga user reads this (s)he might remember a program called Deluxe Paint which is very similar to Personal Paint, but older and does not work on newer Amigas. This program, despite that fact that it is nothing more than a drawing program, is surprizingly well suited to use for electronic designs and layouts with its grid system, nine clipboards and double buffers which makes it possible to layout double sided circuit boards. It was used for the main schematics design and the layout of the double sided main board and combined expansion and serial card. For the interrupt, swap and CPU-clock cards Layo1 was used. This program is a real electronic circuit board design program for PCs with Windows. The demo version can design cards with up to a hundred soldering pads.

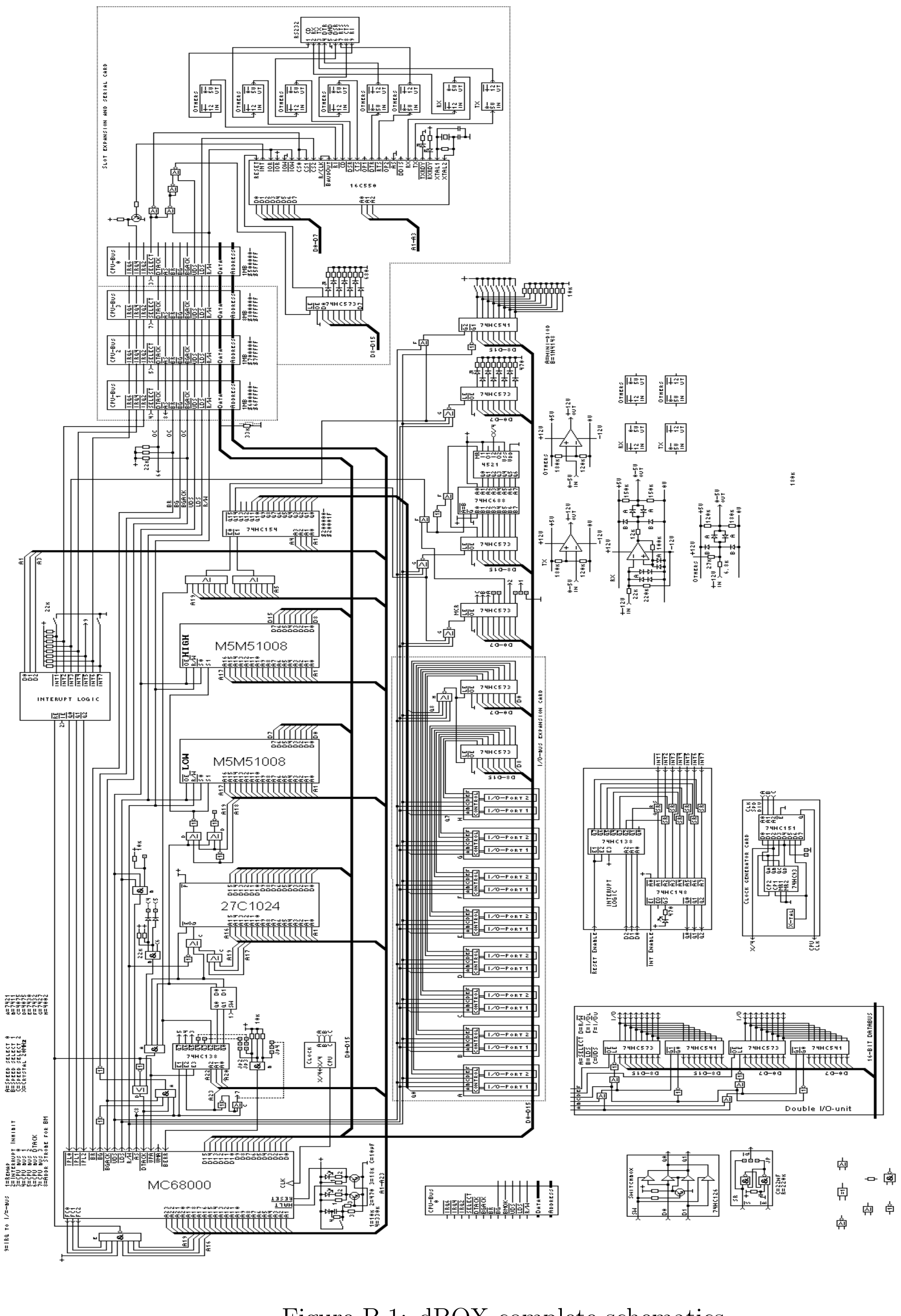

Figure B.1: dBOX complete schematics

C Pinouts

There are two expansion connectors on dBOX. One is the I/O-port expansion bus, or simply the I/O-bus. This bus does not have any signals necessary for bus mastering, nor does it have an address bus. It is mainly intended for simple I/O-ports.

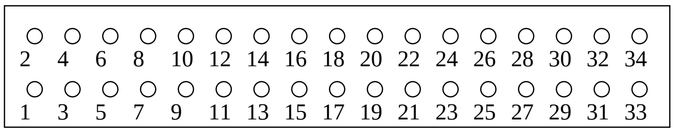

Figure C.1: I/O bus expansion connector

Description	Abbr.	Pin	Pin	Abbr.	Description
Supply voltage 5V	Vcc	1	2	GND	System ground 0V
Upper data strobe	$\overline{UDS}$	3	4	$\overline{LDS}$	Lower data strobe
Data bus bit 0	D0	5	6	D1	Data bus bit 1
Data bus bit 2	D2	7	8	D3	Data bus bit 3
Data bus bit 4	D4	9	10	D5	Data bus bit 5
Data bus bit 6	D6	11	12	D7	Data bus bit 7
Data bus bit 8	D8	13	14	D9	Data bus bit 9
Data bus bit 10	D10	15	16	D11	Data bus bit 11
Data bus bit 12	D12	17	18	D13	Data bus bit 13
Data bus bit 14	D14	19	20	D15	Data bus bit 15
Interrupt request line 5	$\overline{IRQ5}$	21	22	R/$\overline{W}$	Read/Write
I/O-port select 0	$\overline{Q0}$	23	24	$\overline{Q1}$	I/O-port select 1
I/O-port select 2	$\overline{Q2}$	25	26	$\overline{Q3}$	I/O-port select 3
I/O-port select 4	$\overline{Q4}$	27	28	$\overline{Q5}$	I/O-port select 5
I/O-port select 6	$\overline{Q6}$	29	30	$\overline{Q7}$	I/O-port select 7
I/O-port select 8	$\overline{Q8}$	31	32	$\overline{Q9}$	I/O-port select 9
I/O-port select 10	$\overline{Q10}$	33	34	$\overline{Q11}$	I/O-port select 11

Table C.1: I/O bus expansion connector

The other connector is the CPU bus expansion, also known as the CPU-slot. It is depicted in figure C.2. It contains all signals necessary for a device connected to it to master the bus. Bus mastering is necessary if a device needs to perform DMA data transfers. This also means that another, possibly newer and faster, CPU can be connected here to take over from the old CPU.

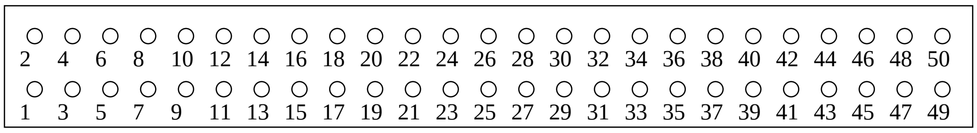

Figure C.2: CPU bus expansion connector

Description	Abbr.	Pin	Pin	Abbr.	Description
Supply voltage 5V	Vcc	1	2	GND	System ground 0V
Bus request	$\overline{BR}$	3	4	$\overline{IRQ6}$	Interrupt request line 6
Bus grand acknowledge	$\overline{BGACK}$	5	6	$\overline{IRQ4}$	Interrupt request line 4
Read/Write	R/$\overline{W}$	7	8	$\overline{IRQ2}$	Interrupt request line 2
Lower data strobe	$\overline{LDS}$	9	10	$\overline{BG}$	Bus grant
Upper data strobe	$\overline{UDS}$	11	12	$\overline{DTACK}$	Data acknowledge
Address strobe	$\overline{AS}$	13	14	$\overline{SELECT}$	CPU-bus select
Data bus bit 0	D0	15	16	D1	Data bus bit 1
Data bus bit 2	D2	17	18	D3	Data bus bit 3
Data bus bit 4	D4	19	20	D5	Data bus bit 5
Data bus bit 6	D6	21	22	D7	Data bus bit 7
Data bus bit 8	D8	23	24	D9	Data bus bit 9
Data bus bit 10	D10	25	26	D11	Data bus bit 11
Data bus bit 12	D12	27	28	D13	Data bus bit 13
Data bus bit 14	D14	29	30	D15	Data bus bit 15
System ground 0V	GND	31	32	A19	Address bus bit 19
Address bus bit 18	A18	33	34	A17	Address bus bit 17
Address bus bit 16	A16	35	36	A15	Address bus bit 15
Address bus bit 14	A14	37	38	A13	Address bus bit 13
Address bus bit 12	A12	39	40	A11	Address bus bit 11
Address bus bit 10	A10	41	42	A9	Address bus bit 9
Address bus bit 8	A8	43	44	A7	Address bus bit 7
Address bus bit 6	A6	45	46	A5	Address bus bit 5
Address bus bit 4	A4	47	48	A3	Address bus bit 3
Address bus bit 2	A2	49	50	A1	Address bus bit 1

Table C.2: CPU bus expansion connector

D Structures used by the OS

Various parts of ExOS uses different structures to be able to store information in comfortable ways. This appendix section describes all structures used. C-syntax is used to make it as understandable as possible for most readers. In the actual assembly sources this syntax is of course not used. See appendix F, or more precisly appendix subsection F.1 on page 100 for more specific information.

D.1 Environment variables

The core structure that connects everything in ExOS is the structure containing all environment variables – envvars. A short description of every member is made as a C-comment. A longer description of some of the fields are made below. All `void*`'s in envvars are actually function pointers that can be set to point to a subroutine to be executed in a particular place. If null, this subroutine call will be ignored.

```
struct EnvVars {
  char mcr;     /* Copy of the write-only Master Control Register on dBOX */
  char lfc;     /* Copy of the write-only Low-Frequency Clock on dBOX */
  char led;     /* Copy of the write-only LED register on dBOX main board */
  char int7;    /* See below */
  char status;  /* Global status register */
  char ramlock; /* Not used */
  char rxlock;  /* Not used */
  char txlock;  /* Not used */
  char ks;      /* Kernel status */
  char pad;     /* Unused spare/pad byte */
  short freez;  /* Free list size: Max # of memory allocations allowed */
  FreeListBase *freel; /* Memory manager user data */
  char  *seriptr; /* Serial Input Pointer: Serial private data */
  ulong seribeg;  /* Serial Input Beginning: Serial private data */
  ulong seriend;  /* Serial Input End: Serial private data */
  char  *seroptr; /* Serial Output Pointer: Serial private data */
  ulong serobeg;  /* Serial Output Beginning: Serial private data */
  ulong seroend;  /* Serial Output End: Serial private data */
  ulong id; /* Startup code uses this to see if envvars has been copied */
  void *srbint1; /* Subroutine Before Interrupt 1 */
  void *sraint1; /* Subroutine After Interrupt 1 */
  void *srbint2; /* -- || -- */
  void *sraint2; /* -- || -- */
  void *srbint3; /* -- || -- */
  void *sraint3; /* -- || -- */
  void *srbint4; /* -- || -- */
  void *sraint4; /* -- || -- */
  void *srbint5; /* -- || -- */
  void *sraint5; /* -- || -- */
  void *srbint6; /* -- || -- */
  void *sraint6; /* -- || -- */
  void *srbint7; /* -- || -- */
  void *sraint7; /* -- || -- */
  void *srarx;    /* Subroutine After serial Receive interrupt */
  void *sratx;    /* Subroutine After serial Transmit interrupt */
  void *sraline;  /* Subroutine After serial Line status interrupt */
  void *sramodem; /* Subroutine After serial Modem status interrupt */
  void *srexcept; /* Subroutine for Exceptions */
  TimerBase *timer;      /* Pointer to private data of the timer device */
  ConnectBase *connect; /* Pointer to private data of Connect */
  ProcessControlBlock *pcb; /* Pointer to the linked list of PCBs */
  Kernel *kernel;        /* Pointer to the private data of the kernel */
  void *crbucket;        /* Crash Bucket subroutine */
}
```

The fields `mcr`, `lfc` and `led` are copies of the registers with the same names on the main board. Whenever such a register it written to, then same value should be written to one of these variables so that it can be read. The field `int7` can contain a value for the interrupt routine for IRQ 1 to read. Usually this is set by interrupt 7 during hardware debugging. The field `status` can have any of the bits in table D.1 set.

The fields `ramlock`, `rxlock` and `txlock` can be used as semaphores to prevent more than one process from executing in the memory manager or serial routines at the same time. Other solutions to this problem have been developed and these variables are not

Bit	0	1
ES_SWAP	ROM at address 0	RAM at address 0
ES_SER	Serial is not present	Serial is present and has been initiated
ES_LFLED	IRQ 3 will not increase LED	IRQ 3 will increase LED
ES_KERNEL	Kernel has not been initiated	Kernel has been initiated

Table D.1: Envvars status bits

currently used.

The fields `srbintN/sraintN` can contain addresses to subroutines to execute either before the body of an interrupt routine executes or after it has executed. They provide a snap-in functionality so that additional code can be attached to an interrupt without interferring with the original interrupt routine. They are all set to null by default which disables this functionality. The fields `srarx/tx/line/modem` are similar to the fields `srbintN/sraintN`, but are more specifically targetted to the serial routines.

The value in field `crbucket` can be an address of a subroutine that will be called whenever an irreversable error such as an address- or bus error has occured while the CPU was in supervisor mode and thus could not swap to another process and return.

D.2 Timer

The timer information is stored in two structures. The TimerBase, which keeps global information about the timer device and an array of timer elements, and a structure that keeps information of a single timer.

```
struct TimerBase {
  short lastReg;
  TimerElement tel[128];
}

struct TimerElement {
  char  userData;
  char  status;
  short startValue;
  short counter;
}
```

`lastReg` is a word containing an offset to the last timer element that could be registered. If this offset is 0 than there are no registered timers in the array. There may be unregistered timers before this one. This offset is used to speed up the traversion of the array when updating elements. `userData` can be used to anything the user wants. `status` is an 8-bit mask where the bits are as shown in table D.2.

Bit	0	1
7	Not registered	Registered
6	Off	On
5	Count down	Count up
4	Repeat	No repeat
3-0	Not used	Not used

Table D.2: Timer status bits

`startValue` is the value that `counter` is loaded with at start and reset to if the timer is set to repeat.

D.3 Connect

The connect information is stored in a linked list with the pointer to the first element stored in envvars. The elements in the list have the structure:

```
struct ConnectBase {
  ConnectBase *next;
  char device;
  char type;
  short unitNumber;
  union uAddress {
    long  userData;  /* type=CON_HANDL */
    ulong signal;    /* type=CON_SIG */
  }
  union iAddress {
    long intHandler; /* type=CON_HANDL */
    ProcessControlBlock *process; /* type=CON_SIG */
  }
}
```

`next` is a pointer to the next element in the list. If it is equal to 0, the element is the last in the list. `device` is a number that indicates the type of device (e.g. DEV_TIMER). `unitNumber` is a number that identifies a unit. Together with `device` it makes an unique identification of an unit. `type` is the event type. It can be a handler or a signal. `uAddress` is an address to user data or a signal number. `iAddress` is an address to a subroutine or a PCB depending on `type`.

D.4 List of free memory

Memory manager structures.

```
struct FreeList {
  short counter; /* the current number of entries in the MemBlock list */
  MemBlock memBlk[1000]; /* the actual list of segments in memory */
}

struct MemBlock {
  ulong startAddress; /* the address where a block starts */
  ulong endAddress;   /* the address where a block ends */
  ulong freeSize;     /* size of a block */
}
```

`freeSize` represents the size of a free block in the list, which is the same as the size of the free segment in the memory. If this is 0 then the block is occupied, otherwise it is free.

memBlk has a default size of 1000 elements but can be change in run-time by a suitable function if desired. No such function has been design when this report was written.

D.5 Kernel

The basic structure used by the kernel functions.

```
struct Kernel {
  ProcessControlBlock* curpcb; /* currently running process */
  long* sspref;  /* the top of the supervisor stack */
  long  toppc;   /* stores the original PC when the intCatcher is used */
  short topsr;   /*    --- || ---       SR      ---- || ----
  char  qcnt;    /* quantum counter */
  char  quantum; /* quantum used=number of timer ticks between rescheduling */
}
```

sspref represents the top of the supervisor stack when any user program is executing. This might not necessarily be the absolute top of the stack as ExOS might have put something there before initiating the kernel. This supervisor stack reference is used so that the intCatcher routine can be put right at the final return address to user mode for delayed rescheduling during interrupt- or exception handling.

qcnt is a counter that tells how many timer ticks there are left before rescheduling should be done. quantum defines how many timer ticks there should be between rescheduling should occur.

D.6 Process control block

The structure containing all information about a program, or process.

```
struct ProcessControlBlock {
  ProcessControlBlock* next; /* this is a linked list.. */
  char* data;     /* a pointer to the actual code or data */
  long  size;      /* size of the code or data in bytes */
  char  type;      /* What this is: PS_* */
  char  status;    /* status, or state, of the process */
  short sr;        /* status register when swapped out of execution */
  long  pc;        /* program counter when swapped out of execution */
  long  regs[15]; /* registers except A7 when swapped out of execution */
  long* sp;        /* stack pointer when swapped out of execution */
  long* stack;     /* pointer to the allocated memory for the stack */
  long  stsize;    /* stack size in bytes */
  ulong sigmask;   /* signals allocated for this process by AllocSignal */
  ulong sigwait;   /* the parameter sent to Block if Block would block */
  ulong sigset;    /* signals that have arrived but not yet been received */
  char* name;      /* the name of this block */
}
```

`data` is a pointer to the actual code or data this PCB represents. A PCB can also store a data block. The `type` field tells if this PCB contains code or data. *Perhaps it was not such a bright idea to have the same structure designed to keep both processes and data and also scripts. From the terminal's point of view (which maintains all PCBs) a PCB is essentially just a file with a type flag. In the beginning there could not be any other files than program files which then had their different states. See section 7. Later other types were added but the PCB remained as the only list, now representing both files, programs and processes.*

`sigwait` is the parameter sent to Block if Block is putting the process to sleep. Signal verifies its parameter agains this field to see if any signals matches which means that this process should be woken up. This field is always zero when the process is running.

`sigset` contains signals that have arrived to the process but not yet been received by a call to Block.

D.7 Taglist

This is a small structure used by a few functions to send and receive various amounts of data. An array of TagItem forms a taglist.

```
struct TagItem {
  ulong code; /* command, or code describing what to do with the data */
  ulong data; /* data field. May be anything;in/in-out/out value,ptr,ref */
}
```

data is a general field which essentially can have any type as long as it fits in an ulong. It may be a value, it may be a pointer or reference, it may be empty and thus meant to be filled with data by the function. The code field implicitly defines the type of the data field. The codes are defined individually for each function using a taglist. A code value of for example 20 may mean two completely different things for two different functions. If the code field i 0 it indicates the end of the taglist. A code can thus never have the value of 0 as this is reserved as an end-of-list code. (This is another idea borrowed from AmigaOS which makes heavy uses of taglists in many OS calls.)

E System calls

This appendix section describes how to use all system calls in detail. The functions are sorted in alphabetic order. They generally follow the same scheme with a number of different fields:

NAME The name of the function and a short description of what it does.

SYNOPSIS A semi-assembly-like and a C-like description of how to call the function. In the semi-assembly-like description registers which are used as parameters or return values are marked in a special way: Data registers can be marked in one of two ways; either Dn.z or Dn.z.e where "n" is the number of the register, "z" is the normal word size and "e" is the word size if an error is to be signalled by a return value. The word size can be ".B" for byte (8 bits), ".W" for word (16 bits) or ".L" for long (32 bits).

Address registers can be marked in one of two ways; An or An.z where "n" is the number of the register and "z" is the word size of the data that is pointed to. If no word size is specified, the address register points to a structure of some kind that can not be labelled by a specific word size, or the word size of the data pointed to is irrelevant.

The distinction between a TRAP call and a syslist call can be seen in the semi-assembly-like description. If the function name is preceded by TRAP it is a TRAP call. If it is preceded by JSR it is a syslist call.

FUNCTION A longer description of what this function does and how to use it. Possibly with details of its implementation.

INPUTS A list of the input parameters to this function and a description of them.

RESULTS A note about the result that this function produces and what return values are given. When a two-sized return value is given by the Dn.z.e notation, and "e" is larger than ”z”, only the word size ”z” is valid unless the error code equals the value in the word of size ”e”.

EXAMPLES An example of how to use this function, possibly together with other functions.

NOTE A special note about this function that needs to be observed.

SEE ALSO Other functions related to this function.

All functions do not necessarily have all of the above fields. The two types of system calls use the following syntax:
TRAP calls: `TRAP #functionname`
Syslist calls: `JSR (FunctionName,An)`
For syslist calls `An` must contain the value of address 4 in memory. All parameters use registers. The parameters are placed in the registers before the actual call. The stack is not used for parameters in ExOS. Note that the synopsis does not reflect the correct syntax but is instead a simplified way of showing how the functions work and the parameters it needs and values it returnes.

E.1 AllocMem

NAME AllocMem – Allocate a block of memory

SYNOPSIS

```
A0 = TRAP #allocmem(D0.L)
void* memoryBlock = AllocMem(long size);
```

FUNCTION Allocates a block of memory for use by a program. All programs that require storage other than that compiled into the program as static data needs to use this function to gain exclusive access to a memory block of a specified size. No prior knowledge to where this block will be in memory is available and no assumptions about its location must be made. All memory allocated with AllocMem must be freed using FreeMem before the process exits.

INPUTS `size (D0.L)` – The number of bytes requested is placed in D0. This number is rounded up to the nearest even number which means that the block actually allocated will always be an even number of bytes.

RESULTS `memoryBlock (A0)` – A pointer to the memory block allocated or NULL if the block could not be allocated for some reason. memoryBlock always points to an even address.

NOTE This function is forbidden to be called in supervisor mode, i.e. interrupts, TRAPs, exception handlers, etc.

SEE ALSO FreeMem, CompFreeList

E.2 AllocSignal

NAME AllocSignal – Allocate a signal bit

SYNOPSIS

```
D0.B.L = JSR AllocSignal
ulong sigbit = AllocSignal(void);
```

FUNCTION Allocate a signal bit for use by a process to connect to an OS function or another process that this can use to notify the process. 24 user signal bits are available. Allocated signals can be freed using FreeSignal. As signals are local to the process they need not to be freed before the program exits. A signal need not to be allocated in order to be used but if different parts of a program needs signals AllocSignal provides a comfortable way of maintaining which bits are used and which are not. Waiting for and receiving signals is made with Block.

RESULTS

`sigbit (D0.B)` – A number between 0 and 23 identifying the bit allocated.

`error (D0.L)` – If all signals were already allocated, -1 is returned in D0.L.

EXAMPLE Allocating a sigbit and make Block wait for this signal. Then free it using FreeSignal:

```
MOVE.L SYSLIST,A5
JSR    (AllocSignal,A5)
MOVE.B D0,D1 *store our allocated sigbit in D1
MOVEQ  #0,D0 *alternatively we could have written MOVEQ #1,D0 here..
BSET.L D1,D0 *..and written LSL.L D1,D0 here.
JSR    (Block,A5)
*Block has now returned and we need not check D0 as we only waited for
*one signal, ours, so only this one could have caused Block to return.
MOVE.B D1,D0
JSR    (FreeSignal,A5)
```

NOTE The sigbit is a bit number that cannot be used directly by Signal or Block but must be used to set one bit in a bitmask that can be sent to Signal or Block.

SEE ALSO FreeSignal, Block, Signal

E.3 Block

NAME Block – Suspend a process until a signal arrives

SYNOPSIS

`D0.L = JSR Block(D0.L)`

`ulong recvSigMask = Block(ulong waitSigMask);`

FUNCTION This function first checks to see if any of the bits in the waitSigMask has already arrived to the process. Block will in this case return immediately with these bits set in recvSigMask. If none of the bits in waitSigMask had arrived, Block will put the process in the waiting state and search for another process that is in the ready state. If such a process is found it will put this process in the running state. If no process is found in the ready state a STOP instruction will be executed and the CPU halts waiting for an interrupt. When an interrupt has arrived and been processed, the STOP instruction will end and the list of processes will be searched again for a process that might have been put in the ready state by the interrupt.

INPUTS `waitSigMask (D0.L)` – A mask of bits that Block shall wait for.

RESULTS `recvSigMask (D0.L)` – A mask with one or more of the bits in the waitSigMask set.

NOTE If waitSigMask is equal to 0, i.e. no bits are set, Block will never return and the process will be in the waiting state forever.

SEE ALSO Signal, AllocSignal, FreeSignal

E.4 CompFreeList

NAME CompFreeList – Compresses the list of free memory

SYNOPSIS

```
JSR CompFreeList
void CompFreeList(void);
```

FUNCTION Merges all memory blocks that are marked as free and that lies continously in the list, which makes blocks that were many and small into one large block.

RESULTS The memory has been defragmented which results in larger free blocks. No return value.

NOTE This function is forbidden to be called in supervisor mode, i.e. interrupts, TRAPs, exception handlers, etc.

SEE ALSO AllocMem, FreeMem.

E.5 Connect

NAME Connect – Connect a device to an event

SYNOPSIS

```
JSR Connect(D0.B, D1.W, D2.B, D3.L, A0)
void Connect(char device, short unitNr, char type,
                          long uAddress, long iAddress);
```

FUNCTION Connects a device (e.g. DEV_TIMER) with an event. When a timer times out the program that owns it must be aware of this in some way. Connect offers two ways to solve this, either via a signal or via a handler.

INPUTS

`device (D0.B)` – The type of device that should be connected. There are defined constants that should be used (e.g. DEV_TIMER).

`unitNr (D1.W)` – The unit number is a number that distinguishes two or more devices of the same type. There cannot be two devices with the same device type and the same unit number.

`type (D2.B)` – Is one of two types of methods to make programs aware of an event. The two types are signal (CON_SIG) and handler (CON_HANDL).

`uAddress (D3.L)` – An address to a memory place where the user program stores data or it is a signal number that is allocated by the user program. Which one it is depends on the choice of type above.

`iAddress (A0)` – An address to a handler or a PCB. The handler takes care of what should be done when an event occurs. The PCB is used for knowledge needed to switch to right process.

type	CON_HANDL	CON_SIG
uAddress	ptr to userdata	signal mask
iAddress	ptr to handler	ptr to PCB

Table E.1: Connect parameters iAddress and uAddress

RESULTS The result of calling Connect is that a device is connected to an event. No return value.

SEE ALSO Event, Timer.

E.6 Debug

NAME Debug – Print current PC, SR, USP and SSP to serial

SYNOPSIS

```
TRAP #debug
void Debug(void);
```

FUNCTION Prints the current program counter, status register and both stack pointers as hexadecimal numbers to the serial port. Useful for debugging purposes.

E.7 Delay

NAME Delay – Busy-loop for a given period of time

SYNOPSIS

```
JSR Delay(D0.W,D1.L)
void Delay(ushort seconds, long micros);
```

FUNCTION Use the CPU to wait for up to 65536 seconds. Useful for debugging purposes to slow a program down without the need to use some kind of hardware timer. Delay is not very accurat but will estimate the time based on the current CPU clock speed. If called while multitasking is enabled the time is very unreliable but will never be shorter than the amount specified.

INPUTS

`seconds (D0.W)` – The number of seconds to wait.

`micros (D1.L)` – The number of microseconds to wait. It is allowed to make this number equal or larger than 1,000,000 microseconds, which is equal to one second.

NOTE If the CPU clock speed is not set properly, Delay may not wait as long as expected.

E.8 Event

NAME Event – Handles an event

SYNOPSIS

```
JSR Event(D0.B, D1.W)
void Event(char device, short unitNr);
```

FUNCTION Checks the connection list for which type of event handler that is connected to the device with the device type and unit number that is passed at call.

INPUTS

`device (D0.B)` – The type of device that should be connected. There are defined constants that should be used (e.g. DEV_TIMER).

`unitNr (D1.W)` – The unit number is a number that distinguishes two or more devices of the same type. There cannot be two devices with the same device type and the same unit number.

RESULTS The device with device and unit number passed, is signaled in some way. No return value.

SEE ALSO Connect

E.9 FlushRx

NAME FlushRx – Empty the serial receive buffers

SYNOPSIS

```
JSR FlushRx
void FlushRx(void);
```

FUNCTION Reset the serial receive buffers in RAM and the serial circuit. Previously received data that has not been read by a program or such will be lost.

SEE ALSO FlushTx

E.10 FlushTx

NAME FlushTx – Empty the serial transmit buffers

SYNOPSIS

```
JSR FlushTx
void FlushTx(void);
```

FUNCTION Wait until the serial transmit buffers in RAM and in the serial circuit are empty. When Flush returns you can be sure that all data previously written to the serial port has been transmitted over the line.

SEE ALSO FlushRx, SendS, SendA, PutS, PutStr

E.11 FreeMem

NAME FreeMem – Free allocated memory previously allocated by AllocMem

SYNOPSIS

```
TRAP #freemem(A0)
void FreeMem(void* address);
```

FUNCTION Frees a memory block allocated by AllocMem. Returned memory block is only made available in whole and not merged with another, continous memory block. This may make the free memory more and more fragmented. To defragment the memory a compression of it must be done.

INPUTS `address (A0)` – The address to memory that should be freed.

NOTE If the address is not an address previously returned by AllocMem, nothing will happen. Freeing a faulty pointer is in other words harmless. This function is forbidden to be called in supervisor mode, i.e. interrupts, TRAPs, exception handlers, etc.

SEE ALSO CompFreeList, AllocMem

E.12 FreeSignal

NAME FreeSignal – Free a signal bit previously allocated by AllocSignal

SYNOPSIS

```
JSR FreeSignal(D0.B)
void FreeSignal(char sigbit);
```

FUNCTION Free a signal in a process that has been previously allocated by AllocSignal. It is not necessary to free signals before a process exits as the signals are local to the process and will thus die with it. Freeing a signal is useful if the use for it has ceased but it might be needed again later in the program.

INPUTS `sigbit (D0.B)` – A bit number between 0-31 that was returned by AllocSignal. If a value greater than 31 is supplied it will be truncated to five bits which allows a number no greater than 31. A signal mask is thus not what this function expects, but a signal number.

RESULTS The sigbit has been freed and can be allocated by AllocSignal again. No return value.

SEE ALSO AllocSignal, Block, Signal

E.13 GetA

NAME GetA – Get a byte, if any, from the serial

SYNOPSIS

```
D0.B.L = JSR GetA
char byte = GetA(void);
```

FUNCTION Query the serial structures if there is any data pending. If so, read one byte and return. Else, return an error. GetA will always return immediately and is thus useful if polling the serial port is necessary or if it is required that the function returns immediately without blocking the caller. Generally GetS should be used which lets other processes run while waiting for data.

RESULTS `byte (D0.B.L)` – If a byte is present in the receive FIFO buffer it will be fetched and returned in D0.B. If D0.L is not equal to -1 only D0.B is valid and contains the byte read. If no data is present in the receive FIFO D0.L will be equal to -1.

NOTE This function may be relayed as a general input function in the future, able to read from any input device.

SEE ALSO GetS, ReadS, PutS

E.14 GetEnvToA6

NAME GetEnvToA6 – Obtain the pointer to the environment variables.

SYNOPSIS

```
A6 = JSR GetEnvToA6
EnvVars* envvars = GetEnvToA6(void);
```

FUNCTION The address to envvars will be put in the register A6. Used mainly by system software as only the system stores information in envvars. No assumptions about the whereabouts of envvars must be made. This function ensures a correct address even when RAM and ROM has been swapped.

RESULTS `envvars (A6)` – The address to envvars.

E.15 GetS

NAME GetS – Get a byte from the serial

SYNOPSIS

```
D0.B = JSR GetS
char byte = GetS();
```

FUNCTION Read a byte from the serial. If no data is present at the time of the call GetS will block until atleast one byte arrives.

RESULTS `byte (D0.B)` – A byte read.

NOTE This function may be relayed as a general input function in the future, able to read from any input device.

SEE ALSO GetA, ReadS, PutS

E.16 GetSerInfo

NAME GetSerInfo – Obtain information about the serial port

SYNOPSIS

```
A0.L = JSR GetSerInfo(A0.L)
TagItem* serinfo = GetSerInfo(TagItem* serinfo);
```

FUNCTION Takes a taglist with requests for information. This taglist will be filled with the requested information.

INPUTS `serinfo (A0.L)` – A writable taglist containing requests for information. The following tags are currently supported by GetSerInfo: SI_SPEED, SI_LINESTATUS, SI_MODSTATUS, SI_FIFOSIZE, SI_RXFIFOLOC, SI_TXFIFOLOC, SI_RXFILLLEV, SI_TXFILLLEV, SI_RXTOTAL, SI_TXTOTAL. Unknown tags will be ignored.

RESULTS `serinfo (A0.L)` – The same taglist sent as input filled with data. Table E.2 shows the available tags and the results they give.

SEE ALSO SetSerSpeed

E.17 IncLed

NAME IncLed - Increase the binary number on the 8 LEDs

SYNOPSIS

```
JSR IncLed
void IncLed(void);
```

Code / Tag	Data / Return value
SI_SPEED	Word: Speed in divisor values. I.e. one of SER_16Xspeed if no custom speed has been set.
SI_LINESTATUS	Byte: Copy of the line status register
SI_MODSTATUS	Byte: Copy of modem status register
SI_FIFOSIZE	Long: The size of the FIFOs
SI_RXFIFOLOC	Long: Address of the receive FIFOs in RAM
SI_TXFIFOLOC	Long: Address of the transmit FIFOs in RAM
SI_RXFILLLEV	Long: Number of bytes currently in the receive FIFO
SI_TXFILLLEV	Long: Number of bytes currently in the transmit FIFO
SI_RXTOTAL	Long: Total number of bytes transferred to the receive FIFO
SI_TXTOTAL	Long: Total number of bytes transferred to the transmit FIFO

Table E.2: GetSerInfo tags

FUNCTION Read the current LED-value from envvars, increase the value by one and output the new value to the LED register on the main board. The new value is also written back to envvars.

E.18 InitSerial

NAME InitSerial - Initiate the serial circuit

SYNOPSIS

```
JSR InitSerial
void InitSerial(void);
```

FUNCTION Detect, reset and initiate the serial circuit. InitSerial is executed on startup. It performs a read-write-read test to see if the serial IC is present. If it is present, it checks envvars .status to see if it has already been initiated. If it has not, it will be reset and the speed will be set to 9600bps. If it has been initiated already, a reset and speed set will not be done. The FIFO pointers and counters will be initiated. The status LEDs on the serial card will be set. The FIFOs in the 16C550 will be reset and enabled. The receive interrupt will be turned on. Envvars.status will be set to signal that the serial IC has been initiated.

SEE ALSO GetSerInfo

E.19 Int2Dec

NAME Int2Dec – Convert a binary value into a decimal value

SYNOPSIS

```
JSR Int2Dec(D0.L, A0.B)
void Int2Dec(long value, char* buffer);
```

FUNCTION This function takes a long value and converts it into a string of up to ten decimal numbers. If a buffer is passed in, the characters are written to this buffer with a trailing null-character. The buffer must be able to store up to 12 bytes (10 digits, possible minus sign and a null-character) unless the caller is sure less digits will be written. No preceding zeroes will be written. If the buffer is a null-pointer the characters will be sent directly to the serial port to ease debugging of software.

INPUTS

`value (D0.L)` – A signed 32-bit value to convert into an ASCII string.

`buffer (A0.B)` – A pointer to a buffer in which to store the converted number. This buffer must be large enough to hold up to 12 bytes. If the user is sure less characters will be written a smaller buffer may be used. If the buffer is null the converted string will be written directly to the serial port.

SEE ALSO Int2Hex, Str2Int

E.20 Int2Hex

NAME Int2Hex – Convert a binary value into a hexadecimal value

SYNOPSIS

```
JSR Int2Hex(D0.L, A0.B)
void Int2Hex(long value, char* buffer);
```

FUNCTION This function takes a long value and converts it into a string of eight hexadecimal numbers. If a buffer is passed in, the 8 hex-characters are written to this buffer without a trailing null-character. The buffer must be able to store 8 bytes. If the value does not fill out 8 characters, preceding zeroes will be padded. If the buffer is a null-pointer, the characters will be sent directly to the serial port to ease debugging of software.

INPUTS

`value (D0.L)` – A 32-bit value to convert into an ASCII string.

`buffer (A0.B)` – A pointer to a buffer in which to store the converted number. This buffer must be large enough to hold exactly 8 bytes. If the buffer is null the converted string will be written directly to the serial port.

NOTE No trailing null-termination will be appended.

SEE ALSO Int2Dec, Str2Int

E.21 MemCheck

NAME MemCheck – Checks the memory for faults

SYNOPSIS

```
D0.L = JSR MemCheck(A0, D0.L)
long result = MemCheck(long startAddress, long amount);
```

FUNCTION Checks the memory from the start address and forward the amount of bytes. If the start address is even, every even byte is checked and vice versa. This way each of the memory ICs can be checked one at a time. If there is one byte that is not ok this is reported, but not which one it is.

INPUTS

`startAddress (A0)` – The address where the checking shall begin.

`amount (D0.L)` – The amount of bytes to check.

RESULTS `result (D0.L)` – If the memory is defect the result will be equal to 1, and if the memory is OK the result will be equal to 0.

NOTE As only even or odd bytes are checked each time, amount will effectively be in number of words, not bytes, although only one byte of every word will be tested in one pass.

E.22 MemInfo

NAME MemInfo – Gives information about memory

SYNOPSIS

```
A0.L = JSR MemInfo(A0.L)
TagItem* meminfo = MemInfo(TagItem* meminfo);
```

FUNCTION Gives information about how much free memory there is, size of the largest block of free mem, how many blocks that are allocated and how many blocks that are free.

INPUTS `meminfo (A0.L)` – The taglist is built up by one command followed by one field for the result, then another pair and so on. The taglist must end with a 0 as command. There can be as many pairs as there are commands.
Four commands exists and are listed in table E.3.

command	Description	Return value
MI_TOTALFREE	How much free memory there is.	Number of free bytes
MI_LARGEST	The largest block of free memory.	Size in bytes
MI_NUMALLOC	Amount of blocks that are allocated.	Number of blocks
MI_NUMFREE	Amount of blocks that are free.	Number of blocks

Table E.3: MemInfo tags

RESULTS `meminfo (A0.L)` – The address to the taglist. The actual result is stored in the fields after each command in the taglist. The result is either a size in bytes or a number in integers. See above.

SEE ALSO AllocMem, FreeMem, CompFreeList

E.23 PutS

NAME PutS – Write a byte synchronously to the serial

SYNOPSIS

```
JSR PutS(D0.B)
void PutS(char byte);
```

FUNCTION Sends one byte of data to the serial circuit or the transmit FIFO buffer if the serial circuit is currently busy. If the FIFO is full it will Block until the FIFO is not full. It will then put the byte in the FIFO and return.

INPUTS `byte (D0.B)` – A byte of data to send over the serial port.

NOTE This function may be relayed as a general output function in the future, able to write to any output device.

SEE ALSO SendS, GetS, GetA

E.24 PutStr

NAME PutStr – Send a string synchronously to the serial

SYNOPSIS

```
JSR PutStr(A0)
void PutStr(char* string);
```

FUNCTION Sends a null-terminated string of any size to the transmit FIFO buffer. If the FIFO is full or becomes full during the call it will Block until the FIFO is not full. It will continue to Block until the whole string has been written to the FIFO. It will then return.

INPUTS `string (A0)` – A null-terminated string that will be written.

NOTE This function may be relayed as a general output function in the future, able to write to any output device.

SEE ALSO SendS

E.25 ReadS

NAME ReadS – Read a number of bytes synchronously from the serial

SYNOPSIS

```
A0.B = JSR ReadS(A0.B, D0.L)
char* buffer = ReadS(char* buffer, ulong size);
```

FUNCTION Reads a specified amount of data from the serial port and puts it in a buffer. The buffer must be big enough to be able to store the specified number of bytes that should be read. This function is synchronous which means that if not enough data is in the receive FIFO when called it will read what data it can and then Block until enough data has arrived to fill up the buffer passed in to the size specified.

INPUTS

`buffer (A0.B)` – Buffer to store the data read.

`size (D0.L)` – The size in bytes of the buffer that will be filled with data.

RESULTS `buffer (A0.B)` – The same buffer passed in filled up with data.

NOTE This function may be relayed as a general input function in the future, able to read from any input device.

SEE ALSO SendS, SendA, GetS, GetA

E.26 ReSchedule

NAME ReSchedule – Private function

SYNOPSIS

```
TRAP #reschedule
void ReSchedule(void);
```

FUNCTION This internal TRAP call will setup the kernel and the currently running process, if any, for a task switch. It will then do a rescheduling if it was called from user mode. If it was called from supervisor mode it will install an interrupt catcher to intercept the last supervisor return and do a recheduling then. If no process wants to run, ReSchedule will execute a STOP instruction to halt the CPU and wait for an interrupt. Only Block and Signal are allowed to call this function as they are the only one that can guarantee a pure stack. A pure stack means that only the return address to the user program may be on the stack, no other nested system calls or stored registers. The function will store the current status of the running user program and put it in the ready state if it was running. It will then find another ready process to start up. This function may however not be the one reviving the current process again. Also it may not have been this function that stored the new process in previous time. That is why the stack must be pure so that no dependencies exists regaring the return point.

NOTE User programs must not call this function but instead use the Block function.

E.27 Reset

NAME Reset – Restart dBOX and ExOS

SYNOPSIS

```
TRAP #reset
void Reset(void);
```

FUNCTION Will perform a hardware reset on the main board and software reset on the OS and the CPU. The 68000 CPU does not have an instruction for resetting itself, only to activate the external reset pin on the CPU to reset external devices connected to this pin. This trap call will simply do the steps normally done by the CPU on an external hardware reset. That is; read the system stack pointer from address $0, read the entry point to the startup code from address $4, then jump to the entry point. The result is a warm restart.

E.28 SendA

NAME SendA – Write a number of bytes asynchronously to the serial

SYNOPSIS

```
D0.L = JSR SendA(A0.B, D0.L)
ulong bytesWritten = SendA(char* buffer, ulong size);
```

FUNCTION Try to write a specified number of bytes to the serial port. This function is asynchronous which means that it will not Block if it could not fit the complete buffer in the transmit FIFO at once but will instead return immediately with the actual number of bytes written.

INPUTS

`buffer (A0.B)` – A buffer containing a specific amound of data to try to send over the serial port.

`size (D0.L)` – The number of bytes requested to be sent.

RESULTS `bytesWritten (D0.L)` – This return value will tell how many bytes could actually be fit into the transmit FIFO and that thus has been sent.

NOTE This function may be relayed as a general output function in the future, able to write to any output device.

SEE ALSO SendS, PutStr, PutS

E.29 SendS

NAME SendS – Write a number of bytes synchronously to the serial

SYNOPSIS

```
JSR SendS(A0.B, D0.L)
void SendS(char* buffer, ulong size);
```

FUNCTION Will write a buffer with the specified number of bytes to the serial port. If the complete buffer does not fit into the transmit FIFO buffer, SendS will Block until enough space has been available to write the whole buffer into the FIFO.

INPUTS

`buffer (A0.B)` – A buffer containing a specific amound of data to send over the serial port.

`size (D0.L)` – The number of bytes to send.

NOTE This function may be relayed as a general output function in the future, able to write to any output device.

SEE ALSO SendA, PutStr, PutS

E.30 SetSerSpeed

NAME SetSerSpeed – Set the transfer speed of the serial port

SYNOPSIS

```
JSR SetSerSpeed(D0.W)
void SetSerSpeed(short divisor);
```

FUNCTION Setting the speed of the serial circuit is made using this function. The speed is defined as a divisor value that will divide the 1/16'th of the crystal frequency further. The 16C550 has a crystal of 15.360MHz connected to it. This frequency is internally divided by 16 to form a frequency of 960kHz. This 960kHz frequency is further divided by the input parameter to SetSerSpeed to form the transfer speed desired.

INPUTS `divisor (D0.W)` – Divisor value to divide the 1/16'th crystal frequency with. There are also a set of predefined divisor values that can be used which are listed in table E.4.

EXAMPLES To set the speed to 19200bps without using a predefined value, the divisor is calculated the following way:

$$divisor = \frac{crystal/16}{speed}$$

Divisor	Speed (bps)	Note
SER_16X2400	2400	
SER_16X4800	4800	
SER_16X9600	9600	
SER_16X19200	19200	
SER_16X38400	38400	
SER_16X57600	57600	Not exactly 57600 but 56471
SER_16X115200	115200	Not exactly 115200 but 120000

Table E.4: Predefined serial transfer speeds

In this case this will be:

$$\frac{15360000/16}{19200} = 50$$

The divisor value to send to this function will be 50.

SEE ALSO GetSetInfo

E.31 Signal

NAME Signal – Signal a process

SYNOPSIS

```
D0.L = JSR Signal(A0, D0.L)
```

```
bool success = Signal(struct ProcessControlBlock* pcb, ulong sigmask);
```

FUNCTION Signal a process by sending it a set of signal bits. If the process signalled is currently waiting for any of these signals in a Block a rescheduling will be made. It is not certain that the process signalled will be the one resuming execution after the rescheduling if there are other ready processes in the process list. What is certain is that the signalling process will stop executing if atleast one other process is ready in the process list, the signalled process is waiting for atleast one of the signals sent and multitasking has not been inhibited.

INPUTS

`pcb (A0)` – A pointer to the process to receive the signals.

`sigmask (D0.L)` – A bitmask containing signal bits to send to a process. The upper 8 bits should not be used by user programs. These are reserved and used by ExOS and may cause confusion if used. No harm will come if used but to avoid peculiarities only the lower 24 bits, allocatable by AllocSignal should be used.

RESULTS `success (D0.L)` – If the process to signal existed then -1 (true) will be returned. If the process to signal did not exist in the process list, 0 (false) will be returned.

NOTE This function may be called from supervisor mode.

SEE ALSO AllocSignal, Block

E.32 SingleTask

NAME SingleTask – Enable or disable multitasking

SYNOPSIS

```
JSR SingleTask(D0.B)
void SingleTask(bool singleTask);
```

FUNCTION When it is critical that a user program does not get interrupted by the rescheduler, this function can be used to disable the multitasking. When the program no longer needs to singletask it should turn on multitasking again as soon as possible to allow other processes to execute. If a program enters singletask mode for an extended period of time without doing any output it would appear to the end user as if the computer has hung. This is why singletask mode should be used sparingly and only for short periods of time.

INPUTS `singleTask (D0.B)` – Set to non-zero to disable multitasking and enter single-task mode. Set to 0 to enable multitasking.

NOTE Interrupts will not be inhibited in singletask mode, only rescheduling. A call to Block will immediately cancel singletask mode and enable multitasking. When Block returns (because a signal arrived) multitasking will not be turned off again.

E.33 Stop

NAME Stop – Stop the CPU

SYNOPSIS

```
TRAP #stop
void Stop(void);
```

FUNCTION Stop intruction fetching and wait for an interrupt. This function will execute the STOP-instruction of the CPU which will make the CPU cease instruction fetching and execution and wait for an interrupt. Not very userful for user programs. It is used by ExOS when Block is called and the kernel is not running. As STOP is a privileged instruction user code cannot execute it but needs a TRAP call to get in supervisor mode first. When no program needs to run, Stop provides a way of halting the CPU and thus put the rest of the computer in low-power standby mode until an interrupt occurs. Busy-looping is never a good idea. The STOP-instruction will set the status register to $2000 which means supervisor mode (cannot return from the TRAP without being in supervisor mode) and lowest interrupt priority.

NOTE User programs should use Block instead when they have nothing to do to let other programs execute while waiting for a signal. Block will execute STOP if no other program wants to execute.

SEE ALSO Block

E.34 StoreProg

NAME StoreProg – Convert and store an S19 file into binary

SYNOPSIS

`A0, D0.L = JSR StoreProg(A0)`

`char* dataBlock, long progSize = StoreProg(LinkedList* s19List);`

FUNCTION Converts a linked list of S19 records into binary. First the linked list is scanned to calculate the size the converted data will take. Then a memory block will be allocated in which the decoded data will be stored. S19 records of type S1, S2 and S3 will then be converted and stored within this block.

INPUTS `s19List (A0)` – A linked list where each node contains one S19 record. One S19 record also equals one line in an S19 file. The linked list must end with a next pointer that is null and has the following form:

```
struct LinkedList {
  char*  s19record;
  struct LinkedList* next;
}
```

RESULTS

`dataBlock (A0)` – The block in memory that was allocated and filled with binary data converted from the S19 file. It is up to the caller to free this block when it is no longer needed. If an error occured this will be null.

`progSize (D0.L)` – If no error occured this return parameter will contain the size in bytes of the binary data block returned. If memory could not be allocated 0 will be returned. If a checksum error was detected during decoding -1 will be returned. If none of the S19 records containd any data -2 will be returned.

NOTE Although this function allocates a memory block to store the decoded data in it still interprets the addresses in every S19 record. This means that if the addresses in all records are not adjecent, because an ORG was used somewhere in the assembly source, StoreProg will most likely write outside the allocated memory block. The allocated memory block will only be used as a base address. Therefore *code written for ExOS must always be written relocatable, never use fixed addresses (ORG) and always use a start address of 0* for normal operation.

E.35 Str2Int

NAME Str2Int – Convert a string with a number to a binary value

SYNOPSIS

```
D0.L, D1.L = Str2Int(A0.B)
long value, long length = Str2Int(char* string);
```

FUNCTION Converts a number represented as ASCII in a string to a normal binary value. Any leading white spaces will be skipped and ignored. The value can be either decimal with no preceding character, hexadecimal with a $-sign as preceding character or binary with a %-sign as a preceding character. Negative value are representing by a '-'-sign preceding the number. The occurence of a character not representing a digit within a number in the string will stop the parsing and return what has been found. If the number was too big for 32-bit 0 will be returned in both value and length.

INPUTS `string (A0.B)` – A pointer to a string to parse for a number.

RESULTS

`value (D0.L)` – The converted value. If no number was found in the string or the number did not fit in 32-bit 0 will be returned.

`length (D1.L)` – The number of approved characters read. This includes leading white spaces and preceding signs. If no number was found in the string or the number didn't fit in 32-bit 0 will be returned.

EXAMPLES Strings that will parse correctly plus the return value and length: `"45" (45,2)`, `"-$8C " (-140,4)`, `" -456abc78" (456,6)`, `" %01110101 " (117,11)`, `" -3.14" (-3,3)`

Strings that will not parse correctly and return 0 in both return values: `"abc"`, `"- 56"`, `"$ B7"`, `" % 101"`, `"K5"`

SEE ALSO Int2Dec, Int2Hex

E.36 StrCmp

NAME StrCmp – Compare two strings in a C-like manner

SYNOPSIS

```
D0.W = JSR StrCmp(A0.B, A1.B, D0.L)
short result = StrCmp(char* str1, char* str2, long length=-1);
```

FUNCTION Compare two strings and return the difference between them in the form of the difference between the first non-equal characters. If null was encountered in both strings at the same position before any non-equal characters, zero will be returned. This function works like the strcmp() function in C. It can be used in sort routines. It will always stop comparing if null is encountered in one of the strings.

INPUTS

`str1 (A0.B)` – The first null-terminated string to compare.

`str2 (A1.B)` – The second null-terminated string to compare.

`length (D0.L)` – An optional length parameter to be used if the comparison should not proceed until a null character arrives in one of the strings. If a comparison until null is desired this parameter is set to -1. If length is set to 0 no characters are compared and 0 (equal) is returned. If length happens to be longer than any of the strings, the comparison will stop when a null character is encountered as if length was set to -1.

RESULTS `result (D0.W)` – This will be 0 if the two strings were equal up until null characters were detected in both strings or the specified number of characters have been compared and found equal. A value less than zero will be returned if a character was encountered in str1 that has a lower ASCII value that the character on the same position in str2. A value greater than zero will be returned if a character was encountered in str1 that has a higher ASCII value than the character on the same position in str2. The magnitude of the value is derived by subtracting the ASCII value of the last compared character in str2 from the last compared character in str1.

EXAMPLES In these examples '\0' will illustrate terminating null characters in the strings and bold font illustrates the characters where the comparison stops. Following are a few examples of calls to StrCmp and their return values:
StrCmp("Hello\0", "Hello\0") = 0
StrCmp("Hello **p**eople\0", "Hello **w**orld\0") = 7
StrCmp("**t**est\0", "**T**est\0") = -32
StrCmp("testin**g** first\0", "testin**g** second\0", 7) = 0
StrCmp("Some string\0", "another string\0", 0) = 0
StrCmp("Str**i**ng\0", "Str\0", 50) = 105

NOTE In assembly the length parameter may not be omitted as detecting an absent parameter is impossible in assembly.

SEE ALSO StrCmpNC

E.37 StrCmpNC

NAME StrCmpNC – Compare two string case insensitive

SYNOPSIS

`D0.L = JSR StrCmpNC(A0.B, A1.B, D0.L)`

`bool equal = StrCmpNC(char* str1, char* str2, long length=-1);`

FUNCTION Compare two strings non case sensitive. This function differs from StrCmp on two things: It ignores if letters are in UPPER CASE or lower case but treats them the same. It also does not return a differential value but rather a bool telling if the strings were equal or different up until the length specified, or until a null character was encountered in one of the strings.

INPUTS

`str1 (A0.B)` – The first null-terminated string to compare.

`str2 (A1.B)` – The second null-terminated string to compare.

`length (D0.L)` – An optional length parameter to be used if the comparison should not proceed until a null character arrives in one of the strings. If a comparison until null is desired this parameter is set to -1. If length is set to 0 no characters are compared and -1 (equal) is returned. If length happen to be longer than any of the strings the comparison will stop when a null character is encountered as if length was set to -1.

RESULTS `equal (D0.L)` – -1 is returned if both strings were equal (case ignored). 0 is returned if the two strings differs in length or characters (other than case).

EXAMPLES In these examples '\0' will illustrate terminating null characters in the strings and bold font illustrates the characters where the comparison stops. Following are a few examples of calls to StrCmpNC and their return values:

StrCmpNC("Hello\0", "Hello\0") = -1

StrCmpNC("Hello **p**eople\0", "Hello **w**orld\0") = 0

StrCmpNC("test\0", "Test\0") = -1

StrCmpNC("TESTin**g** first\0", "testin**G** second\0", 7) = -1

StrCmpNC("Some string\0", "another string\0", 0) = -1

StrCmpNC("Str**i**ng\0", "Str\0", 50) = 0

NOTE Only the letters a-zA-Z are covered by the case insensitivity. Non-english letters like ö or ä does not match against their counterparts in another case.

SEE ALSO StrCmp

E.38 StrCopy

NAME StrCopy – Copy one string into another

SYNOPSIS

```
JSR StrCopy(A0.B, A1.B)
void StrCopy(char* dest, char* source, long length=-1);
```

FUNCTION Copy a null-terminated source string over a destination string that also will be null-terminated. The length parameter can be omitted and the copy will proceed until a null character is encountered in the source string. It is strongly suggested though, that the length parameter is used to prevent overflowing the destination memory buffer. This function works like the strcpy() function in C.

INPUTS

`dest (A0.B)` – The destination string that will be overwritten by the source string. The memory buffer dest points to must be large enough to hold the entire source string plus one additional character for null- termination, or the length specified. It will always be null-terminated unless the length parameter specifies 0 characters.

`source (A1.B)` – The null-terminated string that will be copied to the destination.

The copying will proceed until a null character is encountered or as long as the length parameter specifies.

`length (D0.L)` – This optional parameter tells how many characters that the destination string can hold including a trailing null character. If this parameter is set to -1 the caller must be certain that the source string will fit in the destination. The source string does not have to be null-terminated if length is used. If length defines a longer string than the source is, the copying will stop when a null character is encountered in the source. If length is 0 no characters will be copied and no null-termination will be written to the destination.

EXAMPLES In these examples '\0' will illustrate terminating null characters in the strings. Following are a few examples of calls to StrCopy:

StrCopy(dest, "Test\0") – dest="Test\0"

StrCopy(dest, "Short string\0", 5) – dest="Shor\0"

SEE ALSO StrLen

E.39 StrLen

NAME StrLen – Count the number of characters in a string

SYNOPSIS

```
D0.L = JSR StrLen(A0.B)
ulong length = StrLen(char* string);
```

FUNCTION Counts the number of characters in a string up until a null character is encountered. The null character will not be counted. This function works like the strlen() function in C.

INPUTS `string (A0.B)` – The null-terminated string to count the characters in.

RESULTS `length (D0.L)` – The number of characters in the string up until, but not including, the terminating null character.

EXAMPLES '\0' illustrates the terminating null character.

StrLen("One, two three\0") = 14

SEE ALSO StrCopy, StrCmp, StrCmpNC

E.40 SuperVisor

NAME SuperVisor – Ask if the calling code executes in supervisor mode

SYNOPSIS

```
Z = TRAP #supervisor
bool isSupervisor = SuperVisor(void);
```

FUNCTION As code executing in user mode does not have the privilege of being able to access the status register directly without being trapped in the privilege violation exception, this function was made. Using it makes it possible to find out if in supervisor mode or not without the risk of being trapped in the privilege violation exception. ExOS uses this function where Block might be called in supervisor mode to prevent it from doing so as this is forbidden.

RESULTS `isSupervisor (Z)` – The return value from this function does not come in a register but in the zero-flag. This means that a branch that depends on the Z-flag can be made directly after the call to this function without any previous CMP-instruction or similar test. The Z-flag will be set (1) if the calling code is executing in supervisor mode. The Z-flag will be cleared (0) if the calling code is executing in user mode.

EXAMPLES Example of a piece of code using this function:

```
TRAP  #supervisor
BEQ.S inSupervisorMode
BNE.S notInSupervisorMode

inSupervisorMode:
* code to execute in supervisor mode
BRA.S end

notInSupervisorMode:
* code to execute in user mode

end:
```

E.41 SwapRomRam

NAME SwapRomRam – Logically swap ROM and RAM

SYNOPSIS

```
TRAP #swapromram
```

```
void SwapRomRam(void);
```

FUNCTION Logically swaps the ROM and RAM memory spaces. This is needed if manipulation of the interrupt vector table is desired. SwapRomRam will set up everyting needed for the swap and then do the actual swap. All memory allocations that have been made will be changed to point to the new memory space. All user programs must change their own pointers and addresses that are set to point within the ROM or RAM. The top return address on all user stacks will be changed. Any subsequent return address on any stack caused by a subrountine call must be changed by the programs themselves. To make this as successful as possible it is advisable that no, or at most one program is running, preferably with an empty stack, when this function is called. If a swap has already been made this function will swap the memory areas back and perform a reset on the computer.

RESULTS The memory area at \$000000-\$0FFFFF has been swapped with the memory area at \$100000-\$1FFFFF.

NOTE Think twice before using this function.

E.42 Timer

NAME Timer – Manipulates a timer

SYNOPSIS

```
D0.L = TRAP Timer(D3.L)
```

```
long result = Timer(long command);
```

FUNCTION Via this function the user can manipulate a timer. The user can set, reset, register, unregister, read, start and stop a timer. The user can even initiate the timer device, BUT should be aware of that a normal user not should be doing this.

INPUTS `command (D3.L)` – The command that the user wants to execute. The commands that can be used are listed in table E.5 with extra parameters that must be passed with each of them. The results, were there are any, are listed too.

RESULTS See table E.5.

SEE ALSO Event, Connect

E.43 UserTrap

NAME UserTrap – Call a user function in supervisor mode

SYNOPSIS

```
TRAP #usertrap(A0)
```

```
void UserTrap(void* function);
```

Command	Input	Result
TC_RESET	`timerNr (D0.W)` – The number of the timer that should be reset.	None
TC_START	`timerNr (D0.W)` – The number of the timer that should be started.	None
TC_STOP	`timerNr (D0.W)` – The number of the timer that should be stopped.	None
TC_READ	`timerNr (D0.W)` – The number of the timer that should be read.	`currValue (D0.L)` – The current timer value.
TC_SET	`timerNr (D0.W)` – The number of the timer that should be set. `bitmask (D1.B)` – See appendix D.2.	None
TC_REG	None	`timerNr (D0.L)` – The timer number of the registered timer. A value of -1 means that there were no more timers.
TC_UNREG	`timerNr (D0.W)` – The number of the timer that should be unregistered.	None
TC_INIT	None	None

Table E.5: Timer parameters

FUNCTION This function takes a pointer to a subroutine, or function, that will be called from the TRAP. This means that the subroutine will execute in supervisor mode. This is mainly useful for maintenance software. The function will be called with a simple JSR-instruction. Nothing else then the return address the JSR puts on the stack, and of course the information the TRAP instruction puts on the stack will be there. Note that the stack used is the supervisor stack, not the user stack allocated for the user program itself. Multitasking will be disabled while this function executes as rescheduling cannot be done in supervisor mode.

INPUTS `function (A0)` – The function to call. The return must be made with a normal RTS-instruction, not RTE as one might have thought.

NOTE Registers used in the called function does not need to be preserved.

F Source code

Here follows the source codes for the whole project. Not including help software developed for various things, like making these sources TEX-friendly by wrapping them in `verbatim` etc. and replacing every blank line with ”-” to prevent LATEX from removing these lines all together. Nor does it include testing software. Most code was written using assembly but some parts were written in E, a high-level language similar to C. It is pretty clear who wrote what code as no coding standard was deployed, except for the naming of labels which are further discussed before the file MemLayout.s68 on page 104.

F.1 Definitions.s68

This file contains all constants and definitions used for the operating system stored in ROM. This is, not surprisingly, the most commented file in the project.

```
ROM     EQU 0
RAM     EQU $100000
SYSSP   EQU $140000
SSPSIZE EQU $1000
USPSIZE EQU $1000
MEMSIZE EQU $40000
NULL    EQU 0
NIL     EQU 0
TRUE    EQU -1
FALSE   EQU 0
SLOFF   EQU -6
-
IOA     EQU $200000 * These doesn't really exists yet but the
IOB     EQU $200002 * addresses are reserved in hardware.
IOC     EQU $200004
IOD     EQU $200006
IOE     EQU $200008
IOF     EQU $20000A
IOG     EQU $20000C
IOH     EQU $20000E
IOCTRL  EQU $200010 *  >Word only access!
LFC     EQU $20001C *  >
MCR     EQU $20001D *  >
DIP     EQU $20001E * <
LED     EQU $20001F *  >
-
*TRAP functions
allocmem     EQU 0  * >D0.L=size, <A0=mem
freemem      EQU 1  * >A0=mem
timer        EQU 2  * >D3.L=command
reschedule   EQU 3  * no parameters (callable only from block:)
stop         EQU 4  * no parameters
swapromram   EQU 5  * no parameters (all stacks must be clean for this to work)
debug        EQU 6  * no parameters (prints current PC, SR and SP)
supervisor   EQU 7  * <Z=0 if user mode, Z=1 if supervisor mode
usertrap     EQU 8  * >A0=ptr to function to be called and should end with RTE
reset        EQU 15 * no parameters
-
*SysList functions
Delay        EQU SLOFF*33  >D0.L=microseconds(max 1M), >D1.W=seconds
Event        EQU SLOFF*32
Connect      EQU SLOFF*31  >D0.B=Device (DEV_*), >D1.W=
InitSerial   EQU SLOFF*30  >A6=ptr to envvars, trashes various registers..
SetSerSpeed  EQU SLOFF*29  >D0.W=speed (SER_16Xspeed)
GetSerInfo   EQU SLOFF*28 <>A0.L=ptr to taglist that will be filled with data.
SendS        EQU SLOFF*27  >A0=data,  >D0.L(usig)=size  Send serial data synchronously
SendA        EQU SLOFF*26  >A0=data, <>D0.L(usig)=size  Send serial data asynchronously
GetS         EQU SLOFF*25 < D0.B=data  Get one byte serial data synch
GetA         EQU SLOFF*24 < D0.L(B)=data <D0.L=-1=no data  Get one byte serial data asynch if it exists
PutS         EQU SLOFF*23  >D0.B=data  Send one byte serial data synch
ReadS        EQU SLOFF*22 <>A0=buf, >D0.L(usig)=size  Read serial data synch
FlushTx      EQU SLOFF*21 no parameters
FlushRx      EQU SLOFF*20 no parameters
PutStr       EQU SLOFF*19  >A0=null-terminated string
StrCmp       EQU SLOFF*18  >A0.B=string1, >A1.B=string2, >D0.L=len, <D0.L=result:A0<A1=<0, A0=A1=0, A0>A1=>0
StrCmpNC     EQU SLOFF*17  >A0.B=string1, >A1.B=string2, >D0.L=len, <D0.L=result:A0<>A1=0, A0=A1=-1
StrLen       EQU SLOFF*16  >A0.B=ptr to 0-string, <D0.L=length excl. NIL
```

```
StrCopy      EQU SLOFF*15  >A0.B=dest, >A1.B=source, >D0.L=length
Str2Int      EQU SLOFF*14  >A0.B=string, <D0.L=int, <D1.L=length
Int2Dec      EQU SLOFF*13
Int2Hex      EQU SLOFF*12  >D0.L=int, >A0=buffer of 8 bytes to store the number, or NIL to output to serial
CompFreeList EQU SLOFF*11 no parameters
MemInfo      EQU SLOFF*10 >D0.B=0=total,1=largest,2=numalloc,3=numfree, <D0.L=avail size/num
MemCheck     EQU SLOFF*9  >A0.L=Start address, even or odd, >D0.L=Amount of bytes, D0.L> 0=OK,1=Not OK
StoreProg    EQU SLOFF*8  >A0=linked list of s-recs, <A0=addr to decoded program or NIL
GetEnvToA6   EQU SLOFF*7 < A6=ptr to envvars
IncLed       EQU SLOFF*6  >A0=ptr to envvars or 0 when >D7.B is used
SingleTask   EQU SLOFF*5  >D0.B <>0=forbid, =0=permit multitasking. Block automatically permits multitasking.
Signal       EQU SLOFF*4  >A0=pcb, D0.L=sigmask to signal to A0. <D0.L=-1 ok, 0=pcb doesn't exist
Block        EQU SLOFF*3  >D0.L=signal mask, <D0.L=signals received
FreeSignal   EQU SLOFF*2  >D0.B=the number of the sigbit to clear
AllocSignal  EQU SLOFF*1 < D0.B=the number of the allocated sigbit, <D0.L=-1=no free signal
-
***************** EnvVars *****************
-
EM_SWAP      EQU 0 -> Ram and ROM are swapped
EM_IRQINH    EQU 1 -> inhibit all interrupts
EM_LFCINH    EQU 2 -> inhibit LF-clock interrupts
-
ES_SWAP      EQU 0 -> swapped ROM<->RAM
ES_SER       EQU 1 -> serial is attached
ES_LFLED     EQU 2 -> LF-clock interrupt outputs and increments ENV_LED->LED
ES_KERNEL    EQU 3 -> ENV_KERNEL is initiated.
-
ENV_MCR      EQU 0    *Copy of Master Control Register: EM_*
ENV_LFC      EQU 1    *Copy of LF-clock register
ENV_LED      EQU 2    *Copy of LED register (not always used)
ENV_INT7     EQU 3    *<>0 - int7 got data for int1 to process, =0 - no data for int1
ENV_STATUS   EQU 4    *Status: ES_*
ENV_RAMLOCK  EQU 5    *Semaphore; someone is in the RAM-handler
ENV_RXLOCK   EQU 6    *Semaphore; someone is in an Rx function
ENV_TXLOCK   EQU 7    *Semaphore; someone is in a Tx function
ENV_PAD      EQU 4*2  *pad
ENV_FREEZ    EQU 5*2  *Freelist max size (num alloc (=bytes/12), 65535 max)
ENV_FREEL    EQU 3*4  *Freelist base address; -1=no memhandler
ENV_SERIPTR  EQU 4*4  *Ptr to serial input FIFO
ENV_SERIBEG  EQU 5*4  *Serial Rx nexttoread in FIFO
ENV_SERIEND  EQU 6*4  *Serial Rx nextfree in FIFO
ENV_SEROPTR  EQU 7*4  *Ptr to serial output FIFO
ENV_SEROBEG  EQU 8*4  *Serial Tx nexttosend in FIFO
ENV_SEROEND  EQU 9*4  *Serial Tx lastin in FIFO
ENV_ID       EQU 10*4 *ID-value used to see if envvars has been copied to RAM before
ENV_SRBINT1  EQU 11*4 *Int1 subroutine ptr to be executed before the original int-code (Preserved all regs!)
ENV_SRAINT1  EQU 12*4 *Int1 subroutine ptr to be executed after the original int-code
ENV_SRBINT2  EQU 13*4 *Int2      -     ||     -
ENV_SRAINT2  EQU 14*4
ENV_SRBINT3  EQU 15*4 *This will execute before the scheduler!
ENV_SRAINT3  EQU 16*4 *Used by the timer
ENV_SRBINT4  EQU 17*4 *Int4      -     ||     -
ENV_SRAINT4  EQU 18*4
ENV_SRBINT5  EQU 19*4 *Int5      -     ||     -
ENV_SRAINT5  EQU 20*4
ENV_SRBINT6  EQU 21*4 *Int6      -     ||     -
ENV_SRAINT6  EQU 22*4
ENV_SRBINT7  EQU 23*4 *Int7      -     ||     -
ENV_SRAINT7  EQU 24*4
ENV_SRARX    EQU 25*4 *Int4 Serial. Sub ptr to execute after Rx.
ENV_SRATX    EQU 26*4 *Int4 Serial. Sub ptr to execute after Tx.
ENV_SRALINE  EQU 27*4 *Int4 Serial. Sub ptr to execute after Line.
ENV_SRAMODEM EQU 28*4 *Int4 Serial. Sub ptr to execute after Modem.
ENV_SREXCEPT EQU 29*4 *Exceptions subroutine. On the stack a WORD with the vector number will be.
ENV_TIMER    EQU 30*4 *Timerbase
ENV_CONNECT  EQU 31*4 *Connect base
ENV_PCB      EQU 32*4 *Process Control Block base
ENV_KERNEL   EQU 33*4 *Kernel pointer
ENV_CRBUCKET EQU 34*4 *Place where a supervisor crash can escape
-
***************** Serial *****************
-
FIFO_SIZE EQU 1024 * needs to be 2^n
FIFO_AND  EQU 1023 * needs to be 2^n-1
-
*Note, all odd addr multiples of 16 of SER_BASE are valid up to SER_BASE+$FFFFF.
*All even addr up to SER_BASE+$FFFFE are valid for SER_LED.
SER_BASE EQU $500000    -> Base addr for the serial circuit depending on CPU-slot.
SER_LED EQU SER_BASE+0  ->B  >External LED and control register
SER_RHR EQU SER_BASE+1  ->B < Receive Holding Register
SER_THR EQU SER_BASE+1  ->B  >Transmit Holding Register
SER_DLL EQU SER_BASE+1  ->B <>Divisor Latch LSB (LCR b7=1)
SER_IER EQU SER_BASE+3  ->B <>Interrupt Enable Register
*3=modem stat, 2=rx line, 1=tx holding, 0=rx holding
-
SER_DLM EQU SER_BASE+3  ->B <>Divisor Latch MSB (LCR b7=1)
SER_FCR EQU SER_BASE+5  ->B  >FIFO Control Register
*7,6=rx trig lev, 3=DMA mode, 2=tx FIFO reset, 1=rx FIFO reset, 0=FIFO en
-
SER_ISR EQU SER_BASE+5  ->B < Interrupt Status Register
*7,6=FIFOs en, 3,2,1=irq pri, 0=irq stat
-
SER_LCR EQU SER_BASE+7  ->B <>Line Control Register
*7=DLL,DLM, 6=break, 5=parity, 4=even par, 3=par en, 2=stop bits, 1,0=word len
-
SER_MCR EQU SER_BASE+9  ->B <>Modem Control Register
*4=loopback, 3=/OP2, 2=/OP1, 1=/RTS, 0=/DTR
-
SER_LSR EQU SER_BASE+$B ->B < Line Status Register
*7=FIFO err, 6=all tx empty, 5=tx hold empty, 4=break irq, 3=frame err, 2=par err, 1=overrun, 0=rx ready
```

```
-
SER_MSR EQU SER_BASE+$D ->B < Modem Status Register
*7=CD, 6=RI, 5=DSR, 4=CTS, 3,2,1,0=delta /CD/RI/DSR/CTS
-
SER_SPR EQU SER_BASE+$F ->B <>Scratchpad Register, keeps a copy of SER_LED
-
* Speeds for the SER_DLL/M
SER_16X2400   EQU 400 ->2400   @15.360MHz
SER_16X4800   EQU 200 ->4800   @15.360MHz
SER_16X9600   EQU 100 ->9600   @15.360MHz
SER_16X19200  EQU 50  ->19200  @15.360MHz
SER_16X38400  EQU 25  ->38400  @15.360MHz
SER_16X57600  EQU 17  ->57600  @15.360MHz (exactly 16.666...)
SER_16X115200 EQU 8   ->115200 @15.360MHz (exactly 8.3333...)
-
* Serial bits
SB_LCR_8N1       EQU %00000011
SB_LCR_8ODD1     EQU %00001011
SB_LCR_8EVEN1    EQU %00011011
SB_LCR_DL        EQU %10000000
SB_FCR_RXTRIG_1  EQU %00000001
SB_FCR_RXTRIG_4  EQU %01000001
SB_FCR_RXTRIG_8  EQU %10000001
SB_FCR_RXTRIG_14 EQU %11000001
SB_FCR_FIFORESET EQU %00000111
SB_FCR_RXRESET   EQU %00000011
SB_FCR_TXRESET   EQU %00000101
-
* GetSerInfo tags
SI_SPEED      EQU 1  *.W Speed in divisor values
SI_LINESTATUS EQU 2  *.B Copy of line status register
SI_MODSTATUS  EQU 3  *.B Copy of modem status register
SI_FIFOSIZE   EQU 4  *.L The size of the FIFOs
SI_RXFIFOLOC  EQU 5  *.L Address where the FIFOs are
SI_TXFIFOLOC  EQU 6  *.L Address where the FIFOs are
SI_RXFILLLEV  EQU 7  *.L #of bytes currently in the Rx FIFO
SI_TXFILLLEV  EQU 8  *.L #of bytes currently in the Tx FIFO
SI_RXTOTAL    EQU 9  *.L Total #of bytes transferred to the Rx FIFO
SI_TXTOTAL    EQU 10 *.L Total #of bytes transferred to the Tx FIFO
-
***************** Memory *****************
-
MEM_DEFNUMALLOC EQU 1000
MEM_DEFSIZE     EQU MEM_DEFNUMALLOC*12
FR_LST_E_SIZE   EQU 3*4
-
* MemInfo parameters
MI_TOTALFREE    EQU     1
MI_LARGEST      EQU     2
MI_NUMALLOC     EQU     3
MI_NUMFREE      EQU     4
-
***************** Timer *****************
-
MAX_EL_TA       EQU 8
TIMERELSIZE     EQU 6
TDIVISOR        EQU 52
TOFFSET         EQU 4
DIVOFFSET       EQU 2
-
* Timer offsets
TO_USERDATA     EQU 0
TO_STATUS       EQU 1
TO_STARTVALUE   EQU 2
TO_COUNTER      EQU 4
-
* Timer status values
TSV_UNREG       EQU 0
TSV_REG         EQU 1
TSV_OFF         EQU 0
TSV_ON          EQU 1
TSV_DOWN        EQU 0
TSV_UP          EQU 1
TSV_NOREP       EQU 0
TSV_REP         EQU 1
-
* Timer status bits
TSB_UNREGREG    EQU 7
TSB_OFFON       EQU 6
TSB_DOWNUP      EQU 5
TSB_REPNOREP    EQU 4
-
* Timer commands
TC_RESET EQU 0*4 >D0.W=TimerNr
TC_START EQU 1*4 >D0.W=TimerNr
TC_STOP  EQU 2*4 >D0.W=TimerNr
TC_READ  EQU 3*4 >D0.W=TimerNr, <D0.L=Current timer value
TC_SET   EQU 4*4 >D0.W=TimerNr, >D1.B=Bitmask:0/1 7:UnReg/Reg, 6:Off/On 5:Down/Up, 4:Rep/NoRep, >D2.L=Startval
TC_REG   EQU 5*4 <D0.L=Offset to registred timer, No more timers=-1 (L)
TC_UNREG EQU 6*4 >D0.W=TimerNr
TC_INIT  EQU 7*4
-
***************** Connect *****************
-
* Devices
DEV_TIMER   EQU 0
-
* Types
CON_HANDL   EQU 0
CON_SIG     EQU 1
-
* Offsets
CONOFFNEXT  EQU 0
CONOFFDEV   EQU 4
CONOFFUNIT  EQU 6
```

```
CONOFFTYPE  EQU 5
CONOFFUADR  EQU 8
CONOFFIADR  EQU 12
-
CONBLKSIZE  EQU 16
-
***************** StoreProg *****************
-
LL_EL_SIZE EQU 8
-
***************** Kernel *****************
-
SYSLIST        EQU 4 -> addr to the ptr of syslist
-
KS_INHIBIT     EQU 0 -> inhibit rescheduling on int3 (as block is currently rescheduling)
KS_NOTODO      EQU 1 -> do not reschedule because there is no READY process and some rescheduler is in a STOP
KS_SIGNAL      EQU 2 -> set this to wake up from NOTODO in STOP
KS_FORBID      EQU 3 -> a userprogram requests to be alone. This gets cancelled if block: is called.
KS_SERINTSIGTX EQU 4 -> Serial called block with SIG_SERIAL_TX and expects int4tx to signal.
KS_SERINTSIGRX EQU 5 -> Serial called block with SIG_SERIAL_RX and expects int4rx to signal.
-
KRN_CURPCB    EQU 0  *.L ptr to currently running PCB
KRN_SSPREF    EQU 4  *.L the highest point in the supervisor stack.
KRN_TOPPC     EQU 8  *.L temp storage for the PC at SSPREF
KRN_TOPSR     EQU 12 *.W temp storage for the PC at SSPREF
KRN_STATUS    EQU 14 *.B KS_*
KRN_QCNT      EQU 15 *.B counter from KRN_QUANTUM to 0. When it reaches 0 a rescheduling will occur.
KRN_QUANTUM   EQU 16 *.B how many interrupt that should pass before a rescheduling should occur.
KERNELSIZE    EQU 17 *Always last element, telling the size (for allocation)
-
DEFQUANTUM    EQU 10 * Default number of LFC-ticks between rescheduling
-
PT_CODE       EQU 1
PT_DATA       EQU 2
PT_SCRIPT     EQU 3
-
PS_NEW        EQU 0
PS_SUSPENDED EQU 1
PS_FINISHED  EQU 2
PS_READY      EQU 4
PS_RUNNING    EQU 5
PS_WAITING    EQU 6
PS_CRASHED    EQU $80
-
* Invariants:
* -Ready:
*Process currently not executing on the CPU but wants to.
*KRN_CURPCB can be a ready process while rescheduling is in progress
* which means KS_INHIBIT is set.
*KS_NOTODO can not be set while a ready process exists.
* -Running:
*A running process is currently using the CPU.
*A running process can recieve a signal which means the signal came from an interrupt.
* -Waiting:
*A process that is not running and does not want to.
*KRN_CURPCB can be a waiting process while rescheduling is in progress
* which means KS_INHIBIT is set.
*KS_NOTODO is set when all processes are waiting.
*PCB_SIGWAIT can only have bits set in this state.
*
* The pcb linked list is always valid.
* If PCB_TYPE<>PT_CODE then PCB_STATUS<>PS_READY|PS_RUNNING|PS_WAITING
-
PCB_NEXT      EQU 0   *.L s ptr to next PCB
PCB_DATA      EQU 4   *.L s ptr to the code or data
PCB_SIZE      EQU 8   *.L s the size in bytes of PCB_DATA
PCB_TYPE      EQU 12  *.B s is this executable PT_CODE or non-exe PT_DATA
PCB_STATUS    EQU 13  *.B d if PT_CODE then PS_?
PCB_SR        EQU 14  *.W d current status register
PCB_PC        EQU 16  *.L d current program counter
PCB_REGS      EQU 20  *.Lx15 d all 15 registers, except A7
PCB_SP        EQU 80  *.L d current stack pointer
PCB_STACK     EQU 84  *.L s ptr to the actual stack (low end pointer)
PCB_STSIZE    EQU 88  *.L s size of the stack
PCB_SIGMASK   EQU 92  *.L d allocated signalbits
PCB_SIGWAIT   EQU 96  *.L d Block() was called with this signal mask. Only valid while STATUS=WAITING
PCB_SIGSET    EQU 100 *.L d Signals that have arrived but not yet read by block.
PCB_NAME      EQU 104 *.L s ptr to 0-string.
PCBSIZE       EQU 108 *Always last element, telling the size (for allocation)
-
* Reserved global signals. All processes waiting for such
* a signal gets one when the corresponding event occurs.
SIG_CTRL_C    EQU 31
SIG_SERIAL_RX EQU 30 -> KRN_STATUS must bset KS_SERINTSIGRX for this to work.
SIG_SERIAL_TX EQU 29 -> KRN_STATUS must bset KS_SERINTSIGTX for this to work.
SIG_KEYBOARD  EQU 28
SIG_DISKREAD  EQU 27
SIG_DISKWRITE EQU 26
SIG_RESERVED1 EQU 25
SIG_RESERVED2 EQU 24
-
USERSIGNALS   EQU $00FFFFFF
```

F.2 MemLayout.s68

This is the main file in the project. It contains the interrupt vector table, the startup code, the primitive console, the environment variables and various initialization routines. This is the file that includes all other files that are also burned into the EPROM. The name ”MemLayout” got stuck as this file was originally just the interrupt vector table plus comments about the general memory layout in dBOX.

All label:'s, meaning functions, subroutines, variables, etc., are global in the entire project. All files included in one way or another will have access to all label:'s in all files. The development environment Eterm which was used is made this way. This creates a naming problem which was solved by using the initials of a subroutine name (label:) as a preamble for all branch targets within this subroutine. For example: The subroutine ”initMemory:” has all its labels named ”imLabelName:”. The naming of labels and constants follows the C++/Java standard with labels beginning with a lowercase letter, then capitalized first letter in all sub-words. Constants are all in uppercase with underscore between words as seen in Definitions.s68.

```
 USE definitions.s68
-
* ROM
* ORG 0
-
START EQU start
-
 DC.L SYSSP     *SSP
 DC.L start     *ResetVector and syslist base pointer
 DC.L BusErr    *BUS
 DC.L AddrErr   *ADDR
 DC.L Illegal   *ILLEGAL
 DC.L DivBy0    *DIV/0
 DC.L CHK       *CHK
 DC.L TrapV     *TRAPV
 DC.L PrivViol  *Priv Viol
 DC.L Trace     *Trace
 DC.L LineA     *Line-A (MMU)
 DC.L LineF     *Line-F (FPU)
-
 DC.L res,res,res *Reserved
 DC.L spur        *Spurious
 DC.L res,res,res,res,res,res,res,res *Reserved
 DC.L spur        *Spurious2?
-
 DC.L int1  *Auto1
 DC.L int2  *Auto2
 DC.L int3  *Auto3
 DC.L int4  *Auto4
 DC.L int5  *Auto5
 DC.L int6  *Auto6
 DC.L int7  *Auto7
-
 DC.L tAllocMem   *TRAP 0
 DC.L tFreeMem    *TRAP 1
 DC.L tTimer      *TRAP 2
 DC.L tReSchedule *TRAP 3
 DC.L tStop       *TRAP 4
 DC.L tSwapROMRAM *TRAP 5
 DC.L tDebug      *TRAP 6
 DC.L tSuperVisor *TRAP 7
 DC.L tUserTrap   *TRAP 8
 DC.L trap,trap,trap,trap,trap,trap *TRAP 9-14
 DC.L tReset      *TRAP 15
-
 DC.L res,res,res,res,res,res,res,res,res,res,res,res,res,res,res,res *Reserved
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui *UserInterrupt
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui *Unused
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
 DC.L ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui,ui
-
* D7 = LED increment register during startup
* ORG $400
 LEA (lowAddrErr,PC),A0
 MOVE.L (A0)+,D0
 BSR sendS
lowAddrLoop:
 BRA.S lowAddrLoop
lowAddrErr:
 DC.L 40+45+14
 DC "Instruction execution at low addresses!\n"
 DC "Please check the software that is being executed.\n\n"
 DC "Press reset to continue.\n\n"
 ALIGN
 JMP (delay).L
 JMP (event).L
 JMP (connect).L
 JMP (initSerial).L
 JMP (setSerSpeed).L
 JMP (getSerInfo).L
 JMP (sendS).L
 JMP (sendA).L
 JMP (getS).L
 JMP (getA).L
 JMP (putS).L
 JMP (readS).L
 JMP (flushTx).L
 JMP (flushRx).L
 JMP (putStr).L
 JMP (strCmp).L
 JMP (strCmpNC).L
 JMP (strLen).L
 JMP (strCopy).L
 JMP (str2Int).L
 JMP (int2Dec).L
 JMP (int2Hex).L
 JMP (compFreeList).L
 JMP (memInfo).L
 JMP (memCheck).L
 JMP (storeProg).L
 JMP (getEnvToA6).L
 JMP (incLed).L
 JMP (singleTask).L
 JMP (signal).L
 JMP (block).L
 JMP (freeSignal).L
 JMP (allocSignal).L
start:
 MOVE.B  #1,LED      -> Yes, it works!!
 MOVE.B  #2,LED      -> Did the I/O DTACK work?
 MOVE.B  envvars,MCR -> Set CPU-clock to 2.5MHz etc if coldstart in ROM. Nothing will(should) happen in a hot reboot as envva
 MOVE.B  #3,LED      -> Still alive after MCR?
 MOVEA.L #0,A6
 TST.B   DIP
 BNE.W   mainContinue
mainTestDIP:
  MOVE.B DIP,D0
   BTST.B #7,D0
  BEQ.W  mainTestDIP
  ANDI.B #$7f,D0
  BNE.S  mainNot0
  BRA.W  mainWaitDIP
mainNot0:
  CMPI.B #1,D0        ->1 Continue
  BEQ.W  mainContinue
  CMPI.B #2,D0        ->2 RAM-less test (1=no RAM, 2=RAM at $100k, 3=RAM at $0)
  BNE.S mainNot2
   MOVE.L  ROM,D0
   MOVE.L  D0,D1
   ADDI.L  #$55AA55AA,D0
   MOVE.L  D0,ROM
   CMP.L   ROM,D0
   BEQ.S   mainHotRAM
    MOVE.L  RAM,D0
    MOVE.L  D0,D1
    ADDI.L  #$55AA55AA,D0
    MOVE.L  D0,RAM
    CMP.L   RAM,D0
    BEQ.S   mainHasRAM
     MOVE.B #1,LED
     BRA.S  mainNoRAM
mainHasRAM:
    MOVE.L  D1,RAM
    MOVE.B  #2,LED
    BRA.S   mainNoRAM
mainHotRAM:
   MOVE.L  D1,ROM
   MOVE.B  #3,LED
mainNoRAM:
  BRA.W  mainWaitDIP
mainNot2:
  CMPI.B #3,D0        ->3 testROMRAM (ENV_ST->LED) [1]
  BNE.S  mainNot3
   BSR.W  testROMRAM
   MOVE.B (ENV_STATUS,A6),LED
  BRA.W  mainWaitDIP
mainNot3:
  CMPI.B #4,D0
  BNE.S  mainNot4     ->4 incLed [1]
   BSR.W incLed
  BRA.W  mainWaitDIP
mainNot4:
  CMPI.B #5,D0        ->5 STOP int0 [1]
  BNE.S  mainNot5
   STOP   #$2000
   MOVE.B #5,LED
  BRA.W  mainWaitDIP
mainNot5:
  CMPI.B #6,D0
  BNE.S  mainNot6     ->6 STOP int7 [1]
   STOP   #$2700
   MOVE.B #6,LED
  BRA.W  mainWaitDIP
mainNot6:
  CMPI.B #7,D0
  BNE.S  mainNot7     ->7 noRAM (no return from this) [1]
   BRA.W noRAM
  BRA.W  mainWaitDIP
mainNot7:
  CMPI.B #8,D0
  BNE.S  mainNot8     ->8 initSerial (1=no serial, 2=serial) [3]
   BSR    initSerial
   BTST.B #ES_SER,(ENV_STATUS,A6)
   BEQ.S  mainNoSer
    MOVE.B #2,LED
    BRA.S  mainWasSer
mainNoSer:
    MOVE.B #1,LED
mainWasSer:
  BRA.W  mainWaitDIP
mainNot8:
  CMPI.B #9,D0
  BNE.S  mainNot9     ->9 Write to MCR. Set DIP then release bit 3,
*                     ->  which is unused in MCR, when done. [1]
mainWait9:
   BTST.B #3,DIP
   BNE.S  mainWait9
   MOVE.B DIP,D0
   MOVE.B D0,(ENV_MCR,A6)
   MOVE.B D0,MCR
   MOVE.B #9,LED
  BRA.S  mainWaitDIP
mainNot9:
  CMPI.B #$A,D0
  BNE.S  mainNotA     ->A console [4]
   BSR.W  console
   MOVE.B #$A,LED
  BRA.S  mainWaitDIP
mainNotA:
  CMPI.B #$C,D0
  BNE.S  mainNotC     ->C getEnvToA6 [2]
   BSR.W  getEnvToA6
   MOVE.B #$C,LED
  BRA.S  mainWaitDIP
mainNotC:
  CMPI.B #$E,D0
  BNE.S  mainNotE     ->E Send welcome on serial [4]
   BSR.W  welcomeSerial
   MOVE.B #$E,LED
  BRA.S  mainWaitDIP
mainNotE:
 NOP
mainWaitDIP:
   BTST.B #7,DIP
  BNE.S  mainWaitDIP
  MOVE.B #0,LED
 BRA.W mainTestDIP
mainContinue:
 MOVEQ   #3,D7
 BSR.S   incLed      -> 4 BSR work..
 BSR.S   incLed      -> 5 RTS and the stack works..
 BSR.S   testROMRAM
 BSR.S   incLed      -> 6 Swap-test complete
 BSR     initSerial  -> If serial is present bit 5 in LED will be set
 BSR.S   incLed      -> 7 Serial init complete
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BNE.S   mainSerIsInited
mainStop:
  STOP   #$2000
 BRA.S   mainStop
mainSerIsInited:
 BSR.S   incLed      -> 8
* BSR     initMemory
* BSR.S   incLed      ->
* BSR     initKernel
* BSR.S   incLed      ->
 MOVE    #$2000,SR
 BSR.S   incLed      -> 9
 BSR     welcomeSerial
 BSR.S   incLed      -> A
 TST.L   (ENV_FREEL,A6)
 BEQ.S   mainConsole
  BSR     terminal
mainConsole:
 BRA     console
-
-
incLed:
 CMPA.L #0,A6
 BEQ.S  ilD7
  ADDQ.B #1,(ENV_LED,A6)
  MOVE.B (ENV_LED,A6),LED
  RTS
ilD7:
 ADDQ.B #1,D7
 MOVE.B D7,LED
 RTS
-
-
testROMRAM:
 MOVE.L  ROM,D0
 MOVE.L  D0,D1
 ADDI.L  #$55AA55AA,D0
 MOVE.L  D0,ROM
 CMP.L   ROM,D0
 BEQ.S   trrHot -> Is RAM at addr 0? <=> are we swapped?
  MOVEA.L #envvars+RAM,A6
  MOVEA.L #envvars,A0
  MOVEA.L A6,A1
  MOVE.W  #envend-envvars-1,D0
  CMPI.L  #$DEADBEEF,(ENV_ID,A6) -> Did we do a warm reboot (not cold, but in ROM)?
  BNE.S   trrCold
   SUBQ.W #8,D0
   ADDQ.L #8,A1
   ADDQ.L #8,A0
trrCold:
   LSR.W  #2,D0
trrCopyEnv:
    MOVE.L (A0)+,(A1)+
   DBRA   D0,trrCopyEnv
*trrNoCopy:
  BRA.S  trrNotHot
trrHot:
 MOVE.L  D1,ROM
 TST.B   DIP
 BNE.S   trrNormal
  MOVE.L  #0,(ENV_ID,A6) -> force a fresh copy of MCR
  BCLR.B  #EM_SWAP,envvars
  MOVEA.L #SYSSP,A7
  JMP     start
trrNormal:
 MOVEA.L #envvars,A6
 MOVEA.L #envvars+RAM+ENV_SRARX,A0 -> this is ROM now
 MOVEA.L A6,A1
 ADDA.L  #ENV_SRARX,A1
 MOVEQ   #ENV_CRBUCKET-ENV_SRARX,D0
 LSR.L   #2,D0
trrCopyLastEnv:
  MOVE.L (A0)+,(A1)+
 DBRA    D0,trrCopyLastEnv
 BSET.B  #ES_SWAP,(ENV_STATUS,A6) *We're in RAM
trrNotHot:
 BSET.B  #EM_LFCINH,(ENV_MCR,A6)
 MOVE.B  (ENV_MCR,A6),MCR
 BSET.B  #6,D7
 MOVE.B  D7,(ENV_LED,A6)
_RTS
-
getEnvToA6:
 BTST.B #ES_SWAP,envvars+ENV_STATUS
 BNE.S  gea6NoSwap
  MOVEA.L #envvars+RAM,A6
  RTS
gea6NoSwap:
 MOVEA.L #envvars,A6
_RTS
-
tswapToRam:
 DC.L 59+58+54
 DC     "A swap of RAM and ROM will now be made so that RAM will be "
 DC     "at address 0. A copy of the startup code from ROM will be "
 DC     "made that will overwrite the first few kB of the RAM.\n"
 ALIGN
tswapBack:
 DC.L 54+53+26
 DC     "RAM and ROM will now be swapped back so that ROM will "
 DC     "be at address 0 and a reset will be made. The memory "
 DC     "manager will be disabled.\n"
 ALIGN
tswapDone:
 DC.L 30
 DC     "The swap has now taken place.\n"
_ALIGN
-
tSwapROMRAM:
 BSR     getEnvToA6
 BTST.B  #ES_SWAP,(ENV_STATUS,A6)
 BEQ.S   tswapInRom:
  LEA     (tswapBack,PC),A0
  MOVE.L  (A0)+,D0
  BSR     sendS
  MOVEA.L #RAM+start,A0
  MOVEA.L #ROM+start,A1
  MOVE.L  #mainTestDIP-1,D0
  LSR.L   #2,D0
tswapACopy:
   MOVE.L (A0)+,(A1)+
  DBRA    D0,tswapACopy
  CLR.L   (ENV_FREEL,A6)
  BCLR.B  #EM_SWAP,(ENV_MCR,A6)
  MOVEA.L #SYSSP,A7
  JMP     start
  BRA     tswapOut
tswapInRom:
  LEA     (tswapToRam,PC),A0
  MOVE.L  (A0)+,D0
  BSR     sendS
  MOVEA.L #ROM,A0
  MOVEA.L #RAM,A1
  MOVE.L  #envvars-1,D0
  LSR.L   #2,D0
tswap0Copy:
   MOVE.L (A0)+,(A1)+
  DBRA   D0,tswap0Copy
  MOVE   SR,D7
  MOVE   #$2700,SR
  TST.L  (ENV_FREEL,A6)
  BEQ.S  tswapRomInitMem
*  ->Here the noswap memmgr is inited
   MOVEA.L (ENV_FREEL,A6),A0
   MOVE.W  (A0)+,D0 -> current size of memmgr
tswapOMFixLoop:
    MOVE.L  (A0),D1
    ANDI.L  #$FFF00000,D1
    CMPI.L  #$00100000,D1
    BNE.S   tswapOMNoFix1
     ANDI.L #$000FFFFF,(A0)
tswapOMNoFix1:
    ADDQ.L  #4,A0
    MOVE.L  (A0),D1
    ANDI.L  #$FFF00000,D1
    CMPI.L  #$00100000,D1
    BNE.S   tswapOMNoFix2
     ANDI.L #$000FFFFF,(A0)
tswapOMNoFix2:
    ADDQ.L  #8,A0
   DBRA   D0,tswapOMFixLoop
   MOVE.W (ENV_FREEL,A6),D0
   ANDI.W #$FFE0,D0
   BNE.S  tswapOMNoFixFL
    BCLR.B #4,(ENV_FREEL+1,A6)
tswapOMNoFixFL:
   BRA.S tswapRomCommon
tswapRomInitMem:
* ->Here no memmgr is inited
   MOVEA.L #initialFreeListSwap,A0
   MOVEA.L #RAM+endOfStartUpImage,A1
   MOVE.W  (A0)+,D0
tswapMCopy:
    MOVE.L (A0)+,(A1)+
    MOVE.L (A0)+,(A1)+
    MOVE.L (A0)+,(A1)+
   DBRA   D0,tswapMCopy
   MOVE.L #endOfStartUpImage,(ENV_FREEL,A6)
   MOVE.W #MEM_DEFNUMALLOC,(ENV_FREEZ,A6)
tswapRomCommon:
  MOVE.W (2,A7),D0   -> Fix the callers return address
  ANDI.W #$FFE0,D0
  BNE.S  tswapOMNoFixSt
   BCHG.B #4,(3,A7)
tswapOMNoFixSt:
  MOVE    USP,A0     -> Fix the USP
  MOVE.L  A0,D0
  ANDI.L  #$FFF00000,D0
  CMPI.L  #$00100000,D0
  BNE.S   tswapOMNoFixUSP
   ANDI.L  #$000FFFFF,D0
   MOVEA.L D0,A0
   MOVE    A0,USP
tswapOMNoFixUSP:
  MOVE.L  A7,D0      -> Fix the SSP
  ANDI.L  #$FFF00000,D0
  CMPI.L  #$00100000,D0
  BNE.S   tswapOMNoFixSSP
   ANDI.L  #$000FFFFF,D0
   MOVEA.L D0,A7
tswapOMNoFixSSP:
-
*  -> Fix all programs final return address in ENV_PCB!
-
  BSET.B #EM_SWAP,(ENV_MCR,A6)
  MOVE.B (ENV_MCR,A6),MCR       -> DO IT!
  MOVE   D7,SR
  LEA    (tswapDone,PC),A0
  MOVE.L (A0)+,D0
  BSR    sendS
tswapOut:
 RTS
-
-
* envvars must be an even amount of LONGs
envvars:
 DC.B $24   *MCR (inhibit LF-clock and set 2.5MHz)
 DC.B $FF   *LF-clock (reset LF-clock and set speed to slowest)
 DC.B 0     *LED
 DC.B 0     *<>0 - int7 got data for int1 to process, =0 - no data for int1
 DC.B 0     *Status: bit0=swapped, 1=serial, 2=LF connected to LED, 3=Multitasking engaged
 DC.B 0     *Semaphore; someone is in the RAM-handler
 DC.B 0     *Semaphore; someone is in an Rx function
 DC.B 0     *Semaphore; someone is in a Tx function
 DC.W 0     *pad
 DC.W MEM_DEFNUMALLOC *Freelist max size (num alloc (=bytes/12), 65535 max)
 DC.L 0     *Freelist base address
 DC.L RAM+serInpFifo *Ptr to serial input FIFO
 DC.L 0,0   *Serial Rx nexttoread,nextfree in FIFO
 DC.L RAM+serOutFifo *Ptr to serial output FIFO
 DC.L 0,0   *Serial Tx nexttosend,lastin in FIFO
 DC.L $DEADBEEF *ID-value used to see if envvars has been copied to RAM before
 DC.L 0,0   *Int1 subroutine ptr to be executed before,after the original int-code
 DC.L 0,0   *Int2      -     ||     -
 DC.L 0,0   *Int3      -     ||     -
 DC.L 0,0   *Int4      -     ||     -
 DC.L 0,0   *Int5      -     ||     -
 DC.L 0,0   *Int6      -     ||     -
 DC.L 0,0   *Int7      -     ||     -
 DC.L 0     *Int4 Serial. Sub ptr to execute after Rx.
 DC.L 0     *Int4 Serial. Sub ptr to execute after Tx.
 DC.L 0     *Int4 Serial. Sub ptr to execute after Line.
 DC.L 0     *Int4 Serial. Sub ptr to execute after Modem.
 DC.L 0     *Exceptions subroutine. On the stack a WORD with the vector number will be.
 DC.L 0     *Timer base
 DC.L 0     *Connect base
 DC.L 0     *Process Control Block base
 DC.L 0     *Kernel base
 DC.L 0     *Crash bucket
serInpFifo:
 DS.B FIFO_SIZE
serOutFifo:
 DS.B FIFO_SIZE
envend:
-
endOfStartUpImage:
-
initMemory:
 BSR     getEnvToA6
 BTST.B  #ES_SWAP,(ENV_STATUS,A6)
 BEQ.S   imNoSwap
  MOVEA.L #initialFreeListSwap,A0
  MOVEA.L #endOfStartUpImage,A1
  MOVE.L  A1,(ENV_FREEL,A6)
  BRA.S   imWasSwap
imNoSwap:
  MOVEA.L #initialFreeListNoSwap,A0
  MOVEA.L #RAM+endOfStartUpImage,A1
  MOVE.L  A1,(ENV_FREEL,A6)
imWasSwap:
 MOVE.W  (A0)+,D0
imCopy:
  MOVE.L (A0)+,(A1)+
  MOVE.L (A0)+,(A1)+
  MOVE.L (A0)+,(A1)+
 DBRA   D0,imCopy
 MOVE.W #MEM_DEFNUMALLOC,(ENV_FREEZ,A6)
 RTS
-
-
dummyProc:
 BRA.S dummyProc
 RTS
dummyProcEnd:
-
ikNoMemory:
 DC.L 29
 DC "Memory manager not initiated\n"
 ALIGN
ikOutOfMem:
 DC.L 15
 DC "Out of memory!\n"
 ALIGN
-
initKernel:
 BSR   getEnvToA6
 TST.L (ENV_FREEL,A6)
 BEQ   ikNoMem
  MOVEQ #KERNELSIZE,D0
  TRAP  #allocmem
  CMPA.L #0,A0
  BEQ    ikOutOfMem1
   MOVE.L  A0,(ENV_KERNEL,A6)
   MOVEA.L A0,A1
   MOVE.L  #PCBSIZE+6+100,D0 *+"Dummy\0"+stack
   TRAP    #allocmem
   CMPA.L  #0,A0
   BEQ.S   ikOutOfMem2
    MOVE.L A0,(ENV_PCB,A6)
    MOVE.L #RAM+MEMSIZE-SSPSIZE,(KRN_SSPREF,A1)
    MOVE.B #0,(KRN_STATUS,A1)
    MOVE.B #DEFQUANTUM,(KRN_QCNT,A1)
    MOVE.B #DEFQUANTUM,(KRN_QUANTUM,A1)
    MOVE.L A1,(ENV_KERNEL,A6)
    MOVE.L #NIL,(PCB_NEXT,A0)
    LEA    (dummyProc,PC),A2
    MOVE.L A2,(PCB_DATA,A0)
    MOVE.L #dummyProcEnd-dummyProc,(PCB_SIZE,A0)
    MOVE.B #PT_CODE,(PCB_TYPE,A0)
    MOVE.B #PS_NEW,(PCB_STATUS,A0)
    LEA    (PCBSIZE+100,A0),A2
    MOVE.L A2,(PCB_SP,A0)
    MOVE.L #100-6,(PCB_STSIZE,A0)
    LEA    (PCBSIZE,A0),A2
    MOVE.L A2,(PCB_NAME,A0)
    MOVE.L #$44756D6D,(A2)+ *'Dumm'
    MOVE.W #$7900,(A2)+    *'y\0'
    MOVE.L A2,(PCB_STACK,A0)
    MOVE.L A0,(KRN_CURPCB,A1)
   RTS
ikOutOfMem2:
  MOVEA.L (ENV_KERNEL,A6),A0
  TRAP    #freemem
  MOVE.L  #NIL,(ENV_KERNEL,A6)
ikOutOfMem1:
  LEA    (ikOutOfMem,PC),A0
  MOVE.L (A0)+,D0
  BSR    sendS
 BRA.S ikOut
ikNoMem:
 LEA    (ikNoMemory,PC),A0
 MOVE.L (A0)+,D0
 BSR    sendS
ikOut:
 RTS
-
-
* Used to test the serial without any RAM in the computer.
noRAM:
 MOVE.B #%10,SER_LED ->reset serial
 MOVE.B #0,SER_LED
 MOVE.B SER_SPR,D0   ->test if serial exists
 ADD.B  #$5A,D0
 MOVE.B D0,SER_SPR
 CMP.B  SER_SPR,D0
 BNE    nrNotAttached
  MOVE.B #%10000000,SER_LCR ->enable DLL/DLM
  MOVE.W #SER_16X9600,D0
  MOVE.B D0,SER_DLL
  LSR.W  #8,D0
  MOVE.B D0,SER_DLM
  MOVE.B #%00000011,SER_LCR ->disable DLL/DLM and set 8N1
  MOVE.B #%00000111,SER_FCR ->reset and enable FIFO
  MOVE.B #%00000000,SER_SPR ->indicate nothing on the LEDs
  MOVE.B SER_SPR,SER_LED
  MOVE.B #%00000000,SER_IER ->No IRQ enabled
nrNoDIP:
  BTST.B #0,SER_LSR
  BEQ.S  nrTestDIP
   MOVE.B SER_LSR,D0
   BTST.B #1,D0
   BEQ.S  nrNotOFL
    BSET.B #6,SER_SPR
    MOVE.B SER_SPR,SER_LED
nrNotOFL:
   ANDI.B #%10001100,D0
   BEQ.S  nrNotErr
    BSET.B #7,SER_SPR
    MOVE.B SER_SPR,SER_LED
nrNotErr:
  MOVE.B SER_RHR,LED
nrTestDIP:
  MOVE.B DIP,D0
   BTST.B #7,D0
  BEQ.S  nrNoDIP
  ANDI.B #$7f,D0
  BEQ.S  nrClrSERLED
   MOVE.B D0,SER_THR
   BRA.S  nrWaitDIP
nrClrSERLED:
   MOVE.B #0,SER_SPR
   MOVE.B #0,SER_LED
nrWaitDIP:
   BTST.B #7,DIP
  BNE.S  nrWaitDIP
  BRA.S  nrNoDIP
nrNotAttached:
  MOVE.B DIP,LED
 BRA nrNotAttached
-
-
************************ Console ***************************
cHelp:
 DC.L 60+19+67+55+41+36+80+32+32+33
 DC     "This basic console accepts only single character commands:\n\n"
 DC     "? - This help text\n"
 DC     "m - Start the memory manager so a more advanced console can be used\n"
 DC     "l - Load the following S1-data into addr $0+ in memory\n"
 DC     "e - Execute the code at addr 0 in memory\n"
 DC     "k - Initiate the kernel structures.\n"
 DC     "r - Make a software reset on the computer. DIP=0 means swap to ROM if in RAM\n"
 DC     "2 - Set serial speed to 2400bps\n"
 DC     "9 - Set serial speed to 9600bps\n"
 DC     "3 - Set serial speed to 38400bps\n\0"
 ALIGN
cStartMem:
 DC.L 57+59+27
 DC     "The memory manager will now be initiated so that you may "
 DC     "use the more advanced console and be able to load programs "
 DC     "into memory and run them.\n\n"
 ALIGN
cMemStarted:
 DC.L 37 *+64
 DC     "The memory manager has been started.\n"
* DC     "You will now be put in the advance console, aka. the terminal.\n\n"
 ALIGN
cBack:
 DC.L 43
 DC     "You are now back to the primitive console.\n"
 ALIGN
cExe:
 DC.L 63
 DC     "I will now jump to address RAM+endOfStartUpImage+MEM_DEFSIZE..\n"
 ALIGN
cExeDone:
 DC.L 63
 DC     "\nThe program has now finished and you are back in the console.\n"
 ALIGN
cKernel:
 DC.L 60
 DC     "The kernel will now be initiated and scheduling will begin.\n"
cReset:
 DC.L 37
 DC     "A soft reset will now be performed..\n"
 ALIGN
cSet2400:
 DC.L 83
 DC     "Serial speed will now be set to 2400 bps and a message will be printed afterwards.\n"
 ALIGN
c2400Set
 DC.L 37
 DC     "Serial has now been set to 2400 bps.\n"
 ALIGN
cSet9600:
 DC.L 83
 DC     "Serial speed will now be set to 9600 bps and a message will be printed afterwards.\n"
 ALIGN
c9600Set
 DC.L 37
 DC     "Serial has now been set to 9600 bps.\n"
 ALIGN
cSet38400:
 DC.L 84
 DC     "Serial speed will now be set to 38400 bps and a message will be printed afterwards.\n"
 ALIGN
c38400Set
 DC.L 38
 DC     "Serial has now been set to 38400 bps.\n"
 ALIGN
cNoCommand:
 DC.L 49
 DC     "This command is not implemented. Try ? for help.\n"
 ALIGN
cLoading:
 DC.L 7
 DC     "Loading\0"
 ALIGN
cComplete:
 DC.L 9
 DC     "complete\n\0"
 ALIGN
-
console:
 MOVEQ   #'>',D0
 BSR     putS
 MOVEQ   #0,D2
konsoll:
 BSR     getS
 CMPI.B  #127,D0
 BEQ.S   console
 CMPI.B  #8,D0    ->Backspace?
 BNE.S   cNotBS
  TST.B  D2
  BEQ.S  konsoll
  BSR    putS
  MOVEQ  #0,D2
  BRA.S  konsoll
cNotBS:
 CMPI.B #10,D0   ->Enter?
 BNE    cNotLF
  BSR    putS
  TST.B  D2
  BEQ.S  console
  CMPI.B #'?',D2 ->Help
  BNE.S  cNotHelp
   LEA    (cHelp,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BRA.S  console
cNotHelp:
  CMPI.B #'m',D2 ->Start memory manager
  BNE.S  cNotMem
   LEA     (cStartMem,PC),A0
   MOVE.L  (A0)+,D0
   BSR     sendS
   BSR     initMemory
   LEA     (cMemStarted,PC),A0
   MOVE.L  (A0)+,D0
   BSR     sendS
  BRA    console
cNotMem:
  CMPI.B #'e',D2 ->Run program at addr envend in RAM (JSR $100000+endOfStartUpImage+MEM_DEFSIZE)
  BNE.S  cNotExe
   LEA     (cExe,PC),A0
   MOVE.L  (A0)+,D0
   BSR     sendS
   CMPA.L  #RAM+MEMSIZE-SSPSIZE,A7
   BEQ.S   cInUserMode
    MOVEA.L #RAM+MEMSIZE-SSPSIZE,A0
    MOVE    A0,USP
    MOVE    #0,SR
cInUserMode:
   JSR     RAM+endOfStartUpImage+MEM_DEFSIZE
   MOVEA.L SYSLIST,A5
   LEA     (cExeDone,PC),A0
   MOVE.L  (A0)+,D0
   BSR     sendS
   BRA     console
cNotExe:
  CMPI.B #'k',D2 ->Initiate kernel
  BNE.S  cNotKernel
   LEA    (cKernel,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BSR    initKernel
   BRA    console
cNotKernel:
  CMPI.B #'r',D2 ->Make a softreset
  BNE.S  cNotReset
   LEA    (cReset,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   TRAP   #reset
   BRA    console
cNotReset:
  CMPI.B #'2',D2 ->Set serial to 2400kb
  BNE.S  cNot2400
   LEA    (cSet2400,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BSR    flushTx
   MOVE.W #SER_16X2400,D0
   BSR    setSerSpeed
   LEA    (c2400Set,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BRA    console
cNot2400:
  CMPI.B #'9',D2 ->Set serial to 9600kb
  BNE.S  cNot9600
   LEA    (cSet9600,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BSR    flushTx
   MOVE.W #SER_16X9600,D0
   BSR    setSerSpeed
   LEA    (c9600Set,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BRA    console
cNot9600:
  CMPI.B #'3',D2 ->Set serial to 38400kb
  BNE.S  cNot38400
   LEA    (cSet38400,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BSR    flushTx
   MOVE.W #SER_16X38400,D0
   BSR    setSerSpeed
   LEA    (c38400Set,PC),A0
   MOVE.L (A0)+,D0
   BSR    sendS
   BRA    console
cNot38400:
  LEA    (cNoCommand,PC),A0
  MOVE.L (A0)+,D0
  BSR    sendS
  BRA    console
cNotLF:
 TST.B  D2
 BNE    konsoll
 CMPI.B #'l',D0 ->Load following data into RAM at addr envend
 BNE.S  cNotLoad
  LEA    (cLoading,PC),A0
  MOVE.L (A0)+,D0
  BSR    sendS
  BSR    loadS1
  LEA    (cComplete,PC),A0
  MOVE.L (A0)+,D0
  BSR    sendS
  BRA   console
cNotLoad:
 CMPI.B #13,D0
 BNE.S  cNotCR
  BSR   putS
  BRA   console
cNotCR:
 CMPI.B #32,D0
 BGE.S  cNotLow
  BSR   cSendHex
  MOVEQ #10,D0
  BSR   putS
  BRA   console
cNotLow:
 BTST.B #7,D0
 BEQ.S  cNotHi
  BSR   cSendHex
  MOVEQ #13,D0
  BSR   putS
  MOVEQ #10,D0
  BSR   putS
  BRA   console
cNotHi:
 BSR    putS
 MOVE.B D0,D2
 BRA    konsoll
*RTS
-
cSendHex:
 MOVE.B D0,D1
 LSR.B  #4,D0
 CMPI.B #10,D0
 BPL.S  cshHigh1
  ADDI.B #'0',D0
  BRA.S  cshWasLow1
cshHigh1:
  ADDI.B #'A'-10,D0
cshWasLow1:
 BSR    putS
 MOVE.B D1,D0
 ANDI.B #15,D0
 CMPI.B #10,D0
 BPL.S  cshHigh2
  ADDI.B #'0',D0
  BRA.S  cshWasLow2
cshHigh2:
  ADDI.B #'A'-10,D0
cshWasLow2:
 BSR putS
 RTS
```

```
-
-
loadS1:
 MOVEA.L #RAM+endOfStartUpImage+MEM_DEFSIZE,A0
ls1Start:
 MOVE.B  #'.',D0
 BSR     putS
 BSR     ls1WhiteSp  -> S
 CMPI.B  #'S',D0
 BEQ.S   ls1WasS
  RTS
ls1WasS:
 BSR     getS        -> 1
 CMPI.B  #'1',D0
 BNE.S   ls1Not1
  BSR    ls1GetnDec  -> len
  MOVE.B D0,D2
  SUBQ.B #3,D2
  BSR    getS        -> addr
  BSR    getS
  BSR    getS
  BSR    getS
ls1Read:
   BSR    ls1GetnDec -> data
   MOVE.B D0,(A0)+
   SUBQ.B #1,D2
  BNE.S ls1Read
  BSR   getS         -> checksum
  BSR   getS
  BRA.S ls1Start
ls1Not1:
 CMPI.B  #'2',D0
 BNE.S   ls1Not2
  BSR    ls1GetnDec  -> len
  MOVE.B D0,D2
  SUBQ.B #4,D2
  BSR    getS        -> addr
  BSR    getS
  BSR    getS
  BSR    getS
  BSR    getS
  BSR    getS
ls2Read:
   BSR    ls1GetnDec -> data
   MOVE.B D0,(A0)+
   SUBQ.B #1,D2
  BNE.S ls2Read
  BSR   getS         -> checksum
  BSR   getS
  BRA   ls1Start
ls1Not2:
 MOVE.B D0,D1
 BRA.S  ls1WaitLine
ls1UntilCR:
  BSR    getS
ls1WaitLine:
  CMPI.B #10,D0
  BEQ.S  ls1Out
  CMPI.B #13,D0
 BNE.S  ls1UntilCR
ls1Out
 CMPI.B #'8',D1
 BEQ.S  ls1Finish
 CMPI.B #'9',D1
 BNE    ls1Start
ls1Finish:
 RTS
-
-
ls1GetnDec:
 BSR    getS
 CMPI.B #'A',D0
 BLT.S  gndNum1
  SUBI.B #'A'-10,D0
  BRA.S  gndWasLett1
gndNum1:
  SUBI.B #'0',D0
gndWasLett1:
 LSL.B  #4,D0
 MOVE.B D0,D1
 BSR    getS
 CMPI.B #'A',D0
 BLT.S  gndNum2
  SUBI.B #'A'-10,D0
  BRA.S  gndWasLett2
gndNum2:
  SUBI.B #'0',D0
gndWasLett2:
 OR.B D1,D0
 RTS
-
-
ls1WhiteSp:
 BSR    getS
 CMPI.B #32,D0
 BLE.S  ls1WhiteSp
 RTS
-
************************ Terminal **************************
-
tWelcome:
 DC.L 48+55
 DC     "Welcome to the advanced console - The Terminal\r\n"
 DC     "This is currently under construction and unavailable.\r\n\0"
 ALIGN
tNoMemH:
 DC.L 54+56
 DC     "The Terminal need the memory handler to be started in "
 DC     "order to operate and it is not. Please start it first!\r\n\0"
 ALIGN
-
terminal:
 BSR    getEnvToA6
 TST.L  (ENV_FREEL,A6)
 BEQ.S  tNoMH
  LEA   (tWelcome,PC),A0
  BRA.S tHasMH
tNoMH:
  LEA   (tNoMemH,PC),A0
tHasMH:
 MOVE.L  (A0)+,D0
 BSR     sendS
 RTS
-
************************ Other ******************************
 USE kernel.s68
 USE serial.s68
 USE stdlib.s68
 USE memhandling.s68
 USE interrupts.s68
 USE timer.s68
 USE progInlasning.s68
-
* Do not remove!
* Labels RTS and RTE to be able to detect RTS's and RTE's written
* accidently at the left edge in any subroutine.
RTE
RTS
-
initialFreeListNoSwap:
 DC.W 5
 DC.L RAM,RAM+START-1,0    -> vectors
 DC.L RAM+START,RAM+endOfStartUpImage-1,0    -> essential startup code
 DC.L RAM+endOfStartUpImage,RAM+endOfStartUpImage+MEM_DEFSIZE-1,0 -> the mem mgr buffer
 DC.L RAM+endOfStartUpImage+MEM_DEFSIZE,RAM+MEMSIZE-USPSIZE-SSPSIZE-1 -> free mem
 DC.L  MEMSIZE-endOfStartUpImage-MEM_DEFSIZE-SSPSIZE-USPSIZE
 DC.L RAM+MEMSIZE-SSPSIZE-USPSIZE,RAM+MEMSIZE-SSPSIZE-1,0 -> user stack
 DC.L RAM+MEMSIZE-SSPSIZE,RAM+MEMSIZE-1,0    -> supervisor stack
-
initialFreeListSwap:
 DC.W 5
 DC.L ROM,START-1,0    -> vectors
 DC.L START,endOfStartUpImage-1,0    -> essential startup code
 DC.L endOfStartUpImage,endOfStartUpImage+MEM_DEFSIZE-1,0 -> the mem mgr buffer
 DC.L endOfStartUpImage+MEM_DEFSIZE,MEMSIZE-USPSIZE-SSPSIZE-1 -> free mem
 DC.L  MEMSIZE-endOfStartUpImage-MEM_DEFSIZE-SSPSIZE
 DC.L MEMSIZE-SSPSIZE-USPSIZE,MEMSIZE-SSPSIZE-1,0 -> user stack
 DC.L MEMSIZE-SSPSIZE,MEMSIZE-1,0    -> supervisor stack
```

F.3 Kernel.s68

This file contains the scheduler and signal handling functions and a few TRAPs for debugging and other things. The functions Connect and Event are not in this file, but in Timer.s68.

```
* WARNING! This file contains a lot of stack manipulation and non-standard
* branshes and is not intended for purists or weak-minded people. :-)
-
tReset:
 RESET
 MOVE    #$2700,SR
 MOVEA.L ROM,A7
 MOVEA.L ROM+4,A0
 JMP     (A0)
-
-
tStop:
 STOP #$2000
 RTE
-
-
debugTextPC:
 DC "PC:  $\0"
debugTextSR:
 DC "\nSR:  $\0"
debugTextUSP:
 DC "\nUSP: $\0"
debugTextSSP:
 DC "\nSSP: $\0"
 ALIGN
-
tDebug:
 MOVEM.L D0/A0,-(A7)
 LEA     (debugTextPC,PC),A0
 BSR     putStr
 SUBA.L  A0,A0
 MOVE.L  (14,A7),D0
 BSR     int2Dec
 LEA     (debugTextSR,PC),A0
 BSR     putStr
 SUBA.L  A0,A0
 MOVEQ   #0,D0
 MOVE.W  (12,A7),D0
 BSR     int2Dec
 LEA     (debugTextUSP,PC),A0
 BSR     putStr
 MOVE    USP,A0
 MOVE.L  A0,D0
 SUBA.L  A0,A0
 BSR     int2Dec
 LEA     (debugTextSSP,PC),A0
 BSR     putStr
 SUBA.L  A0,A0
 MOVE.L  A7,D0
 BSR     int2Dec
 MOVEQ   #10,D0
 BSR     putS
 MOVEM.L (A7)+,D0/A0
 RTE
-
* >A0=ptr to subroutine that must end with RTE
tUserTrap:
 JMP (A0)
-
-
* <Z=1 if in SVM, Z=0 if in user mode
tSuperVisor:
 BTST.B  #5,(A7)
 BEQ.S   tsvNo
  BSET.B #2,(1,A7)
  RTE
tsvNo:
 BCLR.B #2,(1,A7)
 RTE
-
-
* LF-clock interrupt
int3:
 MOVEM.L A4-A6,-(A7) -> BSET numreg*4 down the line!!!
 BSR     getEnvToA6
 TST.L   (ENV_SRBINT3,A6)
 BEQ.S   i3NoBeforeSub
  MOVEA.L (ENV_SRBINT3,A6),A5
  JSR     (A5)
i3NoBeforeSub:
 BTST.B  #ES_KERNEL,(ENV_STATUS,A6)
 BEQ.S   i3NoKernel
  MOVEA.L (ENV_KERNEL,A6),A5
  TST.B   (KRN_QCNT,A5)
  BNE.S   i3Cont
   MOVE.B  (KRN_QUANTUM,A5),(KRN_QCNT,A5)
   MOVE.B  (KRN_STATUS,A5),(-1,A7)
   ANDI.B  #%1011,(-1,A7) -> are we inhibited?
   BNE.S   i3NoKernel
    BSET.B  #KS_INHIBIT,(KRN_STATUS,A5)
    BTST.B  #5,(12,A7) -> STACK! Are we already in supervisor mode?
    BEQ.S   i3Normal
     MOVEA.L (KRN_SSPREF,A5),A4  -> intercept the last supervisor return by installing the intCatcher
     MOVE.L  -(A4),(KRN_TOPPC,A5)
     MOVE.W  -(A4),(KRN_TOPSR,A5)
     LEA     (intCatcher,PC),A5
     MOVE.W  #$2300,(A4)+
     MOVE.L  A5,(A4)
     BRA.S   i3NoKernel
i3Normal:
    MOVEM.L D0-D7/A0-A3,-(A7)
    MOVEA.L (A5),A1
    CMPI.B  #PS_RUNNING,(PCB_STATUS,A1)
    BNE.S   i3NotRun
     MOVE.B  #PS_READY,(PCB_STATUS,A1)
i3NotRun:
    LEA     (i3Back,PC),A0
    BRA     forceResched
i3Back:
    BEQ.S   i3Done
     BSET.B #KS_NOTODO,(KRN_STATUS,A5)
i3Done:
   BCLR.B  #KS_INHIBIT,(KRN_STATUS,A5)
   MOVEM.L (A7)+,D0-D7/A0-A3
   BRA.S   i3NoKernel
i3Cont:
  SUBQ.B #1,(KRN_QCNT,A5)
i3NoKernel:
 BTST.B  #ES_LFLED,(ENV_STATUS,A6)
 BEQ.S   i3out
  MOVE.B (ENV_LED,A6),LED
  ADDQ.B #1,(ENV_LED,A6)
i3out:
 TST.L   (ENV_SRAINT3,A6)
 BEQ.S   i3NoAfterSub
  MOVEA.L (ENV_SRAINT3,A6),A5 -> The timer device hooks up here
  JSR     (A5)
i3NoAfterSub:
 MOVEM.L (A7)+,A4-A6
 RTE
-
-
* This gets called when the last supervisor routine returns if some supervisor routine
* wanted to reschedule but other calls was already on the stack making it unpure.
intCatcher:
 SUBQ.L  #6,A7
 MOVEM.L D0-D7/A0-A6,-(A7)
 BSR     getEnvToA6
 MOVEA.L (ENV_KERNEL,A6),A5
 MOVE.W  (KRN_TOPSR,A5),(60,A7) -> STACK!
 MOVE.L  (KRN_TOPPC,A5),(62,A7)
 MOVEA.L (A5),A1  *-> A1=curpcb
 CMPI.B  #PS_RUNNING,(PCB_STATUS,A1)
 BNE.S   icNotRun
  MOVE.B  #PS_READY,(PCB_STATUS,A1)
icNotRun:
 LEA     (icBack,PC),A0
 BRA.S   forceResched
icBack:
 BEQ.S   icDone
  BSET.B  #KS_NOTODO,(KRN_STATUS,A5)
icDone:
 BCLR.B  #KS_INHIBIT,(KRN_STATUS,A5)
 MOVEM.L (A7)+,D0-D7/A0-A6
 RTE
-
* requires a pure stack with all 15 regs on it and SVM
* curpcb.status must be set.
* >A0=return addr, >A1=curpcb, >A5=KRN, >A6=ENV, <D0=0=switched,1=NoToDo
forceResched:
 MOVEA.L (A1),A2 *-> pcb:=cur.next
frLoop:
  MOVEQ   #NIL,D0 -> if pcb else pcb=beginning
  CMP.L   A2,D0
  BNE.S   frNotEnd
   MOVEA.L (ENV_PCB,A6),A2
frNotEnd:
  CMPI.B  #PS_READY,(PCB_STATUS,A2)
  BNE.S   frNext
   MOVEQ   #14,D0
   MOVEA.L A7,A3
   LEA     (PCB_REGS,A1),A4
frSt2Regs:
    MOVE.L (A3)+,(A4)+
   DBRA    D0,frSt2Regs
   MOVE.W  (A3)+,(PCB_SR,A1)
   MOVE.L  (A3),(PCB_PC,A1)
   MOVE    USP,A4
   MOVE.L  A4,(PCB_SP,A1)
   MOVEA.L (PCB_SP,A2),A4 -> now move in this new ready process
   MOVE    A4,USP
   MOVE.L  (PCB_PC,A2),(A3)
   MOVE.W  (PCB_SR,A2),-(A3)
   MOVEQ   #14,D0
   MOVEA.L A7,A3
   LEA     (PCB_REGS,A2),A4
frRegs2St:
    MOVE.L (A4)+,(A3)+
   DBRA    D0,frRegs2St
   MOVE.B  #PS_RUNNING,(PCB_STATUS,A2)
   MOVE.L  A2,(A5)
   BCLR.B  #KS_NOTODO,(KRN_STATUS,A5)
   MOVEQ   #0,D0
   JMP     (A0)
frNext:
  CMPA.L  A1,A2 -> if we've checked all and no process wants to run
  BEQ.S   frNoToDo
  MOVEA.L (A2),A2
 BRA.S frLoop
frNoToDo:
 MOVEQ #-1,D0
 JMP   (A0)
-
* To be called only from Block, Signal and finalReturn.
tReSchedule:
 BTST.B  #5,(A7) -> in supervisor mode already from signal???
 BNE.S   trsNoResched
  MOVEM.L D0-D7/A0-A6,-(A7)
  BSR     getEnvToA6
  MOVEA.L (ENV_KERNEL,A6),A5
  MOVEA.L (A5),A1 *-> A1=curpcb
  CMPI.B  #PS_RUNNING,(PCB_STATUS,A1)
  BNE.S   trsStart
   MOVE.B  #PS_READY,(PCB_STATUS,A1)
trsStart:
   LEA     (trsBack,PC),A0
   BRA     forceResched
trsBack:
   BNE.S   trsWait
    MOVE.B  (KRN_QUANTUM,A5),(KRN_QCNT,A5) -> reset the timer so it doesn't switch until one quantum again.
    BCLR.B  #KS_INHIBIT,(KRN_STATUS,A5)
    MOVEM.L (A7)+,D0-D7/A0-A6
    RTE -> return back to Block or Signal with a new user program on the stack
trsWait:
   BCLR.B #KS_SIGNAL,(KRN_STATUS,A5) *only place it's needed to clear this as it is the only place it is read.
   BSET.B #KS_NOTODO,(KRN_STATUS,A5)
   BCLR.B #KS_INHIBIT,(KRN_STATUS,A5)
trsStop:
    STOP   #$2000
    BCLR.B #KS_SIGNAL,(KRN_STATUS,A5)
   BEQ.S trsStop
  BRA.S trsStart
trsNoResched:
 MOVEM.L A5-A6,-(A7)
 BSR     getEnvToA6
 MOVEA.L (ENV_KERNEL,A6),A5
 MOVEA.L (KRN_SSPREF,A5),A6  -> intercept the last supervisor return
 MOVE.L  -(A6),(KRN_TOPPC,A5)
 MOVE.W  -(A6),(KRN_TOPSR,A5)
 LEA     (intCatcher,PC),A5
 MOVE.W  #$2300,(A6)+
 MOVE.L  A5,(A6)
 MOVEM.L (A7)+,A5-A6
 RTE
-
-
* This function is called by the last RTS when a user program exits.
finalReturn:
 MOVE.L  A6,-(A7)
 BSR     getEnvToA6
 MOVEA.L (ENV_KERNEL,A6),A6
 BSET.B  #KS_INHIBIT,(KRN_STATUS,A6)
 MOVEA.L (A6),A6
 MOVE.B  #PS_FINISHED,(PCB_STATUS,A6)
 MOVEA.L (A7)+,A6
 TRAP    #reschedule
 RTS * The same process will never return
-
-
* callable only from user mode, SVM is undefined
* >D0.L sigmask to wait for, <D0.L=signals received
block:
 MOVEM.L D1/A1/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_KERNEL,(ENV_STATUS,A6) -> do we have a running kernel?
 BEQ.S   bStop
  MOVEA.L (ENV_KERNEL,A6),A6
  BSET.B  #KS_INHIBIT,(KRN_STATUS,A6) -> inhibit multitasking while in Block.
  BCLR.B  #KS_FORBID,(KRN_STATUS,A6)
  MOVEA.L (A6),A1 -> A1=curpcb
  MOVE.L  D0,D1
  AND.L   (PCB_SIGSET,A1),D1
  BEQ.S   bNewTask -> this task didn't wait on any of the set signals or there are no set signals
   MOVE.L  D1,D0
   EOR.L   D1,(PCB_SIGSET,A1)
   BCLR.B  #KS_INHIBIT,(KRN_STATUS,A6) -> enable multitasking
   MOVEM.L (A7)+,D1/A1/A6
   RTS  *-> return D0 with the signals set and keep on with the same task.
bNewTask:
  MOVE.B  #PS_WAITING,(PCB_STATUS,A1)
  MOVE.L  D0,(PCB_SIGWAIT,A1)
  MOVEM.L (A7)+,D1/A1/A6
  TRAP    #reschedule
  RTS
bStop:
 TRAP    #stop
 MOVEM.L (A7)+,D1/A1/A6
 RTS
-
-
* >A0=pcb, D0.L=sigmask to signal to A0.
* <D0.L=-1 ok, 0=pcb doesn't exist
signal:
 MOVEM.L D1/A0/A5/A6,-(A7)
 BSR     getEnvToA6
 MOVEA.L (ENV_PCB,A6),A5
 MOVEQ   #NIL,D1
sigFind:
  CMP.L   A5,D1
  BEQ.S   sigNotFound
  CMPA.L  A0,A5
  BEQ.S   sigFound
  MOVEA.L (A5),A5
 BRA.S sigFind
sigFound:
 MOVEA.L (ENV_KERNEL,A6),A5
 MOVE.B  (KRN_STATUS,A5),(-1,A7)
 ANDI.B  #%1011,(-1,A7)
 OR.L    D0,(PCB_SIGSET,A0) -> set the actual sigbits
 MOVEQ   #TRUE,D0
 CMPI.B  #PS_WAITING,(PCB_STATUS,A0)
 BNE.S   sigNotWaiting
  MOVE.L (PCB_SIGWAIT,A0),D1
  AND.L  (PCB_SIGSET,A0),D1
  BEQ.S  sigOut
   MOVE.B #PS_READY,(PCB_STATUS,A0)
   BRA.S  sigSwitch
sigNotWaiting:
 CMPI.B #PS_READY,(PCB_STATUS,A0)
 BNE.S  sigOut
sigSwitch:
  TST.B   (-1,A7) ->inhibited?
  BNE.S   sigSignal
   BSET.B  #KS_INHIBIT,(KRN_STATUS,A5)
   MOVEM.L (A7)+,D1/A0/A5/A6
   TRAP    #reschedule
   RTS
sigSignal:
  BSET.B  #KS_SIGNAL,(KRN_STATUS,A5)
  BRA.S   sigOut
sigNotFound:
 MOVEQ   #FALSE,D0
sigOut:
 MOVEM.L (A7)+,D1/A0/A5/A6
 RTS
-
```

```
* <D0.B.L=bitnumber.B allocated, or -1.L if no free signals
allocSignal:
 MOVEM.L D1/A6,-(A7)
 BSR     getEnvToA6
 MOVEA.L (ENV_PCB,A6),A6
 MOVEA.L (A6)+,A6
 MOVE.L  (PCB_SIGMASK,A6),D1
 ANDI.L  #USERSIGNALS,D1
 CMPI.L  #USERSIGNALS,D1 -> All but reserved
 BEQ.S   asNoSig
  MOVEQ  #-1,D0
asNotFree:
   ADDQ.L #1,D0
   BTST.L D0,D1
  BNE.S  asNotFree
  BRA.S  asOut
asNoSig:
 MOVEQ #-1,D0
asOut:
 MOVEM.L (A7)+,D1/A6
_RTS
-
-
* >D0.B5=signal number to return
freeSignal:
 MOVEM.L D1/A6,-(A7)
 BSR     getEnvToA6
 MOVEA.L (ENV_KERNEL,A6),A6
 MOVEA.L (A6)+,A6
 MOVE.L  (PCB_SIGMASK,A6),D1
 BCLR.L  D0,D1
 MOVE.L  D1,(PCB_SIGMASK,A6)
 MOVEM.L (A7)+,D1/A6
_RTS
-
-
* >D0.B <>0=forbid multitasking, =0=permit multitasking.
* A Block() automatically permits multitasking.
singleTask:
 MOVE.L  A6,-(A7)
 BSR     getEnvToA6
 MOVEA.L (ENV_KERNEL,A6),A6
 TST.B   D0
 BEQ.S   stPermit
  BSET.B #KS_FORBID,(KRN_STATUS,A6)
  BRA.S  stOut
stPermit:
 BCLR.B #KS_FORBID,(KRN_STATUS,A6)
stOut:
 MOVEA.L (A7)+,A6
 RTS
```

F.4 Memhandling.s68

This file contains the memory manager and all functions associated with it.

```
MI_CMD_SIZE EQU     4
MI_INFO_SIZE    EQU     4
-
***************************************************
* Rensa upp i frstrande funktioner !!!!!!!!!!    OK 1703
*
* Jmna minnesadresser                        OK 0302
* MOVEM p flyttning i minnet                      OK 0302
* Stt maxgrns p free lists storlek           OK 0302
* Komprimering av sammanhngande minne i free list OK 0302
*
* Sammanslagning av sammanhngande fritt minne vid
* allokering
*
* Deallokering av icke utnyttjat minne.
***************************************************
-
*getEnvToA6:
*   MOVEA.L #envvars,A6
*   RTS
-
*envvars:
*   DC.W    $10
*   DC.L    $5000
-
*********************************************
* Input:    size in D0.L
* Output:   index in D1.W, $FFFF in D1 if none
seekFreeBlock:
    MOVEM.L A0/D2,-(A7)
    CMPI.W  #$FFFF,(A6)
    BEQ     sfb_noneFound
    MOVE.W  #0,D1
    LEA     (2,A6),A0
    ADDA.L  #8,A0
sfb_nextBlock:
    MOVE.L  (A0),D2
    CMP.L   D0,D2
    BCC     sfb_memFound
    CMP.W   (A6),D1
    BEQ     sfb_noneFound
    ADD.W   #1,D1
    ADDA.L  #12,A0
    BRA     sfb_nextBlock
sfb_noneFound:
    MOVE.L  #$FFFF,D1
sfb_memFound:
    MOVEM.L (A7)+,A0/D2
    RTS
-
-
********************************************
* Input:    index in D1.W where the hole will be
* Output:   index + 1 set in memory
makeHole:
    MOVEM.L D2-D5/A2,-(A7)
    MOVE.W  (A6),D3
    MULU    #FR_LST_E_SIZE,D3   // Block's adress in D3
    LEA     (2,A6),A2
    MOVE.W  (A6),D2
    SUB.W   D1,D2           // Counter in D2
    ADDA.L  D3,A2
moveBlocksDown:
     MOVEM.L    (0,A2),D3-D5
     MOVEM.L    D3/D4/D5,(FR_LST_E_SIZE,A2)
     SUBA.L #FR_LST_E_SIZE,A2
     SUBI.W #1,D2
     CMPI.W #$FFFF,D2
    BNE     moveBlocksDown
    ADDI.W  #1,(A6)
    MOVEM.L (A7)+,D2-D5/A2
    RTS
-
-
********************************************
* Input:    index in D1.W
* Output:   index - 1 set in memory
listRemoveBlock:
    MOVEM.L D0/D2-D5/A1,-(A7)
    MOVE.W  D1,D0
    MULU    #FR_LST_E_SIZE,D0
    LEA     (2,A6),A1
    MOVE.W  (A6),D2
    SUB.W   D1,D2
removeBlock:
     MOVEM.L    (FR_LST_E_SIZE,A1,D0.L),D3-D5
     MOVEM.L    D3-D5,(0,A1,D0.L)
     ADDI.L #FR_LST_E_SIZE,D0
     SUBI.W #1,D2
     CMPI.W #$FFFF,D2
    BNE     removeBlock
    SUBI.W  #1,(A6)
    MOVEM.L (A7)+,D0/D2-D5/A1
    RTS
-
-
********************************************
* Input:    wanted size in D0.L, index in D1.W
* Output:   adress in A0.L
listGetMem:
    MOVEM.L D0/D2-D3/A1,-(A7)
    MOVE.L  D1,D3
    MULU    #FR_LST_E_SIZE,D3   // Rkna fram adress
    LEA     (2,A6),A1
    ADDA.L  D3,A1           // Adress + offset
    MOVEA.L (A1),A0         // Lgg RAM adress i A0
    CMP.L   (8,A1),D0
    BEQ     allMemInBlock   // nskat minne tar upp helt block
    BSR     makeHole        // Block ska splittras
    MOVE.L  (8,A1),D2
    SUB.L   D0,D2
    MOVE.L  D2,(20,A1)
    MOVE.L  (A1),D2
    ADD.L   D0,D2
    MOVE.L  D2,(4,A1)
    MOVE.L  (4,A1),(12,A1)
    SUBI.L  #1,(4,A1)
allMemInBlock:
    MOVE.L  #0,(8,A1)
    MOVEM.L (A7)+,D0/D2-D3/A1
    RTS
-
********************************************
* Input:    size in D0.L
* Output:   adress in A0.L, else 0 in A0.L
tAllocMem:
    MOVEM.L D0-D1/A5/A6,-(A7)
    TST.L   D0
    BEQ.S   am_noMemFree
    BSR     getEnvToA6
    MOVEA.L A6,A5
    MOVEA.L (ENV_FREEL,A6),A6
    MOVE.W (A6),D1          // Check if free limit reached
    CMP.W   (ENV_FREEZ,A5),D1       //
    BEQ     am_expandFreeList
    ADDI.L  #1,D0             // Round up to even bytes
    ANDI.L  #$FFFFFFFE,D0   //
    BSR     seekFreeBlock
    CMPI.W  #$FFFF,D1
    BEQ     am_noMemFree
    BSR     listGetMem
    BRA.S   amOut
am_noMemFree:
    MOVEA.L #0,A0
    BRA.S   amOut
am_expandFreeList:
* VAD SKA GRAS HR!!!!!!! CompFreeList
    MOVEA.L #0,A0
amOut:
    MOVEM.L (A7)+,D0-D1/A5/A6
    RTE
-
********************************************
* Input:    Adress in A0.W
* Output:   Index in D1.W, otherwise $FFFF in D1.W
seekBlock:
    MOVEM.L A0-A1,-(A7)
    MOVE.W  #0,D1           // Reset counter
    LEA     (2,A6),A1
seekB:
     CMPA.L (A1),A0
     BEQ    sb_blockFound
     CMP.W  (A6),D1
     BEQ    sb_blockNotFound
     ADDI.W #1,D1
     ADDA.L #FR_LST_E_SIZE,A1
    BRA     seekB
sb_blockFound:
    BRA.S   sbOut
sb_blockNotFound:
    MOVE.W  #$FFFF,D1
sbOut
    MOVEM.L (A7)+,A0-A1
    RTS
-
********************************************
* Input:    Adress in A0.L
* Output:   Memory returned to free list
tFreeMem:
    MOVEM.L D1-D2/A1/A6,-(A7)
    BSR     getEnvToA6
    MOVEA.L (ENV_FREEL,A6),A6
* Leta upp index med startadress A0
    BSR     seekBlock
    CMPI.W  #$FFFF,D1
    BEQ     tfmError
* Rkna ut storlek ( end-start ) och lgg in.
    MULU    #FR_LST_E_SIZE,D1
    LEA     (2,A6),A1
    MOVE.L  (4,A1,D1.L),D2
    SUB.L   (A1,D1),D2
    ADDI.L  #1,D2
    MOVE.L  D2,(8,A1,D1.L)
    BRA.S   tfmOut
tfmError:
* VAD SKA GRAS HR!!!!!!!
    NOP
tfmOut:
    MOVEM.L (A7)+,D1-D2/A1/A6
    RTE
-
********************************************
* Input:    None
* Output:   free list compressed
compFreeList:
    MOVEM.L D0-D5/A0/A6,-(A7)
    BSR     getEnvToA6
    MOVEA.L (ENV_FREEL,A6),A6
    MOVE.W  (A6),D0
*   MOVE.W  FR_LAST_INDX,D0 // Offset in D0
    TST.W   D0
    BEQ     cfl_compressDone
    MULU    #FR_LST_E_SIZE,D0 //
    LEA     (2,A6),A0
*   MOVEA.L #FREE_LIST,A0   // Start of free list in A0
cfl_Loop:
    MOVE.L  (8,A0,D0.L),D1
    TST.L   (8,A0,D0.L) // Check if free block
    BEQ     cfl_doLoop
    TST.L  (-4,A0,D0.L) // Check earlier block
    BEQ     cfl_doLoop
    MOVE.L  (4,A0,D0.L),D1  // Merge
    MOVE.L  (-4,A0,D0.L),D2
    ADD.L   (8,A0,D0.L),D2
    MOVEM.L D1/D2,(-8,A0,D0.L)
    MOVE.L  D0,D1           // Move blocks up
    MOVE.W (A6),D2
*   MOVE.W  FR_LAST_INDX,D2
    MULU    #FR_LST_E_SIZE,D2
    ADDI.W  #FR_LST_E_SIZE,D2
cfl_moveBlocksUp:
     MOVEM.L    (12,A0,D1.L),D3-D5
     MOVEM.L    D3/D4/D5,(A0,D1.L)
     ADDI.W #FR_LST_E_SIZE,D1
     CMP.W  D2,D1
    BNE     cfl_moveBlocksUp
    SUBI.W  #1,(A6)
*   SUBI.W  #1,FR_LAST_INDX
cfl_doLoop:
    SUBI.W  #FR_LST_E_SIZE,D0
    TST.W   D0
    BNE     cfl_Loop
cfl_compressDone:
    MOVEM.L (A7)+,D0-D5/A0/A6
    RTS
-
```

```
-
-
*******************************************
* NEW
* >A0.L = Address to taglist (CMD.L,INFO.L,...,0)
*
memInfo:
    MOVEM.L A6/D0-D3,-(A7)
    BSR     getEnvToA6
    MOVEA.L (ENV_FREEL,A6),A6   // Address to FreeList in A6
    MOVE.L  #0,D3               // OFFSET
miLoop:
    CMPI.L  #0,(A0,D3)          // End of taglist
    BEQ.S   miOut
-
    CMPI.L  #MI_TOTALFREE,(A0,D3)
    BNE.S   mi1
    BSR     miFreeMem
    MOVE.L  D0,(MI_CMD_SIZE,A0,D3)
    BRA.S   miDoLoop
mi1:
    CMPI.L  #MI_LARGEST,(A0,D3)
    BNE.S   mi2
    BSR     miLargest
    MOVE.L  D0,(MI_CMD_SIZE,A0,D3)
    BRA.S   miDoLoop
mi2:
    CMPI.l  #MI_NUMALLOC,(A0,D3)
    BNE.S   mi3
    BSR     miNumAlloc
    MOVE.L  D0,(MI_CMD_SIZE,A0,D3)
    BRA.S   miDoLoop
mi3:
    CMPI.L  #MI_NUMFREE,(A0,D3)
    BNE.S   miDoLoop
    BSR     miNumFree
    MOVE.L  D0,(MI_CMD_SIZE,A0,D3)
miDoLoop:
    ADDI.L  #MI_CMD_SIZE+MI_INFO_SIZE,D3
    BRA.S   miLoop
miOut:
    MOVEM.L   (A7)+,A6/D0-D3
    RTS
-
miFreeMem:
    MOVE.L  #0,D0       // Reset sum
    MOVE.W  (A6),D1     // Counter in D1
    MOVE.L  #0,D2       // Offset
mifmLoop:
    TST.L   (2+8,A6,D2.L)   // Check if free block
    BEQ.S   mifmDoLoop
    ADD.L   (2+8,A6,D2.L),D0    // Add to Sum
mifmDoLoop:
    ADDI.L  #FR_LST_E_SIZE,D2   // Increase offset
    DBRA    D1,mifmLoop     // Check if more
    BRA.S   miOut2
-
miLargest:
    MOVE.L  #0,D0       // Reset largest
    MOVE.W  (A6),D1     // Counter in D2
    MOVE.L  #0,D2       // Offset
milLoop:
    TST.L   (2+8,A6,D2.L)   // Check if free block
    BEQ.S   milDoLoop
    CMP.L   (2+8,A6,D2.L),D0
    BGT     milDoLoop
    MOVE.L  (2+8,A6,D2.L),D0    // New largest
milDoLoop:
    ADDI.L  #FR_LST_E_SIZE,D2   // Increase offset
    DBRA    D1,milLoop          // Check if more
    BRA.S   miOut2
-
miNumAlloc:
    MOVE.L  #0,D0       // Reset sum
    MOVE.W  (A6),D1     // Counter in D2
    MOVE.L  #0,D2       // Offset
minaLoop:
    TST.L   (2+8,A6,D2.L)   // Check if free block
    BNE.S   minaDoLoop
    ADDI.L  #1,D0           // Increase Sum
minaDoLoop:
    ADDI.L  #FR_LST_E_SIZE,D2   // Increase offset
    DBRA    D1,minaLoop     // Check if more
    BRA.S   miOut2
-
miNumFree:
    MOVE.L  #0,D0       // Reset sum
    MOVE.W  (A6),D1     // Counter in D2
    MOVE.L  #0,D2       // Offset
minfLoop:
    TST.L   (2+8,A6,D2.L)   // Check if free block
    BEQ.S   minfDoLoop
    ADDI.L  #1,D0           // Increase Sum
minfDoLoop:
    ADDI.L  #FR_LST_E_SIZE,D2   // Increase offset
    DBRA    D1,minfLoop     // Check if more
    BRA.S   miOut2
    NOP
-
miOut2:
    RTS
-
-
```

```
************************************************
* >A0.L = Start address, even to check even addresses
*         odd to check odd addresses
* >D0.L = Amount of bytes to check
* D0.L> = 0 = OK,  1 = Not OK
memCheck:
    MOVEM.L D1/A0,-(A7)
    SUBI.L  #1,D0
    CMPI.L  #-1,D0
    BEQ.S   mcError
mcLoop:
    MOVE.B  (A0),D1
    MOVE.B  #55,(A0)
    CMPI.B  #55,(A0)
    BNE.S   mcError
    MOVE.B  D1,(A0)
    ADDA.L  #2,A0
    SUBI.L  #1,D0
    CMPI.L  #-1,D0
    BNE.S   mcLoop
mcOk:
    CLR.L   D0
    BRA.S   mcOut
mcError:
    MOVE.L  #1,D0
mcOut:
    MOVEM.L (A7)+,D1/A0
    RTS
```

F.5 Timer.s68

This file contains the timer device. It also includes the functions Connect and Event which are currently only associated with the timer.

```
*************************************************
* >D0.B=Device TIMER
* >D1.W=Unit number
* >D2.B=Type CONHANDL OR CONSIG
* >D3.L=Address to userdata OR signal number
* >A0.L=Address to interrupt handler OR process
connect:
    MOVEM.L D0/D4/D5/A0/A1/A6,-(A7)
    MOVEA.L A0,A1
    BSR     getEnvToA6
    MOVEA.L (ENV_CONNECT,A6),A6
* Kolla om paret Device och Unit finns redan
conChkExist:
    TST.L   (A6)
    BEQ.S   conMoveOn
    CMP.B   (CONOFFDEV,A6),D0
    BNE.S   conChkNext
    CMP.W   (CONOFFUNIT,A6),D1
    BNE.S   conChkNext
    BRA.S   conExist
conChkNext:
    MOVEA.L (A6),A6
    BRA.S   conChkExist
conMoveOn:
* Om paret inte finns allokera minne och lgg in.
    MOVE.B  D0,D5
    MOVE.L  #CONBLKSIZE,D0
    TRAP    #allocmem
* Koll att minne fanns
    MOVE.L  A0,D4
    TST.L   D4      * Memsize = 0 ?
    BEQ.S   conErr
* Minne fanns, lgg in
    BSR     getEnvToA6
    MOVE.L  (ENV_CONNECT,A6),(A0)
    MOVE.B  D5,(CONOFFDEV,A0)
    MOVE.W  D1,(CONOFFUNIT,A0)
    MOVE.B  D2,(CONOFFTYPE,A0)
    MOVE.L  D3,(CONOFFUADR,A0)
    MOVE.L  A1,(CONOFFIADR,A0)
    MOVE.L  A0,(ENV_CONNECT,A6)
    BRA.S   conOut
conExist:
    NOP
conErr:
*   FEL!!!!!!!! Vad gra?
    NOP
conOut:
    MOVEM.L (A7)+,D0/D4/D5/A0/A1/A6
    RTS
-
************************************************
* >D0.B=Device e.g. TIMER
* >D1.W=Unit number
event:
    MOVEM.L D0/D1/D2/A0/A1/A6,-(A7)
    BSR     getEnvToA6
    MOVEA.L (ENV_CONNECT,A6),A6
    MOVE.L  A6,D2
    TST.L   D2
    BEQ.S   evOut
* Kolla om Device och Unit finns
evChkExist:
    CMP.B   (CONOFFDEV,A6),D0
    BNE.S   evChkNext
    CMP.W   (CONOFFUNIT,A6),D1
    BNE.S   evChkNext
* Type CONHANDL or CONSIG
* Address to userdata or signal number
* Address to interrupt handler or process
    CMPI.B  #CON_HANDL,(CONOFFTYPE,A6)
    BEQ.S   evConHandl
    CMPI.B  #CON_SIG,(CONOFFTYPE,A6)
    BNE.S   evOut
evConSig:
    MOVEA.L (CONOFFIADR,A6),A0
    MOVE.L  (CONOFFUADR,A6),D0
    MOVEA.L 4,A5
    JSR     (Signal,A5)
    BNE.S   evOut
* Processen finns inte. Device borde tas bort.
    BRA.S   evOut
evConHandl:
    MOVEA.L (CONOFFUADR,A6),A0
    MOVEA.L (CONOFFIADR,A6),A1
    JSR     (A1)
    BRA.S   evOut
evChkNext:
    MOVEA.L (A6),A6
    MOVE.L  A6,D2
    TST.L   D2
    BNE.S   evChkExist
evOut:
    MOVEM.L (A7)+,D0/D1/D2/A0/A1/A6
    RTS
-
* Jump list for timer commands
tflist:
    JMP     (timerReset,PC)
    JMP     (timerStart,PC)
    JMP     (timerStop,PC)
    JMP     (timerRead,PC)
    JMP     (timerSet,PC)
    JMP     (timerReg,PC)
    JMP     (timerUnReg,PC)
    JMP     (timerDeviceInit,PC)
****************************************************
* >D3.L=Command
* <D0.L=Result if there is any (see above)
tTimer:
    MOVEM.L D1-D7/A0-A6,-(A7)
    BSR     getEnvToA6
    CMPI.L  #TC_INIT,D3
    BEQ.S   ttCont
    TST.L   (ENV_TIMER,A6)
    BNE.S   ttCont
    MOVEQ   #-1,D0
    BRA.S   ttOut
ttCont:
    MOVEA.L (ENV_TIMER,A6),A0
    LEA     (tflist,PC),A1
    JSR     (A1,D3)
ttOut:
    MOVEM.L (A7)+,D1-D7/A0-A6
    RTE
-
****************************************************
* Timer initialization
timerDeviceInit:
    MOVE.L  #MAX_EL_TA*TIMERELSIZE+TOFFSET,D0
    TRAP    #allocmem
* Check to see if we have memory
    MOVE.L  A0,(ENV_TIMER,A6)
    BEQ.S   tdiOut
    MOVEQ   #0,D1
    SUBQ.L  #1,D0
tdiNullLoop:
    MOVE.B  D1,(A0)+
    DBRA    D0,tdiNullLoop
    MOVEA.L (ENV_TIMER,A6),A0
    MOVE.W  #TDIVISOR,(DIVOFFSET,A0)
    LEA     (timerAdvance,PC),A0
    MOVE.L  A0,(ENV_SRAINT3,A6)
    MOVE.B  #%00000001,(ENV_LFC,A6)
    MOVE.B  (ENV_LFC,A6),LFCLK
    BCLR.B  #EM_LFCINH,(ENV_MCR,A6)
    MOVE.B  (ENV_MCR,A6),MCR
tdiOut:
    RTS
-
**************************************************************
* <D0.L=Offset to registred timer, No more timers=-1 (L)
timerReg:
    MOVEM.L D1/A0,-(A7)
    MOVEA.L (ENV_TIMER,A6),A0
    CLR.L   D0
trLoop:
    BTST.B #TSB_UNREGREG,(TO_STATUS+TOFFSET,A0,D0)
    BEQ.S   trDone
trNext:
    ADDQ.W  #TIMERELSIZE,D0
    CMPI.W  #MAX_EL_TA*TIMERELSIZE,D0
    BNE.S   trLoop
    MOVEQ   #-1,D0
    BRA.S   trOut
trDone:
    BCLR.B  #TSB_OFFON,(TO_STATUS+TOFFSET,A0,D0)
    BSET.B  #TSB_UNREGREG,(TO_STATUS+TOFFSET,A0,D0)
-
** SKA TESTAS *****************
* Koll mste ske om D0 strre n LastIndex
    MOVE.W  D0,D1
    ADDI.W  #TIMERELSIZE,D1
    CMP.W   (A0),D1
    BLE.S   trOut
    MOVE.W  D1,(A0)
*******************
-
trOut:
    MOVEM.L (A7)+,D1/A0
    RTS
-
**************************************************************
* >A0.L=Adress to timerArray
* >D0.W=TimerNr
timerUnReg:
    MOVEM.L D1,-(A7)
* SKA TESTAS **********************************
* Om D0 r lika med offset till sista registrerade, fixa det.
    MOVE.W  (A0),D1
    SUBI.W  #TIMERELSIZE,D1
    CMP.W   D1,D0
    BNE.S   turNotLast
    MOVE.W  D1,(A0)
turNotLast:
*********************
    BCLR.B  #TSB_UNREGREG,(TO_STATUS+TOFFSET,A0,D0)
    MOVEM.L (A7)+,D1
    RTS
-
**************************************************************
* >A0.L=Adress to timerArray
* >D0.W=TimerNr
* <D0.L=Current timer value
timerRead:
    MOVEM.L D1,-(A7)
    BTST.B  #TSB_DOWNUP,(TO_STATUS+TOFFSET,A0,D0)
    BNE.B   treUp
    MOVE.W  (TO_COUNTER+TOFFSET,A0,D0),D0
    BRA.S   treOut
treUp:
    MOVE.W  (TO_COUNTER+TOFFSET,A0,D0),D1
    ADD.W   (TO_STARTVALUE+TOFFSET,A0,D0),D1
    MOVE.W  D1,D0
treOut:
    MULU    (DIVOFFSET,A0),D0
    MOVEM.L (A7)+,D1
    RTS
-
**************************************************************
* >A0.L=Address to timerArray
* >D0.W=TimerNr
timerStart:
    BSET.B  #TSB_OFFON,(TO_STATUS+TOFFSET,A0,D0)
    RTS
-
**************************************************************
* >A0.L=Address to timerArray
* >D0.W=TimerNr
timerStop:
    BCLR.B  #TSB_OFFON,(TO_STATUS+TOFFSET,A0,D0)
    RTS
-
**************************************************************
* >A0.L=Address to timerArray
* >D0.W=TimerNr
* >D1.B=Bitmask 0/1 7:UnReg/Reg, 6:Off/On 5:Down/Up, 4:Repeat/NoRepeat
* >D2.L=Startvalue
timerSet:
    MOVEM.L D1-D2,-(A7)
    CMP.W   (A0),D0
* SIGNAL ?
    BGT     tSetDone
    BTST.B  #TSB_UNREGREG,(TO_STATUS+TOFFSET,A0,D0)
    BEQ.S   tSetDone
    BSET.B  #TSB_UNREGREG,D1
    MOVE.B  D1,(TO_STATUS+TOFFSET,A0,D0)
    BCLR.B  #TSB_OFFON,(TO_STATUS+TOFFSET,A0,D0)
    DIVU    (DIVOFFSET,A0),D2
    MOVE.W  D2,(TO_STARTVALUE+TOFFSET,A0,D0)
    BTST.B  #TSB_DOWNUP,(TO_STATUS+TOFFSET,A0,D0)
    BNE.B   tSetUp
    MOVE.W  (TO_STARTVALUE+TOFFSET,A0,D0),(TO_COUNTER+TOFFSET,A0,D0)
    BRA.S   tSetDone
tSetUp:
    CLR.W   (TO_COUNTER+TOFFSET,A0,D0)
    MOVE.W  (TO_STARTVALUE+TOFFSET,A0,D0),D1
    SUB.W   D1,(TO_COUNTER+TOFFSET,A0,D0)
tSetDone:
    MOVEM.L (A7)+,D1-D2
    RTS
-
**************************************************************
* >A0.L=Address to timerArray
* >D0.W=TimerNr
timerReset:
    MOVEM.L D1-D2,-(A7)
    CMP.W   (A0),D0
* SIGNAL ?
    BGT     tRstDone
    BTST.B  #TSB_UNREGREG,(TO_STATUS+TOFFSET,A0,D0)
    BEQ.S   tRstDone
    BTST.B  #TSB_DOWNUP,(TO_STATUS+TOFFSET,A0,D0)
    BNE.B   tRstUp
    MOVE.W  (TO_STARTVALUE+TOFFSET,A0,D0),(TO_COUNTER+TOFFSET,A0,D0)
    BRA.S   tRstDone
tRstUp:
    MOVE.B  (TO_STATUS+TOFFSET,A0,D0),D1
    BCLR.B  #TSB_OFFON,(TO_STATUS+TOFFSET,A0,D0)
    CLR.W   (TO_COUNTER+TOFFSET,A0,D0)
    MOVE.W  (TO_STARTVALUE+TOFFSET,A0,D0),D2
    SUB.W   D2,(TO_COUNTER+TOFFSET,A0,D0)
    MOVE.B  D1,(TO_STATUS+TOFFSET,A0,D0)
tRstDone:
    MOVEM.L (A7)+,D1-D2
    RTS
-
************************************************************
* >A0.L=Address to timerArray
timerAdvance:
    MOVEM.L D0-D2/A0/A6,-(A7)
    BSR     getEnvToA6
    MOVEA.L (ENV_TIMER,A6),A0
    MOVE.W  (A0),D0
    TST.W   D0
    BEQ.S   taDone
    ADDQ.L  #TOFFSET,A0
taLoop:
    SUBQ.W  #TIMERELSIZE,D0
    BTST.B #TSB_UNREGREG,(TO_STATUS,A0,D0)
    BEQ.S   taNext
    BTST.B  #TSB_OFFON,(TO_STATUS,A0,D0)
    BEQ.S   taNext
    BTST.B  #TSB_DOWNUP,(TO_STATUS,A0,D0)
    BEQ.S   taDown
    ADDQ.W  #1,(TO_COUNTER,A0,D0)
    BRA.S   taChkZero
taDown:
    SUBQ.W  #1,(TO_COUNTER,A0,D0)
taChkZero:
    BNE.S   taNext
    MOVE.L  D0,D2       // Save
    MOVE.W  D0,D1
    MOVE.B  #DEV_TIMER,D0
    BSR     event
    MOVE.L  D2,D0       // Restore
    BTST.B  #TSB_REPNOREP,(TO_STATUS,A0,D0)
    BNE.S   taNoRep
* Reset grs hr
    BTST.B  #TSB_DOWNUP,(TO_STATUS,A0,D0)
    BNE.B   taRstUp
    MOVE.W  (TO_STARTVALUE,A0,D0),(TO_COUNTER,A0,D0)
    BRA.S   taRstDone
taRstUp:
    CLR.W   (TO_COUNTER,A0,D0)
    MOVE.W  (TO_STARTVALUE,A0,D0),D1
    SUB.W   D1,(TO_COUNTER,A0,D0)
taRstDone:
    BRA.S   taNext
taNoRep:
    BCLR.B  #TSB_OFFON,(TO_STATUS,A0,D0)
taNext:
    TST.W   D0
    BNE.S   taLoop
taDone:
    MOVEM.L (A7)+,D0-D2/A0/A6
    RTS
```

F.6 Serial.s68

This file contains all serial routines including the interrupt routine associated with the serial circuit.

```
*LEDs on the slot-bus card:
-
*1=Reset serial - Reset the 16C550, no LED
*0=RxIRQ        - Reciever interrupt enabled (orange narrow flat)
*2=TxIRQ        - Transmitter interrupt enabled (orange narrow flat)
*4=FIFO         - FIFOs enabled (green narrow flat)
*6=DataError    - 16C550 has detected a data-error (red flat)
*7=RxOverRun    - Memory receive buffer of 1kB is overflown (red flat)
*3,5=unused
-
*RxRDY          - Signal from the 16C550 (5mm transp. orange)
*TxRDY          - Signal from the 16C550 (5mm transp. green)
-
* Pre: Needs envvars in A6
initSerial:
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BNE.S   isSerIsInited
 MOVE.B #%10,SER_LED ->reset serial
 NOP
 MOVE.B #0,SER_LED
 MOVE.B SER_SPR,D0 ->test if serial exists
 ADDI.B #$55,D0
 MOVE.B D0,SER_SPR
 CMP.B  SER_SPR,D0
 BNE    isNotAttached
  MOVE.W #SER_16X9600,D0
  BSR.S  setSerSpeed -> this sets 8N1
isSerIsInited:
  MOVE.B #SB_FCR_FIFORESET|SB_FCR_RXTRIG_4,SER_FCR
  MOVE.B #%00010001,SER_SPR ->indicate RxIRQ and FIFO on the LEDs
  MOVE.B SER_SPR,SER_LED
  BTST.B #ES_SWAP,(ENV_STATUS,A6)
  BEQ.S  isNoSwap
   MOVE.L #serInpFifo,(ENV_SERIPTR,A6)
   MOVE.L #serOutFifo,(ENV_SEROPTR,A6)
   BRA.S  isWasSwap
isNoSwap:
   MOVE.L #RAM+serInpFifo,(ENV_SERIPTR,A6)
   MOVE.L #RAM+serOutFifo,(ENV_SEROPTR,A6)
isWasSwap:
  MOVEQ  #0,D0
  MOVE.L D0,(ENV_SERIBEG,A6)
  MOVE.L D0,(ENV_SERIEND,A6)
  MOVE.L D0,(ENV_SEROBEG,A6)
  MOVE.L D0,(ENV_SEROEND,A6)
  BSET.B #ES_SER,(ENV_STATUS,A6)
  BSET.B #5,(ENV_LED,A6)
  MOVE.B #%00000001,SER_IER ->Rx IRQ enabled
  RTS
isNotAttached:
 BCLR.B #ES_SER,(ENV_STATUS,A6)
 RTS
-
-
* Pre: Serial must exist
setSerSpeed:
 MOVE.B #SB_LCR_8N1|SB_LCR_DL,SER_LCR
 MOVE.B D0,SER_DLL
 LSR.W  #8,D0
 MOVE.B D0,SER_DLM
 MOVE.B #SB_LCR_8N1,SER_LCR
 RTS
-
-
* <>A0.L=ptr to taglist that will be filled with data.
getSerInfo:
 MOVEM.L D0/A0/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ     gsiOut
gsiLoop:
  TST.L   (A0)
  BEQ     gsiOut
   MOVE.L  (A0)+,D0
   CLR.L   (A0)
   CMPI.L  #SI_SPEED,D0
   BNE.S   gsiNoSpeed
    MOVE.B #SB_LCR_8N1|SB_LCR_DL,SER_LCR
    MOVE.B SER_DLM,D0
    LSL.W  #8,D0
    MOVE.B SER_DLL,D0
    MOVE.B #SB_LCR_8N1,SER_LCR
    MOVE.L D0,(A0)
   BRA     gsiMore
gsiNoSpeed:
   CMPI.L  #SI_LINESTATUS,D0
   BNE.S   gsiNoLine
    MOVE.B SER_LSR,(3,A0)
   BRA     gsiMore
gsiNoLine:
   CMPI.L  #SI_MODSTATUS,D0
   BNE.S   gsiNoModem
    MOVE.B SER_MSR,(3,A0)
   BRA.S   gsiMore
gsiNoModem:
   CMPI.L  #SI_FIFOSIZE,D0
   BNE.S   gsiNoSize
    MOVE.L #FIFO_SIZE,(A0)
   BRA.S   gsiMore
gsiNoSize:
   CMPI.L  #SI_RXFIFOLOC,D0
   BNE.S   gsiNoRxLoc
    MOVE.L (ENV_SERIPTR,A6),(A0)
   BRA.S   gsiMore
gsiNoRxLoc:
   CMPI.L  #SI_TXFIFOLOC,D0
   BNE.S   gsiNoTxLoc
    MOVE.L (ENV_SEROPTR,A6),(A0)
   BRA.S   gsiMore
gsiNoTxLoc:
   CMPI.L  #SI_RXFILLLEV,D0
   BNE.S   gsiNoRxFill
    MOVE.L (ENV_SERIEND,A6),D0
    SUB.L  (ENV_SERIBEG,A6),D0
    MOVE.L D0,(A0)
   BRA.S   gsiMore
gsiNoRxFill:
   CMPI.L  #SI_TXFILLLEV,D0
   BNE.S   gsiNoTxFill
```

```
    MOVE.L (ENV_SEROEND,A6),D0
    SUB.L  (ENV_SEROBEG,A6),D0
    MOVE.L D0,(A0)
   BRA.S   gsiMore
gsiNoTxFill:
   CMPI.L  #SI_RXTOTAL,D0
   BNE.S   gsiNoRxTotal
    MOVE.L (ENV_SERIEND,A6),(A0)
   BRA.S   gsiMore
gsiNoRxTotal:
   CMPI.L  #SI_TXTOTAL,D0
   BNE.S   gsiNoTxTotal
    MOVE.L (ENV_SEROEND,A6),(A0)
gsiNoTxTotal:
gsiMore:
   ADDQ.L #4,A0
  BRA gsiLoop
gsiOut:
 MOVEM.L (A7)+,D0/A0/A6
_RTS
-
putStr:
 MOVEM.L D0/A1,-(A7)
 MOVEQ   #-1,D0
 MOVEA.L A0,A1
psCount:
  ADDQ.L #1,D0
  TST.B  (A1)+
 BNE.S   psCount
 BSR     sendS
 MOVEM.L (A7)+,D0/A1
_RTS
-
flushRx:
 MOVEM.L A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   frOut
  MOVE.L  (ENV_SERIBEG,A6),(ENV_SERIEND,A6)
  MOVE.B  #SB_FCR_RXRESET,SER_FCR
frOut:
 MOVEM.L (A7)+,A6
_RTS
-
flushTx:
 MOVEM.L D0/A5/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   ftOut
  MOVEA.L (ENV_KERNEL,A6),A5
ftWait:
   MOVE.L (ENV_SEROEND,A6),D0
   CMP.L  (ENV_SEROBEG,A6),D0
   BEQ.S  ftUntilEmpty
    BSET.B #KS_SERINTSIGTX,(KRN_STATUS,A5)
    MOVE.L #SIG_SERIAL_TX,D0
    BSR    block
  BRA.S ftWait
ftUntilEmpty:
   BTST.B  #6,SER_LSR
  BEQ.S   ftUntilEmpty
ftOut:
 MOVEM.L (A7)+,D0/A5/A6
_RTS
-
putS:
 MOVEM.L D0-D3/A0/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   tpsOut
  BTST.B #6,SER_LSR
  BEQ.S  tpsPutInFIFO
   MOVE.B D0,SER_THR
   BRA.S  tpsOut
tpsPutInFIFO:
  MOVE.L  (ENV_SEROBEG,A6),D3
  MOVE.L  (ENV_SEROEND,A6),D2
  MOVE.L  D2,D1
  SUB.L   D3,D2
  CMPI.L  #FIFO_SIZE,D2
  BLT.S   tpsNotFull
   TRAP    #supervisor
   BEQ.S   tpsPutInFIFO
   MOVEA.L (ENV_KERNEL,A6),A0
   BSET.B  #KS_SERINTSIGTX,(KRN_STATUS,A0)
   MOVE.L  D0,D1
   MOVE.L  #SIG_SERIAL_TX,D0
   BSR     block
   MOVE.L  D1,D0
   BRA.S   tpsPutInFIFO
tpsNotFull:
  ANDI.L  #FIFO_AND,D1
  MOVEA.L (ENV_SEROPTR,A6),A0
  MOVE.B  D0,(A0,D1)
  ADDQ.L  #1,(ENV_SEROEND,A6)
  BSR.S   serSetTxIRQ
tpsOut:
 MOVEM.L (A7)+,D0-D3/A0/A6
 RTS
-
-
serSetTxIRQ:
 BSET.B #2,SER_SPR
 MOVE.B SER_SPR,SER_LED
 BSET.B #1,SER_IER
 RTS
-
-
sendS:
 MOVEM.L D0-D3/A0/A1/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   tssOut
  TST.L   D0
  BEQ.S   tssOut
tssTryAgain:
   MOVEA.L (ENV_SEROPTR,A6),A1
   MOVE.L  (ENV_SEROEND,A6),D1
   MOVE.L  D1,D2
   SUB.L   (ENV_SEROBEG,A6),D2
   MOVEQ   #0,D3
tssMore:
    CMPI.L #FIFO_SIZE,D2
    BLT.S  tssNotFull
     ADD.L   D3,(ENV_SEROEND,A6)
     BSR.S   serSetTxIRQ
     TRAP    #supervisor
     BEQ.S   tssTryAgain
     MOVEA.L (ENV_KERNEL,A6),A1
     BSET.B  #KS_SERINTSIGTX,(KRN_STATUS,A1)
     MOVE.L  D0,D1
     MOVE.L  #SIG_SERIAL_TX,D0
     BSR     block
     MOVE.L  D1,D0
     BRA.S   tssTryAgain
tssNotFull:
    ANDI.L #FIFO_AND,D1
    MOVE.B (A0)+,(A1,D1)
    ADDQ.L #1,D1
    ADDQ.L #1,D2
    ADDQ.L #1,D3
    SUBQ.L #1,D0
   BNE.S tssMore
   ADD.L D3,(ENV_SEROEND,A6)
   BSR   serSetTxIRQ
tssOut:
 MOVEM.L (A7)+,D0-D3/A0/A1/A6
 RTS
-
-
sendA:
 MOVEM.L D2-D4/A0/A1/A6,-(A7)
 BSR     getEnvToA6
 MOVEQ   #0,D4
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   tsaOut
  TST.L  D0
  BEQ.S  tsaOut
   MOVEA.L (ENV_SEROPTR,A6),A1
   MOVE.L  (ENV_SEROEND,A6),D3
   MOVE.L  D3,D2
   SUB.L   (ENV_SEROBEG,A6),D2
tsaMore:
    CMPI.L #FIFO_SIZE,D2
    BGE.S  tsaDone
    ANDI.L #FIFO_AND,D3
    MOVE.B (A0)+,(A1,D3)
    ADDQ.L #1,D2
    ADDQ.L #1,D3
    ADDQ.L #1,D4
    SUBQ.L #1,D0
   BNE.S tsaMore
tsaDone:
  ADD.L D4,(ENV_SEROEND,A6)
  BSR   serSetTxIRQ
tsaOut:
 MOVE.L  D4,D0
 MOVEM.L (A7)+,D2-D4/A0/A1/A6
 RTS
-
-
readS:
 MOVEM.L D0-D3/A0/A1/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   trsOut
  TST.L  D0
  BEQ.S  trsOut
trsTryAgain:
   MOVEA.L (ENV_SERIPTR,A6),A1
   MOVE.L  (ENV_SERIBEG,A6),D1
   MOVE.L  (ENV_SERIEND,A6),D2
   MOVE.L  D1,D3
   SUB.L   D1,D2
   BNE.S   trsHasData
    TRAP    #supervisor
    BEQ.S   trsTryAgain
    MOVEA.L (ENV_KERNEL,A6),A1
    BSET.B  #KS_SERINTSIGRX,(KRN_STATUS,A1)
    MOVE.L  D0,D1
    MOVE.L  #SIG_SERIAL_RX,D0
    BSR     block
    MOVE.L  D1,D0
    BRA.S  trsTryAgain
```

```
trsHasData:
    ANDI.L #FIFO_AND,D3
    MOVE.B (A1,D3),(A0)+
    ADDQ.L #1,D1
    ADDQ.L #1,D3
    SUBQ.L #1,D0
    BEQ.S  trsDone
    SUBQ.L #1,D2
   BNE.S  trsHasData
   MOVE.L D1,(ENV_SERIBEG,A6)
  BRA.S trsTryAgain
trsDone:
  MOVE.L D1,(ENV_SERIBEG,A6)
trsOut:
 MOVEM.L (A7)+,D0-D3/A0/A1/A6
 RTS
-
-
getS:
 MOVEM.L A0/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   tgsNoSer
tgsTryAgain:
  MOVE.L (ENV_SERIBEG,A6),D0
  CMP.L  (ENV_SERIEND,A6),D0
  BNE.S  tgsHasData
   TRAP    #supervisor
   BEQ.S   tgsTryAgain
   MOVEA.L (ENV_KERNEL,A6),A0
   BSET.B  #KS_SERINTSIGRX,(KRN_STATUS,A0)
   MOVE.L  #SIG_SERIAL_RX,D0
   BSR     block
   BRA.S   tgsTryAgain
tgsHasData:
  MOVEA.L (ENV_SERIPTR,A6),A0
  ANDI.L  #FIFO_AND,D0
  MOVE.B  (A0,D0),D0
  ADDQ.L  #1,(ENV_SERIBEG,A6)
  BRA.S   tgsOut
tgsNoSer:
 MOVEQ #-1,D0
tgsOut:
 MOVEM.L (A7)+,A0/A6
 RTS
-
-
getA:
 MOVEM.L A0/A6,-(A7)
 BSR     getEnvToA6
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ.S   tgaOut
  MOVE.L  (ENV_SERIBEG,A6),D0
  CMP.L   (ENV_SERIEND,A6),D0
  BPL.S   tgaNoSer
   MOVEA.L (ENV_SERIPTR,A6),A0
   ANDI.L  #FIFO_AND,D0
   MOVE.B  (A0,D0),D0
   ADDQ.L  #1,(ENV_SERIBEG,A6)
   BRA.S   tgaOut
tgaNoSer:
 MOVEQ #-1,D0
tgaOut:
 MOVEM.L (A7)+,A0/A6
 RTS
-
-
int4:
 MOVEM.L A0/A5/A6,-(A7)
 BSR     getEnvToA6
 TST.L   (ENV_SRBINT4,A6)
 BEQ.S   i4NoBeforeSub
  MOVEA.L (ENV_SRBINT4,A6),A0
  JSR     (A0)
i4NoBeforeSub:
 MOVEM.L D0-D3,-(A7)
 BTST.B  #ES_SER,(ENV_STATUS,A6)
 BEQ     i4Out
i4TryAgain:
 MOVE.B  SER_ISR,D0
 ANDI.B  #7,D0
******************* Receive *****
 CMPI.B  #4,D0
 BNE     i4NotRx
  MOVEA.L (ENV_SERIPTR,A6),A0
  MOVE.L  (ENV_SERIEND,A6),D1
  MOVEA.L #SER_LSR,A5
  MOVE.B  (A5),D2
  BTST.B  #7,D2       -> an error?
  BNE.S   i4RxErr
  BTST.B  #1,D2
  BEQ.S   i4NoRxErr
i4RxErr:
   BSET.B #6,SER_SPR
   MOVE.B SER_SPR,SER_LED -> use the dipinp to clear this led-bit
i4NoRxErr:
  MOVE.L  D1,D2
  SUB.L   (ENV_SERIBEG,A6),D2 -> D2 is now fill level
  MOVEQ   #0,D0      -> cntr to add to ENV_SERIEND when done
i4MoreRx:
   CMPI.W  #FIFO_SIZE,D2 -> change this to .L for a larger FIFO_SIZE
   BLT.S   i4NotOFL
    BSET.B #7,SER_SPR
    MOVE.B SER_SPR,SER_LED
    MOVE.B #SB_FCR_RXRESET,SER_FCR -> flush RxFIFO
    BRA.S  i4WasOFL
i4NotOFL:
   ANDI.L #FIFO_AND,D1
   MOVE.B SER_RHR,(A0,D1)
   ADDQ.L #1,D0
   ADDQ.L #1,D1
   ADDQ.L #1,D2
   BTST.B #0,(A5)
  BNE.S  i4MoreRx
i4WasOFL:
  ADD.L   D0,(ENV_SERIEND,A6)
-
  BTST.B  #ES_KERNEL,(ENV_STATUS,A6)
  BEQ.S   i4RxNoKrn
   MOVEA.L (ENV_KERNEL,A6),A5
   MOVE.B  (KRN_STATUS,A5),D1
   BTST.B  #KS_SERINTSIGRX,D1
   BEQ.S   i4RxNoKrn
    ANDI.B  #%1011,D1 -> are we inhibited?
    BSET.B  #KS_INHIBIT,(KRN_STATUS,A5) ->this will get cleared by the intCatcher or whoever already set it.
    MOVEA.L (ENV_PCB,A6),A0
i4RxKrnLoop:
    BTST.B  #SIG_SERIAL_RX-24,(PCB_SIGWAIT,A0) ->is SIG_SERIAL_RX set?
    BEQ.S   i4RxNoSig
     BSET.B  #SIG_SERIAL_RX-24,(PCB_SIGSET,A0)
     MOVE.B  #PS_READY,(PCB_STATUS,A0)
     TST.B   D1
     BNE.S   i4RxNoResch
      MOVEA.L (KRN_SSPREF,A5),A0  ->intercept the last supervisor return
      MOVE.L  -(A0),(KRN_TOPPC,A5)
      MOVE.W  -(A0),(KRN_TOPSR,A5)
      LEA     (intCatcher,PC),A5
      MOVE.W  #$2300,(A0)+
      MOVE.L  A5,(A0)
      MOVEQ   #1,D1
      BRA.S   i4RxNoSig
i4RxNoResch:
     BSET.B #KS_SIGNAL,(KRN_STATUS,A5) ->signal an eventual STOP
i4RxNoSig:
    MOVEA.L (A0),A0 ->pcb.next
    CMPA.L  #0,A0
   BNE.S i4RxKrnLoop
i4RxNoKrn:
-
  TST.L   (ENV_SRARX,A6)
  BEQ.S   i4NoRxSub
   MOVEA.L (ENV_SRARX,A6),A0
   JSR     (A0)
i4NoRxSub:
  BRA   i4TryAgain
i4NotRx:
******************* Transmit *****
 CMPI.B #2,D0
 BNE    i4NotTx
  BCLR.B  #2,SER_SPR
  MOVE.B  SER_SPR,SER_LED
  BCLR.B  #1,SER_IER
  MOVE.L  (ENV_SEROBEG,A6),D0
  MOVE.L  (ENV_SEROEND,A6),D2
  MOVEA.L #SER_THR,A5
  SUB.L   D0,D2
  BLS.S   i4TxOut
   MOVEA.L (ENV_SEROPTR,A6),A0
   MOVEQ   #15,D1
i4MoreTx:
    ANDI.L #FIFO_AND,D0
    MOVE.B (A0,D0),(A5) ->SER_THR
    ADDQ.L #1,D0
    SUBQ.L #1,D2
    BEQ.S  i4TxDone
   DBRA   D1,i4MoreTx
   ADDI.L #16,(ENV_SEROBEG,A6)
   BSR     serSetTxIRQ
   BRA.S  i4TxOut
i4TxDone:
   MOVE.L (ENV_SEROEND,A6),(ENV_SEROBEG,A6)
i4TxOut:
-
  BTST.B  #ES_KERNEL,(ENV_STATUS,A6)
  BEQ.S   i4TxNoKrn
   MOVEA.L (ENV_KERNEL,A6),A5
   MOVE.B  (KRN_STATUS,A5),D1
   BTST.B  #KS_SERINTSIGTX,D1
   BEQ.S   i4TxNoKrn
    ANDI.B  #%1011,D1 -> are we inhibited?
    BSET.B  #KS_INHIBIT,(KRN_STATUS,A5) ->this will get cleared by the intCatcher or whoever already set it.
    MOVEA.L (ENV_PCB,A6),A0
i4TxKrnLoop:
    BTST.B  #SIG_SERIAL_TX-24,(PCB_SIGWAIT,A0) ->is SIG_SERIAL_TX set?
    BEQ.S   i4TxNoSig
     BSET.B #SIG_SERIAL_TX-24,(PCB_SIGSET,A0)
     MOVE.B #PS_READY,(PCB_STATUS,A0)
     TST.B  D1
     BNE.S  i4TxNoResch
      MOVEA.L (KRN_SSPREF,A5),A0  ->intercept the last supervisor return
      MOVE.L  -(A0),(KRN_TOPPC,A5)
      MOVE.W  -(A0),(KRN_TOPSR,A5)
      LEA     (intCatcher,PC),A5
      MOVE.W  #$2300,(A0)+
      MOVE.L  A5,(A0)
      MOVEQ   #1,D1
      BRA.S   i4TxNoSig
i4TxNoResch:
     BSET.B #KS_SIGNAL,(KRN_STATUS,A5) ->signal an eventual STOP
i4TxNoSig:
    MOVEA.L (A0),A0 ->pcb.next
    CMPA.L  #0,A0
   BNE.S i4TxKrnLoop
i4TxNoKrn:
-
  TST.L   (ENV_SRATX,A6)
  BEQ.S   i4NoTxSub
   MOVEA.L (ENV_SRATX,A6),A0
   JSR     (A0)
i4NoTxSub:
  BRA   i4TryAgain
i4NotTx:
******************* Line and Modem *****
 CMPI.B #6,D0
 BNE.S  i4NotLSR
  MOVEA.L (ENV_SRALINE,A6),A0
  CMPA.L  #0,A0
  BEQ.S   i4NoLSRSub
   JSR    (A0)
i4NoLSRSub:
  BRA   i4TryAgain
i4NotLSR:
 CMPI.B #0,D0
 BNE.S  i4Out
  TST.L   (ENV_SRAMODEM,A6)
  BEQ.S   i4NoMSRSub
   MOVEA.L (ENV_SRAMODEM,A6),A0
   JSR    (A0)
i4NoMSRSub:
  BRA   i4TryAgain
i4Out:
 MOVEM.L (A7)+,D0-D3
 TST.L   (ENV_SRAINT4,A6)
 BEQ.S   i4NoAfterSub
  MOVEA.L (ENV_SRAINT4,A6),A0
  JSR     (A0)
i4NoAfterSub:
 MOVEM.L (A7)+,A0/A5/A6
 RTE
-
-
welcomeSerTxt:
 DC "Welcome to ExOS v0.2\n"
 DC "This version is designed for MC68000, 128kB ROM, 256kB RAM.\n\n"
 DC "R\0M is now located at address $0 where we are now executing.\n"
 DC "The memory manager is currently \0running.\n"
 DC "The CPU is running at speed \0\n"
 DC "You will be sent to the \0 console where you can enter ? for help.\n\n\0"
-
wstAdvanced:  DC "advanced\0"
wstPrimitive: DC "primitive\0"
 ALIGN
-
welcomeSerial:
 BSR    getEnvToA6
 LEA    (welcomeSerTxt,PC),A0
 BSR.S  wsPutStr
 MOVE.B #'O',D0
 BTST.B #ES_SWAP,(ENV_STATUS,A6)
 BEQ.S  wsInROM
  MOVE.B #'A',D0
wsInROM:
 BSR    putS
 BSR.S  wsPutStr
 TST.L  (ENV_FREEL,A6)
 BNE.S  wsNoMH1
  MOVE.B #'n',D0
  BSR    putS
  MOVE.B #'o',D0
  BSR    putS
  MOVE.B #'t',D0
  BSR    putS
  MOVE.B #' ',D0
  BSR    putS
wsNoMH1:
 BSR.S   wsPutStr
 MOVE.B  (ENV_MCR,A6),D0
 LSR.B   #5,D0
 ADDI.B  #'1',D0
 BSR     putS
 BSR.S   wsPutStr
 MOVEA.L A0,A1
 TST.L   (ENV_FREEL,A6)
 BEQ.S   wsIsMH2
  LEA    (wstAdvanced,PC),A0
  BRA.S  wsNoMH2
wsIsMH2:
  LEA    (wstPrimitive,PC),A0
wsNoMH2:
 BSR.S   wsPutStr
 MOVEA.L A1,A0
 BSR.S   wsPutStr
 RTS
-
wsPutStr:
 MOVE.B (A0)+,D0
 BEQ.S  wsOut
  BSR   putS
 BRA.S  wsPutStr
wsOut:
 RTS
```

F.7 ProgInlasning.s68

This file, along with the next file: ”S19convert.s68”, contains the function StoreProg which decodes S19-records into binary.

```
* >A0.L = Adress till lnkad lista
* A0.L> = Adress till programminne
* D0.L> = Programstorlek, 0 = out of memory,
*         -1 = Checksum ERROR, -2 = No data
storeProg:
    MOVEM.L D1-D3/A2-A4,-(A7)
    MOVEA.L A0,A3
    BSR         progSize
    TST.L   D0
    BEQ.S   spNoData
    TRAP    #allocmem
    CMPA.L  #0,A0
    BEQ.S   spNoMem
    MOVEA.L A0,A2 // Konvertera och lgg in i minne, adress i A2
    MOVE.L  D0,-(A7)
    BSR     convert
    MOVE.L (A7)+,D1
    CMPI.B  #1,D0       // Check if OK
    BEQ.S       spDone
* Something went wrong, return memory.
spError:
    MOVEQ   #-1,D0
    MOVEA.L A2,A0
    TRAP   #freemem
    MOVEA.L #0,A0
    BRA.S   spOut
spNoData:
    MOVEQ   #-2,D0
    BRA.S   spOut
spNoMem:
    MOVEQ   #0,D0
    BRA.S   spOut
* Returnera pekare till program A2>
spDone:
    MOVEA.L A2,A0
    MOVE.L  D1,D0
spOut:
    MOVEM.L (A7)+,D1-D3/A2-A4
    RTS
-
*
* Get program size
*
progSize:
    MOVEA.L A3,A4       // Spara adress fr senare bruk
    MOVE.L  #0,D3       // Nollstll storlek
* Ls en post i den lnkade listan ( adress, next )
psLoop:
    MOVEA.L (A3),A0     // Lgg adressen till strngen i A0
    BSR     ps_getLineSize
    ADD.L   D2,D3       // ka storleken
* Slut p listan
    CMPI.L  #0,(4,A3)
    BEQ     psDone
* Nej, ta nsta post
    ADDA.L  #LL_EL_SIZE,A3
    BRA     psLoop
* Ja, returnera
psDone:
    MOVEA.L A4,A3
    MOVE.L  D3,D0
    RTS
-
ps_getLineSize:
* Ls typ ( 0x53xx )
    MOVE.W  (A0)+,D1    // och lgg typ i D1
    BSR     doByte      // Hmta byte
    MOVE.L  D0,D2       // och lgg lngd i D2
    ADDA.L  #$2,A0      // Flytta fram till adress
-
* Ls adress med lngd beroende av typ
ps_s1:
    CMPI.W  #$5331,D1       // S1
    BNE     ps_s2
    SUBI.L  #3,D2
    RTS
ps_s2:
    CMPI.W  #$5332,D1       // S2
    BNE     ps_s3
    SUBI.L  #4,D2
    RTS
ps_s3:
    CMPI.W  #$5333,D1       // S3
    BNE     ps_default
    SUBI.L  #5,D2
    RTS
ps_default:
    MOVE.L  #0,D2
    RTS
-
    USE s19convert.s68
```

F.8 S19convert.s68

```
*BASE   EQU     $4200
*LEDS  EQU      $20000B
-
*LL_EL_SIZE EQU 8   // BYTES
-
* >A2 = basadressen till programminne
* >A3 = adress till lnkad lista
* D0> = 1 = OK, 0 = Not OK
convert:
    MOVEA.L A3,A4       // Spara adress fr senare bruk
* Ls en post i den lnkade listan ( adress, next )
convLoop:
    MOVEA.L (A3),A0     // Lgg adressen till strngen i A0
* Konvertera posten och lgg in i minnet
    BSR     convLine
    CMPI.B  #1,D0           // Check if OK
    BNE.S   convDone
* Slut p listan
    CMPI.L  #0,(4,A3)
    BEQ     convDone
* Nej, ta nsta post
    ADDA.L  #LL_EL_SIZE,A3
    BRA     convLoop
* Ja, returnera
convDone:
    MOVEA.L A4,A3
    RTS
-
* >A2 = basadressen till programminne
* >A0 = adressen till strngen
convLine:
* Kontrollera checksumma
    MOVEM.L A0,-(SP)    // Spara adressen p stacken
    ADDA.L  #$2,A0      // Hoppa ver typen
    BSR     checkChecksum   // Kontrollera checksumman
    MOVEM.L (SP)+,A0    // Hmta adressen frn stacken
    CMPI.B  #$1,D0
    BNE     convLineDone
* Ls typ ( 0x53xx )
    MOVE.W  (A0)+,D1    // och lgg typ i D1
    BSR     doByte      // Hmta byte
    MOVE.L  D0,D2       // och lgg lngd i D2
    ADDA.L  #$2,A0      // Flytta fram till adress
-
* Ls adress med lngd beroende av typ
    CMPI.W  #$5330,D1   // S0
    BNE     s1
* NGOT SKA KANSKE GRAS HR
    MOVE.B  #1,D0           // Allt OK
    BRA     convLineDone
s1:
    CMPI.W  #$5331,D1   // S1
    BNE     s2
    SUBI.L  #3,D2
    MOVE.L  #$4,D3      // Stt rknare till 4
    BSR     doAdress
    ADDA.L  #$4,A0
    BRA     storeData   // Hoppa till gemensam kod
s2:
    CMPI.W  #$5332,D1   // S2
    BNE     s3
    SUBI.L  #4,D2
    MOVE.L  #$6,D3      // Stt rknare till 6
    BSR     doAdress
    ADDA.L  #$6,A0
    BRA     storeData   // Hoppa till gemensam kod
s3:
    CMPI.W  #$5333,D1   // S3
    BNE     s4
    SUBI.L  #5,D2
    MOVE.L  #$8,D3      // Stt rknare till 8
    BSR     doAdress
    ADDA.L  #$8,A0
    BRA     storeData   // Hoppa till gemensam kod
s4:
    CMPI.W  #$5334,D1   // S4
    BNE     s5
    BRA     storeData   // Hoppa till gemensam kod
s5:
    CMPI.W  #$5335,D1   // S5
    BNE     s7
    BRA     storeData   // Hoppa till gemensam kod
s7:
    CMPI.W  #$5337,D1   // S7
    BNE     s8
    MOVE.L  #$8,D3      // Stt rknare till 8
    BSR     doAdress
    ADDA.L  #$8,A0
    BRA     endRecords  // Hoppa till gemensam kod
```

```
s8:
    CMPI.W  #$5338,D1   // S8
    BNE     s9
    MOVE.L  #$6,D3      // Stt rknare till 6
    BSR     doAdress
    ADDA.L  #$6,A0
    BRA     endRecords  // Hoppa till gemensam kod
s9:
    CMPI.W  #$5339,D1   // S9
    BNE     convLineDone
    MOVE.L  #$4,D3      // Stt rknare till 4
    BSR     doAdress
    ADDA.L  #$4,A0
    BRA     endRecords  // Hoppa till gemensam kod
storeData:
* Ls data och lgg in med brjan p adress
    BSR     doByte
    MOVE.B  d0,(0,A1,A2)    // Flytta data till adress+basadress
    ADDA.L  #$1,A1
    ADDA.L  #$2,A0
    SUBI.B  #$1,D2
    BNE     storeData
    MOVE.B  #1,D0           // Allt OK
    BRA     convLineDone
endRecords:
* Vad ska gras hr. Lite olika i txt-fil och exempel
    MOVE.B  #1,D0           // Allt OK
    BRA     convLineDone
convLineDone:
    RTS
-
-
* doAdress:
* PVERKAR: D3
* PRE:  D3 innehller adresslngd (4,6,8 tecken)
* POST: A1 innehller adressen D3 = 0
doAdress:
    MOVEM.L A0/D0/D1,-(SP)  // Spara pverkade p stacken
    MOVE.L  #0,D1           // Rensa address
doA_loop:
    MOVE.B  (A0)+,D0
    BSR     asciiToHex
    ASL.L   #$4,D1
    ADD.L   D0,D1
    SUBI.L  #$1,D3
    BNE     doA_loop
    MOVEA.L D1,A1
    MOVEM.L (SP)+,A0/D0/D1  // Spara pverkade p stacken
    RTS
-
-
* asciiToHex:
* PVERKAR: D0
* PRE:  D0 innehller ASCII-vrde
* POST: D0 innehller HEX-vrde
asciiToHex:
    CMPI.B  #65,D0
    BGE     ath1
    SUBI.B  #48,D0      // Omvandla frn ASCII till siffra
    BRA     ath_end
ath1:
    SUBI.B  #55,D0      // Omvandla frn ASCII till siffra
ath_end:
    RTS
-
-
* doByte:
* PVERKAR: D0
* PRE:  A0 innehller adressen till frsta asciitecknet
* POST: D0 innehller byte
doByte:
    MOVEM.L A0/D1,-(SP)     // Spara pverkade p stacken
    MOVE.L  #$0,D0      // Nollstll D0
    MOVE.L  #$0,D1      // Nollstll D1
    MOVE.B  (A0)+,D0    // Lgg frsta byten i D0
    CMPI.B  #65,D0
    BGE     alpha1
    SUBI.B  #48,D0      // Omvandla frn ASCII till siffra
    BRA     multi1
alpha1:
    SUBI.B  #55,D0      // Omvandla frn ASCII till siffra
multi1:
    MULU    #$10,D0     // Multiplicera med 16; Hg del av byte
    MOVE.B  (A0),D1     // Lgg andra byten i D1
    CMPI.B  #65,D1
    BGE     alpha2
    SUBI.B  #48,D1      // Omvandla frn ASCII till siffra
    BRA     add
alpha2:
    SUBI.B  #55,D1      // Omvandla frn ASCII till siffra
add:
    ADD.B   D1,D0           // Skapa byte i D0
    MOVEM.L (SP)+,A0/D1     // Spara pverkade p stacken
    RTS
* checkChecksum:
* Kontrollerar checksumman genom att ta 8-bitars 1-komp.
* p summan av lngd, adress, data
* PVERKAR: D0, D1, D2, A0
* PRE: A0 innehller adressen till strngen
* POST: D0 innehller svaret 0=false 1=true
checkChecksum:
* Ta reda p lngden och lgg i D0
    BSR     doByte      // Omvandla de tv frsta byten till lngd i D0
-
* Summera ihop lngd, adress och data och lgg i D2
```

```
    MOVE.L  D0,D1       // Flytta lngd till D1
    MOVE.L  #0,D2       // Nollstll summa
chk_loop:
    BSR     doByte      // Omvandla de tv frsta byten till och lggs i D0
-
    ADD.L   D0,D2       // Summera
    ADDA.L  #$2,A0      // Flytta fram tv bytes
    SUBI.L  #$1,D1      // Minska lngd med 1
    BNE     chk_loop
    BSR     doByte      // Omvandla de tv frsta byten till lngd i D0
    NOT.B   D2          // 1-komplement
    CMP.B   D0,D2
    BNE     ret0
    MOVE.L  #$1,D0
    RTS
ret0:
    MOVE.L  #$0,D0
    RTS
```

F.9 Interrupts.s68

This file contains all interrupt routines not assigned to anything special, like the scheduler or the serial. It also contains all exception handlers for address error, division by zero, illegal instruction, etc. A bit of scheduling code is also included to be able to safely crash any user program that caused an exception. Interrupt routines 1 and 7, which are directly connected to switches on the main board, are able to read the DIP-switch and execute simple commands while debugging.

```
*LED outputs:
*BusErr    =82 -> 1000 0010
*AddrErr   =83 -> 1000 0011
*Illegal   =84 -> 1000 0100
*DivBy0    =85 -> 1000 0101
*CHK       =86 -> 1000 0110
*TrapV     =87 -> 1000 0111
*PrivViol  =88 -> 1000 1000
*Trace     =89 -> 1000 1001
*LineA     =8A -> 1000 1010
*LineF     =8B -> 1000 1011
*Reserved  =8C -> 1000 1100
*Spurious  =8D -> 1000 1101
*Uninit Trap=8E-> 1000 1110
*UserInt   =8F -> 1000 1111
*int1      =F9 -> 1111 1001
*int2      =FA -> 1111 1010
*int3      =FB -> 1111 1011
*int4      =FC -> 1111 1100
*int5      =FD -> 1111 1101
*int6      =FE -> 1111 1110
*int7      =FF -> 1111 1111
-
-
*********** Autovector interrupts (except int4=serial):
-
**** Secondary DIPINP interrupt
int1:
 MOVEM.L A0/A6,-(A7)
 BSR     getEnvToA6
 TST.L   (ENV_SRBINT1,A6)
 BEQ.S   i1NoBeforeSub
  MOVEA.L (ENV_SRBINT1,A6),A0
  JSR     (A0)
i1NoBeforeSub:
 MOVEM.L D0-D2,-(A7)
 MOVE.B  #$F9,LED
 MOVE.B  (ENV_INT7,A6),D0
 BEQ     i1NotQuiteOut
  CMPI.B #$80,D0            ->Just output the DIPs to the LED
  BNE.S  i1try1
   MOVE.B DIP,LED
   MOVE.B DIP,(ENV_LED,A6)
   BRA    i1out
i1try1
  CMPI.B #$81,D0            ->Set LF clock
  BNE.S  i1try2
   MOVE.B DIP,LFCLK
   BRA    i1out
i1try2:
  CMPI.B #$82,D0            ->Set serial bitrate
  BNE.S  i1try3
   MOVE.B #%10000011,SER_LCR
   MOVE.B SER_DLL,LED
   MOVE.B #0,SER_DLM
   MOVE.B DIP,SER_DLL
   MOVE.B #%00000011,SER_LCR
   BRA.S  i1out
i1try3:
  CMPI.B #$83,D0             ->Output DIPs to serial
  BNE.S  i1try4
   MOVE.B DIP,SER_THR
   BRA.S  i1out
i1try4:
  CMPI.B #$84,D0             ->Read serial to LEDs
  BNE.S  i1try4
i14NoData:
    BTST.B #0,SER_LSR
   BEQ.S  i14NoData
   MOVE.B SER_RHR,LED
   BRA.S  i1out
i1NotQuiteOut:
 CMPI.B #$FF,DIP
 BNE.S  i1out
  MOVE.B (ENV_LED,A6),LED
  SUBI.B #1,(ENV_LED,A6)
  MOVE.L #1000,D2
i1Delay:
   MOVE.L #$5555,D0
   MOVE.L #12345678,D1
   DIVS   D1,D0
  DBRA  D2,i1Delay
  BRA.S i1OverOut
i1out:
  BCLR.B #ES_LFLED,(ENV_STATUS,A6)
i1OverOut:
 MOVEM.L (A7)+,D0-D2
 TST.L   (ENV_SRAINT1,A6)
 BEQ.S   i1NoAfterSub
  MOVEA.L (ENV_SRAINT1,A6),A0
  JSR     (A0)
i1NoAfterSub:
 MOVEM.L (A7)+,A0/A6
 RTE
-
**** Slot-bus interrupt
int2:
 MOVEM.L A0/A6,-(A7)
 BSR     getEnvToA6
 TST.L   (ENV_SRBINT2,A6)
 BEQ.S   i2NoBeforeSub
  MOVEA.L (ENV_SRBINT2,A6),A0
  JSR     (A0)
i2NoBeforeSub:
 BTST.B  #ES_LFLED,(ENV_STATUS,A6)
 BNE.S   i2out
  MOVE.B #$FA,LED
i2out:
 TST.L   (ENV_SRAINT2,A6)
 BEQ.S   i2NoAfterSub
  MOVEA.L (ENV_SRAINT2,A6),A0
  JSR     (A0)
i2NoAfterSub:
 MOVEM.L (A7)+,A0/A6
 RTE
-
*int3 is in kernel.s68
*int4 is in serial.s68
-
**** I/O-bus interrupt
int5:
 MOVEM.L A0/A6,-(A7)
 BSR     getEnvToA6
 TST.L   (ENV_SRBINT5,A6)
 BEQ.S   i5NoBeforeSub
  MOVEA.L (ENV_SRBINT5,A6),A0
  JSR     (A0)
i5NoBeforeSub:
 BTST.B  #ES_LFLED,(ENV_STATUS,A6)
 BNE.S   i5out
  MOVE.B #$FD,LED
i5out:
 TST.L   (ENV_SRAINT5,A6)
 BEQ.S   i5NoAfterSub
  MOVEA.L (ENV_SRAINT5,A6),A0
  JSR     (A0)
i5NoAfterSub:
 MOVEM.L (A7)+,A0/A6
 RTE
-
**** Slot-bus interrupt
int6:
 MOVEM.L A0/A6,-(A7)
 BSR     getEnvToA6
 TST.L   (ENV_SRBINT6,A6)
 BEQ.S   i6NoBeforeSub
  MOVEA.L (ENV_SRBINT6,A6),A0
  JSR     (A0)
i6NoBeforeSub:
 BTST.B  #ES_LFLED,(ENV_STATUS,A6)
 BNE.S   i6out
  MOVE.B #$FE,LED
i6out:
 TST.L   (ENV_SRAINT6,A6)
 BEQ.S   i6NoAfterSub
  MOVEA.L (ENV_SRAINT6,A6),A0
  JSR     (A0)
i6NoAfterSub:
 MOVEM.L (A7)+,A0/A6
 RTE
**** Primary DIPINP interrupt
int7:
 MOVEM.L A0/A6,-(A7)
 BSR     getEnvToA6
 TST.L   (ENV_SRBINT7,A6)
 BEQ.S   i7NoBeforeSub
  MOVEA.L (ENV_SRBINT7,A6),A0
  JSR     (A0)
i7NoBeforeSub:
 MOVEM.L D0/D1,-(A7)
 MOVE.B  #$FF,LED
 BSR     getEnvToA6
 MOVE.B  DIP,D0
 BTST.B  #7,D0
 BEQ.S   i7tryNormal
  MOVE.B D0,(ENV_INT7,A6)
  BRA    i7out
i7tryNormal
 MOVE.B  D0,D1
 LSR.B   #4,D1
 ANDI.B  #$F,D0
 CMPI.B  #%0000,D1
 BNE.S   i7try1
* Do nothing
  BRA     i7out
i7try1:
 CMPI.B  #%0001,D1 ->Remove interrupt subroutines
 BNE     i7try2
  CMPI.B #0,D0
  BNE.S  i71Not0
   LEA   (ENV_SRBINT1,A6),A0
   MOVEQ #13,D0
i71ClearAll:
    MOVE.L #0,(A0)+
   DBRA D0,i71ClearAll
   BRA   i7out
i71Not0:
  CMPI.B #8,D0
  BNE.S  i71Not8
   MOVE.L #0,(ENV_SRARX,A6)
   MOVE.L #0,(ENV_SRATX,A6)
   MOVE.L #0,(ENV_SRALINE,A6)
   MOVE.L #0,(ENV_SRAMODEM,A6)
   BRA   i7out
i71Not8:
  CMPI.B #1,D0
  BNE.S  i71Not1
   MOVE.L #0,(ENV_SRBINT1,A6)
   BRA   i7out
i71Not1:
  CMPI.B #2,D0
  BNE.S  i71Not2
   MOVE.L #0,(ENV_SRBINT2,A6)
   BRA   i7out
i71Not2:
  CMPI.B #3,D0
  BNE.S  i71Not3
   MOVE.L #0,(ENV_SRBINT3,A6)
   BRA   i7out
i71Not3:
  CMPI.B #4,D0
  BNE.S  i71Not4
   MOVE.L #0,(ENV_SRBINT4,A6)
   BRA   i7out
i71Not4:
  CMPI.B #5,D0
  BNE.S  i71Not5
   MOVE.L #0,(ENV_SRBINT5,A6)
   BRA   i7out
i71Not5:
  CMPI.B #6,D0
  BNE.S  i71Not6
   MOVE.L #0,(ENV_SRBINT6,A6)
   BRA   i7out
i71Not6:
  CMPI.B #7,D0
  BNE.S  i71Not7
   MOVE.L #0,(ENV_SRBINT7,A6)
   BRA   i7out
i71Not7:
  CMPI.B #$9,D0
  BNE.S  i71Not9
   MOVE.L #0,(ENV_SRAINT1,A6)
   BRA   i7out
i71Not9:
  CMPI.B #$A,D0
  BNE.S  i71NotA
   MOVE.L #0,(ENV_SRAINT2,A6)
   BRA   i7out
i71NotA:
  CMPI.B #$B,D0
  BNE.S  i71NotB
   MOVE.L #0,(ENV_SRAINT3,A6)
   BRA   i7out
i71NotB:
  CMPI.B #$C,D0
  BNE.S  i71NotC
   MOVE.L #0,(ENV_SRAINT4,A6)
   BRA   i7out
i71NotC:
  CMPI.B #$D,D0
  BNE.S  i71NotD
   MOVE.L #0,(ENV_SRAINT5,A6)
   BRA   i7out
i71NotD:
  CMPI.B #$E,D0
  BNE.S  i71NotE
   MOVE.L #0,(ENV_SRAINT6,A6)
   BRA   i7out
i71NotE:
  CMPI.B #$F,D0
  BNE.S  i71NotF
   MOVE.L #0,(ENV_SRAINT7,A6)
   BRA   i7out
i71NotF:
  BRA    i7out
i7try2:
 CMPI.B  #%0010,D1 ->Set serial IRQ (write DATA to SER_IER)
 BNE.S   i7try3
  MOVE.B D0,SER_IER
  BTST.B #0,D0
  BEQ.S  i72NoRx
   BSET.B #0,SER_SPR
   BRA.S  i72RxOut
i72NoRx:
   BCLR.B #0,SER_SPR
i72RxOut:
  BTST.B #1,D0
  BEQ.S  i72NoTx
   BSET.B #2,SER_SPR
   BRA.S  i72TxOut
i72NoTx:
   BCLR.B #2,SER_SPR
i72TxOut:
  MOVE.B SER_SPR,SER_LED
  BRA    i7out
i7try3:
 CMPI.B  #%0011,D1 ->Execute STOP
 BNE.S   i7try4
  STOP   #$2000
  BRA    i7out
i7try4:
 CMPI.B  #%0100,D1 ->Read env on LED (0000=MCR, 0001=Stat, 0010=RAM lock, 0100=Rx lock, 0101=Tx lock)
 BNE.S   i7try5
  CMPI.B #0,D0
  BNE.S  i74NotMCR
   MOVE.B (ENV_MCR,A6),LED
   BRA.S  i74Out
i74NotMCR:
  CMPI.L #1,D0
  BNE.S  i74NotStat
   MOVE.B (ENV_STATUS,A6),LED
   BRA.S  i74Out
i74NotStat:
  CMPI.L #2,D0
  BNE.S  i74NotRAM
   MOVE.B (ENV_RAMLOCK,A6),LED
   BRA.S  i74Out
i74NotRAM:
  CMPI.L #4,D0
  BNE.S  i74NotRxLock
   MOVE.B (ENV_RXLOCK,A6),LED
   BRA.S  i74Out
i74NotRxLock
  CMPI.L #5,D0
  BNE.S  i7out
   MOVE.B (ENV_TXLOCK,A6),LED
i74Out:
  BCLR.B #ES_LFLED,(ENV_STATUS,A6)
  BRA.S   i7out
i7try5:
 CMPI.B  #%0101,D1 ->Connect LF-clock to LED (xxx1=connect, xxx0=disc.)
 BNE.S   i7try6
  BTST.B #0,D0
  BEQ.S  i7lfLedOff
   MOVE.B #0,(ENV_LED,A6)
   BSET.B #ES_LFLED,(ENV_STATUS,A6)
   BRA.S  i7out
i7lfLedOff:
   BCLR.B #ES_LFLED,(ENV_STATUS,A6)
  BRA.S   i7out
i7try6:
 CMPI.B  #%0110,D1 ->Write DATA to MCR LSN
 BNE.S   i7try7
  MOVE.B (A6),D1
  MOVE.B D1,LED
  ANDI.B #$F0,D1
  OR.B   D0,D1
  MOVE.B D1,(A6)
  MOVE.B D1,MCR
  BRA.S   i7out
i7try7:
 CMPI.B  #%0111,D1 ->Write DATA to MCR MSN
 BNE.S   i7out
  MOVE.B (A6),D1
  MOVE.B D1,LED
  ANDI.B #$F,D1
  LSL.B  #4,D0
  OR.B   D0,D1
  MOVE.B D1,(A6)
  MOVE.B D1,MCR
i7out:
 MOVEM.L (A7)+,D0/D1
 TST.L   (ENV_SRAINT7,A6)
 BEQ.S   i7NoAfterSub
  MOVEA.L (ENV_SRAINT7,A6),A0
  JSR     (A0)
i7NoAfterSub:
 MOVEM.L (A7)+,A0/A6
 RTE
```

```
*********** Exceptions:
-
msgBusErr:
 DC.L 14
 DC "Bus error at $\0"
 ALIGN
msgAddrErr:
 DC.L 18
 DC "Address error at $\0"
 ALIGN
msgIllegal:
 DC.L 24
 DC "Illegal instruction at $\0"
 ALIGN
msgDivBy0:
 DC.L 21
 DC "Division by zero at $\0"
 ALIGN
msgCHK:
 DC.L 27
 DC "CHK instruction failed at $\0"
 ALIGN
msgTrapV:
 DC.L 18
 DC "Trap oVerflow at $\0"
 ALIGN
msgPrivViol:
 DC.L 24
 DC "Privilige violation at $\0"
 ALIGN
msgTrace:
 DC.L 20
 DC "Trace exception at $\0"
 ALIGN
msgLineA:
 DC.L 31
 DC "MMU trap, line-A exception at $\0"
 ALIGN
msgLineF:
 DC.L 31
 DC "FPU trap, line-F exception at $\0"
 ALIGN
msgSupVisEnd:
 DC.L 66+30
 DC "The CPU was in Supervisor mode and will now permit only interrupts"
 DC "unless a crash bucked exists.\n\0"
 ALIGN
-
BusErr:
 MOVE.B  #$82,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgBusErr,PC),A0
 MOVE.L  #$10002,D0
 JMP     (excCommonEnd,PC)
-
AddrErr:
 MOVE.B  #$83,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgAddrErr,PC),A0
 MOVE.L  #$10003,D0
 JMP     (excCommonEnd,PC)
-
Illegal:
 MOVE.B  #$84,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgIllegal,PC),A0
 MOVEQ   #4,D0
 JMP     (excCommonEnd,PC)
-
DivBy0:
 MOVE.B  #$85,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgDivBy0,PC),A0
 MOVEQ   #5,D0
 JMP     (excCommonEnd,PC)
-
CHK:
 MOVE.B  #$86,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgCHK,PC),A0
 MOVEQ   #6,D0
 JMP     (excCommonEnd,PC)
-
TrapV:
 MOVE.B  #$87,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgTrapV,PC),A0
 MOVEQ   #7,D0
 JMP     (excCommonEnd,PC)
-
PrivViol:
 MOVE.B  #$88,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgPrivViol,PC),A0
 MOVEQ   #8,D0
 JMP     (excCommonEnd,PC)
-
Trace:
 MOVE.B  #$89,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgTrace,PC),A0
 MOVEQ   #9,D0
 JMP     (excCommonEnd,PC)
```

```
-
LineA:
 MOVE.B  #$8A,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgLineA,PC),A0
 MOVE.L  #$1000A,D0
 JMP     (excCommonEnd,PC)
-
LineF:
 MOVE.B  #$8B,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgLineF,PC),A0
 MOVE.L  #$1000B,D0
 JMP     (excCommonEnd,PC)
-
*********** Uninitialited exceptions:
-
msgReserved:
 DC.L 20
 DC "Reserved exception\n\0"
 ALIGN
msgSpurious:
 DC.L 20
 DC "Spurious exception\n\0"
 ALIGN
msgTrap:
 DC.L 30
 DC "Uninitialized TRAP exception\n\0"
 ALIGN
msgUserInt:
 DC.L 29
 DC "Uninitialized user interrupt\n\0"
-
res:
 MOVE.B  #$8C,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgReserved,PC),A0
 MOVEQ   #$81,D0
 JMP     (excCommonEnd,PC)
-
spur:
 MOVE.B  #$8D,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgSpurious,PC),A0
 MOVEQ   #$82,D0
 JMP     (excCommonEnd,PC)
-
trap:
 MOVE.B  #$8E,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgTrap,PC),A0
 MOVEQ   #$83,D0
 JMP     (excCommonEnd,PC)
-
ui:
 MOVE.B  #$8F,LED
 MOVEM.L D0/A0/A5-A6,-(A7)
 LEA     (msgUserInt,PC),A0
 MOVEQ   #$84,D0
 JMP     (excCommonEnd,PC)
-
msgCausedBy:
 DC.L 9
 DC "Process: \0"
-
excCommonEnd:
 BSR     getEnvToA6
 TST.L   (ENV_SREXCEPT,A6)
 BEQ.S   ecxNoExcSub
  MOVEA.L (ENV_SREXCEPT,A6),A5
  MOVE.W  D0,-(A7)
  JSR     (A5)
  ADDQ.L  #2,A7
  BRA     ecxOut
ecxNoExcSub:
 MOVEA.L D0,A5
 MOVE.L  (A0)+,D0
 BSR     sendS
 SUBA.L  A0,A0
 MOVE.L  (18,A7),D0      -> STACK!
 BSR     int2Hex
 MOVEQ   #10,D0
 BSR     putS
 BTST.B  #5,(16,A7)      -> STACK!
 BNE     ecxSupVis
  BTST.B  #ES_KERNEL,(ENV_STATUS,A6)
  BEQ     ecxStop
   MOVE.L  A5,D0
   MOVEA.L (ENV_KERNEL,A6),A5
   BSET.B  #KS_INHIBIT,(KRN_STATUS,A5)
   MOVEA.L (A5),A0
   ORI.B   #PS_CRASHED,D0
   MOVE.B  D0,(PCB_STATUS,A0)
   LEA     (msgCausedBy,PC),A0
   MOVE.L  (A0)+,D0
   BSR     sendS
   MOVEA.L (A5),A0
   MOVEA.L (PCB_NAME,A0),A0
   BSR     putStr
   MOVEQ   #10,D0
   BSR     putS
   MOVEM.L (A7)+,D0/A0       -> STACK!
   MOVEM.L D0-D7/A0-A4,-(A7) -> STACK!
   LEA     (ecxBack,PC),A0
   MOVEA.L (A5),A1 -> curpcb
ecxStart:
   BRA     forceResched
ecxBack:
   BEQ.S   ecxDone
    BCLR.B #KS_SIGNAL,(KRN_STATUS,A5)
    BSET.B #KS_NOTODO,(KRN_STATUS,A5)
    BCLR.B #KS_INHIBIT,(KRN_STATUS,A5)
ecxStopLoop
    STOP   #$2000
    BCLR.B #KS_SIGNAL,(KRN_STATUS,A5)
   BEQ.S ecxStopLoop
  BRA.S ecxStart
ecxDone:
  MOVE.B  (KRN_QUANTUM,A5),(KRN_QCNT,A5) -> reset the timer so it doesn't switch until one quantum again.
  BCLR.B  #KS_INHIBIT,(KRN_STATUS,A5)
  MOVEM.L (A7)+,D0-D7/A0-A6
  RTE -> return with a new process
ecxSupVis:
  LEA     (msgSupVisEnd,PC),A0
  MOVE.L  (A0)+,D0
  BSR     sendS
ecxStop:
  MOVE.L  A5,D0
  BTST.L  #16,D0
  BEQ.S   ecxOut
  TST.L   (ENV_CRBUCKET,A6)
  BEQ.S   ecxNoCrBkt
   MOVE.L  (ENV_CRBUCKET,A6),(-4,A7)
   MOVEM.L (A7)+,D0/A0/A5-A6 -> STACK!
   JMP     (-20,A7)          -> STACK!
ecxNoCrBkt:
  MOVEM.L (A7)+,D0/A0/A5-A6 -> STACK!
ecxLoop:
  STOP    #$2000
  BRA.S   ecxLoop
ecxOut:
 MOVEM.L (A7)+,D0/A0/A5-A6 -> STACK!
 RTE
```

F.10 Stdlib.s68

This file contains support functions for string handling and converting numbers to strings and strings to numbers.

```
* >A0.B=string1, >A1.B=string2, >D0.L=len, <D0.L=result:A0<A1=<0, A0=A1=0, A0>A1=>0
strCmp:
 MOVEM.L D1/A0/A1,-(A7)
 TST.L   D0
 BEQ.S   scOut
scLoop:
  SUBQ.L #1,D0
  BEQ.S  scDiff
  TST.B  (A0)
  BEQ.S  scDiff
  CMPM.B (A0)+,(A1)+
 BEQ.S   scLoop
 SUBQ.L  #1,A0
 SUBQ.L  #1,A1
scDiff:
 MOVEQ  #0,D0
 MOVEQ  #0,D1
 MOVE.B (A0),D0
 MOVE.B (A1),D1
 SUB.W  D1,D0
scOut:
 MOVEM.L (A7)+,D1/A0/A1
 RTS
-
-
* >A0.B=string1, >A1.B=string2, >D0.L=len, <D0.L=result:A0<>A1=0, A0=A1=-1
strCmpNC:
 MOVEM.L D1/D2/A0/A1,-(A7)
 TST.L   D0
 BEQ.S   scncEqual
scncLoop:
  MOVE.B (A0)+,D1
  BSR.S  scncAlfaCase
  EXG    D1,D2
  MOVE.B (A1)+,D1
  BEQ.S  scncDiff
  BSR.S  scncAlfaCase
  SUBQ.L #1,D0
  BEQ.S  scncDiff
  CMP.B  D2,D1
 BEQ.S   scncLoop
 MOVEQ   #0,D0
 BRA.S   scncOut
```

```
scncDiff:
 CMP.B  D2,D1
 BEQ.S  scncEqual
scncNotEqual
  MOVEQ #0,D0
  BRA.S scncOut
scncEqual:
  MOVEQ #-1,D0
scncOut:
 MOVEM.L (A7)+,D1/D2/A0/A1
 RTS
-
scncAlfaCase:
 CMPI.B #'a',D1
 BLT.S  scncNotAlfa
  CMPI.B #'z',D1
  BGT.S  scncNotAlfa
   BCLR.B #5,D1
scncNotAlfa:
 RTS
-
-
* >A0.B=ptr to 0-string, <D0.L=length excl. NIL
strLen:
 MOVE.L A0,D0
slLoop:
  TST.B (A0)+
 BNE.S  slLoop
 EXG    D0,A0
 SUBQ.L #1,D0
 SUB.L  A0,D0
 RTS
-
-
* >A0.B=dest, >A1.B=source, >D0.L=length
strCopy:
 MOVEM.L D0/A0/A1,-(A7)
 TST.L   D0
 BEQ.S   scyOut
scyMore:
  SUBQ.L  #1,D0
  BEQ.S   scyNull
  MOVE.B  (A1)+,(A0)+
 BNE.S   scyMore
 BRA.S   scyOut
scyNull:
 CLR.B   (A0)
scyOut:
 MOVEM.L (A7)+,D0/A0/A1
 RTS
-
delay:
 RTS
-
-
int2Dec:
 MOVEM.L D0,-(A7)
 MOVEM.L (A7)+,D0
 RTS
-
-
* >D0.L=int, >A0=buffer of 8 bytes to store the number, or NIL to output to serial
int2Hex:
 MOVEM.L D0-D2/A0,-(A7)
 MOVE.L  D0,D1
 MOVEQ   #7,D2
i2hLoop:
 ROL.L   #4,D1
 MOVE.B  D1,D0
 ANDI.B  #$F,D0
 CMPI.B  #9,D0
 BGT.S   i2hAPlus
  ADDI.B #'0',D0
  BRA.S  i2hDone
i2hAPlus:
  ADDI.B #'A'-10,D0
i2hDone:
  CMPA.L #0,A0
  BEQ.S  i2hSerial
   MOVE.B D0,(A0)+
   DBRA   D2,i2hLoop
i2hSerial:
  BSR    putS
 DBRA    D2,i2hLoop
 MOVEM.L (A7)+,D0-D2/A0
 RTS
-
-
* <D0.L=int, <D1.L=length, >A0.B=string
str2Int:
 MOVEM.L D2-D6/A0-A4,-(A7)
 MOVEA.L A0,A2
 MOVEA.L A0,A4
 MOVEQ   #0,D6
s2i20:
 MOVE.B  (A2)+,D0
 BEQ     s2i29
 CMPI.B  #$21,D0
 BMI.S   s2i20
 SUBQ.L  #1,A2
 CMPI.B  #$2D,(A2)
 BNE.S   s2i21
 ADDQ.L  #1,A2
 MOVEQ   #-1,D6
s2i21:
 CMPI.B  #$24,(A2)
 BEQ     s2i2B
 CMPI.B  #$25,(A2)
 BEQ     s2i35
 LEA     ($B,A2),A1
 MOVEA.L A2,A0
s2i22:
 MOVE.B  (A2)+,D0
 CMPI.B  #$3A,D0
 BPL.S   s2i23
 CMPI.B  #$30,D0
 BPL.S   s2i22
s2i23:
 SUBQ.L  #1,A2
 MOVE.L  A2,D4
 CMPA.L  A2,A0
 BEQ.S   s2i29
 MOVEQ   #0,D0
 CMPA.L  A1,A2
 BPL.S   s2i29
 LEA     (s2i25,PC),A3
s2i24:
 MOVEQ   #0,D3
 MOVE.B  -(A2),D3
 MOVE.L  (A3)+,D2
 SUBI.B  #$30,D3
 MOVE.L  D3,D1
 MULU    D2,D3
 SWAP    D2
 MULU    D2,D1
 SWAP    D1
 ADD.L   D1,D3
 ADD.L   D3,D0
 BCS.S   s2i29
 CMPA.L  A0,A2
 BNE.S   s2i24
 BRA.S   s2i26
s2i25:
 DC.L    1,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000
-
s2i26:
 SUB.L   A4,D4
 TST.L   D6
 BEQ.S   s2i28
 NEG.L   D0
s2i28:
 MOVE.L  D4,D1
 MOVEM.L (A7)+,D2-D6/A0-A4
 RTS
-
s2i29:
 MOVEQ   #0,D0
 MOVEQ   #0,D1
 MOVEM.L (A7)+,D2-D6/A0-A4
 RTS
-
s2i2B:
 ADDQ.L  #1,A2
 MOVEA.L A2,A0
 MOVE.L  A0,D3
 MOVEQ   #0,D1
 MOVEQ   #0,D2
s2i2C:
 MOVE.B  (A0)+,D0
 CMPI.B  #$47,D0
 BPL.S   s2i32
 CMPI.B  #$30,D0
 BMI.S   s2i2D
 CMPI.B  #$3A,D0
 BPL.S   s2i31
 BRA.S   s2i2C
s2i2D:
 SUBQ.L  #1,A0
 MOVE.L  A0,D4
 CMP.L   A0,D3
 BEQ.S   s2i29
s2i2E:
 CMP.L   A0,D3
 BEQ.S   s2i30
 MOVEQ   #0,D0
 MOVE.B  -(A0),D0
 CMPI.B  #$41,D0
 BPL.S   s2i33
 SUBI.B  #$30,D0
s2i2F:
 CMPI.W  #$20,D2
 BEQ     s2i29
 LSL.L   D2,D0
 ADD.L   D0,D1
 ADDQ.L  #4,D2
 BRA.S   s2i2E
s2i30:
 MOVE.L  D1,D0
 BRA.S   s2i26
s2i31:
 CMPI.B  #$41,D0
 BPL.S   s2i2C
 BRA.S   s2i2D
s2i32:
 CMPI.B  #$61,D0
 BMI.S   s2i2D
 CMPI.B  #$67,D0
 BPL.S   s2i2D
 BRA.S   s2i2C
s2i33:
 CMPI.B  #$61,D0
 BPL.S   s2i34
 SUBI.B  #$37,D0
 BRA.S   s2i2F
s2i34:
 SUBI.B  #$57,D0
 BRA.S   s2i2F
s2i35:
 ADDQ.L  #1,A2
 MOVEA.L A2,A0
 MOVE.L  A0,D3
 MOVEQ   #0,D1
 MOVEQ   #0,D2
s2i36:
 MOVE.B  (A0)+,D0
 CMPI.B  #$31,D0
 BEQ.S   s2i36
 CMPI.B  #$30,D0
 BEQ.S   s2i36
 SUBQ.L  #1,A0
 MOVE.L  A0,D4
 CMP.L   A0,D3
 BEQ     s2i29
s2i37:
 CMP.L   A0,D3
 BEQ.S   s2i39
 MOVEQ   #0,D0
 MOVE.B  -(A0),D0
 CMPI.B  #$30,D0
 BEQ.S   s2i38
 BSET.L  D2,D1
s2i38:
 CMPI.W  #$20,D2
 BEQ     s2i29
 ADDQ.L  #1,D2
 BRA.S   s2i37
s2i39:
 MOVE.L  D1,D0
 BRA     s2i26
```

F.11 Terminal.e

This file is the terminal written in E. It contains a lot of preprocessor directives which makes it possible to compile for either ExOS or AmigaOS.

```
OPT PREPROCESS
OPT REG=5
-
MODULE '*RawDoFmt'
-
-> Define AMIGA to compile for AmigaOS. Undefine to compile for ExOS.
->#define AMIGA
-
#ifdef AMIGA
 MODULE 'intuition/intuition', 'exec/tasks'
 #define SYSLIST syslist(PC)
#endif
-
#ifndef AMIGA
 #define EXOS
 #define SYSLIST 4
#endif
-
-> To be able to copy constants from assembly, EQU must be redefined as "=".
#define EQU =
#define DC CHAR
-
CONST PCB_NEXT EQU 0,  ->.L s ptr to next PCB
 PCB_DATA     EQU 4,   ->.L s ptr to the code or data
 PCB_SIZE     EQU 8,   ->.L s the size in bytes of PCB_DATA
 PCB_TYPE     EQU 12,  ->.B s is this executable PT_CODE or non-exe PT_DATA
 PCB_STATUS   EQU 13,  ->.B d if PT_CODE then PS_?
 PCB_SR       EQU 14,  ->.W d current status register
 PCB_PC       EQU 16,  ->.L d current program counter
 PCB_REGS     EQU 20,  ->.Lx15 d all 15 registers, except A7
 PCB_SP       EQU 80,  ->.L d current stack pointer
 PCB_STACK    EQU 84,  ->.L s ptr to the actual stack (low end pointer)
 PCB_STSIZE   EQU 88,  ->.L s size of the stack
 PCB_SIGMASK  EQU 92,  ->.L d allocated signalbits
 PCB_SIGWAIT  EQU 96,  ->.L d Block() was called with this signal mask. Only valid while STATUS=WAITING
 PCB_SIGSET   EQU 100, ->.L d Signals that have arrived but not yet read by block.
 PCB_NAME     EQU 104 ->.L s ptr to 0-string.
-
 -> Reserved global signals. All processes waiting for such
 -> a signal gets one when the corresponding event occurs.
CONST SIG_CTRL_C EQU 31,
 SIG_SERIAL_RX EQU 30, -> KRN_STATUS must bset KS_SERINTSIGRX for this to work.
 SIG_SERIAL_TX EQU 29, -> KRN_STATUS must bset KS_SERINTSIGTX for this to work.
 SIG_KEYBOARD  EQU 28,
 SIG_DISKREAD  EQU 27,
 SIG_DISKWRITE EQU 26,
 SIG_RESERVED1 EQU 25,
 SIG_RESERVED2 EQU 24,
 USERSIGNALS   EQU $00FFFFFF
-
CONST SI_SPEED EQU 1,  ->*.W Speed in divisor values
 SI_LINESTATUS EQU 2,  ->*.B Copy of line status register
 SI_MODSTATUS  EQU 3,  ->*.B Copy of modem status register
 SI_FIFOSIZE   EQU 4,  ->*.L The size of the FIFOs
 SI_RXFIFOLOC  EQU 5,  ->*.L Address where the FIFOs are
 SI_TXFIFOLOC  EQU 6,  ->*.L Address where the FIFOs are
 SI_RXFILLLEV  EQU 7,  ->*.L #of bytes currently in the Rx FIFO
 SI_TXFILLLEV  EQU 8,  ->*.L #of bytes currently in the Tx FIFO
 SI_RXTOTAL    EQU 9,  ->*.L Total #of bytes transferred to the Rx FIFO
 SI_TXTOTAL    EQU 10  ->*.L Total #of bytes transferred to the Tx FIFO
-
CONST FIFO_SIZE     EQU 1024,
 SER_16X2400   EQU 400, ->2400   @15.360MHz
 SER_16X4800   EQU 200, ->4800   @15.360MHz
 SER_16X9600   EQU 100, ->9600   @15.360MHz
 SER_16X19200  EQU 50,  ->19200  @15.360MHz
 SER_16X38400  EQU 25,  ->38400  @15.360MHz
 SER_16X57600  EQU 17,  ->57600  @15.360MHz (exactly 16.666...)
 SER_16X115200 EQU 8    ->115200 @15.360MHz (exactly 8.3333...)
-
CONST MI_TOTALFREE   EQU 0
CONST MI_LARGEST     EQU 1
CONST MI_NUMALLOC    EQU 2
CONST MI_NUMFREE     EQU 3
CONST MEM_DEFNUMALLOC EQU 1000,
      MEM_DEFSIZE     EQU 1000*12,
      FR_LST_E_SIZE   EQU 3*4
-
CONST KS_INHIBIT     EQU 0 -> inhibit rescheduling on int3 (as block: is currently rescheduling)
CONST KS_NOTODO      EQU 1 -> do not reschedule because there is no READY process
CONST KS_SIGNAL      EQU 2 -> set this to wake up from NOTODO
CONST KS_FORBID      EQU 3 -> a userprogram requests to be alone. This gets cancelled if block: is called.
CONST KS_SERINTSIGTX EQU 4 -> Serial called block with SIG_SERIAL_TX and expects int4tx to signal.
CONST KS_SERINTSIGRX EQU 5 -> Serial called block with SIG_SERIAL_RX and expects int4rx to signal.
CONST DEFQUANTUM     EQU 3 ->10 -> Default number of LFC-ticks between rescheduling
-
CONST TAllocMem EQU 0, -> >D0.L=size, <A0=mem
 TFreeMem      EQU 1, -> >A0=mem
-> TTimer        EQU 2, -> >D3.L=command
 TReSchedule   EQU 3, -> no parameters (callable only from block:)
 TStop         EQU 4, -> no parameters (execute STOP #2000)
-> TSwapROMRAM   EQU 5, -> no parameters (all stacks must be clean for this to work)
 TDebug        EQU 6, -> no parameters (prints current PC, SR and SP)
 TUserTrap     EQU 7 -> >A0=ptr to function to be called and should end with RTE
-> TReset        EQU 15 -> no parameters
-
CONST FDelay   EQU -6*33,->  >D0.L=microseconds(max 1M), >D1.W=seconds
 FEvent        EQU -6*32,->
 FConnect      EQU -6*31,->  >D0.B=Device (DEV_*), >D1.W=
 FInitSerial   EQU -6*30,->  >A6=ptr to envvars, trashes various registers..
 FSetSerSpeed  EQU -6*29,->  >D0.W=speed (SER_16Xspeed)
 FGetSerInfo   EQU -6*28,-> <>A0.L=ptr to taglist that will be filled with data.
 FSendS        EQU -6*27,->  >A0=data,  >D0.L(usig)=size  Send serial data synchronously
 FSendA        EQU -6*26,->  >A0=data, <>D0.L(usig)=size  Send serial data asynchronously
 FGetS         EQU -6*25,-> < D0.B=data  Get one byte serial data synch
 FGetA         EQU -6*24,-> < D0.L(B)=data <D0.L=-1=no data  Get one byte serial data asynch if it exists
 FPutS         EQU -6*23,->  >D0.B=data  Send one byte serial data synch
 FReadS        EQU -6*22,-> <>A0=buf, >D0.L(usig)=size  Read serial data synch
 FFlushTx      EQU -6*21,->
 FFlushRx      EQU -6*20,-> no parameters
 FPutStr       EQU -6*19,->  >A0=null-terminated string
 FStrCmp       EQU -6*18,->
 FStrCmpNC     EQU -6*17,->
 FStrLen       EQU -6*16,->
 FStrCopy      EQU -6*15,->
 FStr2Int      EQU -6*14,->
 FInt2Dec      EQU -6*13,->
 FInt2Hex      EQU -6*12,->  >D0.L=value to convert, >A0=buffer of 8 bytes to store the number, or NIL to get the output to s
 FCompFreeList EQU -6*11,-> no parameters
 FMemInfo      EQU -6*10,-> >D0.B=0=total,1=largest,2=numalloc,3=numfree, <D0.L=avail size/num
 FMemCheck     EQU -6*9, -> >A0.L=Start address, even or odd, >D0.L=Amount of bytes, D0.L> 0=OK,1=Not OK
 FStoreProg    EQU -6*8, -> >A0=linked list of s-recs, <A0=addr to decoded program or NIL
 FGetEnvToA6   EQU -6*7, ->< A6=ptr to envvars
 FIncLed       EQU -6*6, -> >A0=ptr to envvars or 0 when >D7.B is used
 FSingleTask   EQU -6*5, -> >D0.B <>0=forbid multitasking, =0=permit multitasking. A Block() automatically permits multitaski
 FSignal       EQU -6*4, -> >A0=pcb, D0.L=sigmask to signal to A0. <D0.L=-1 ok, 0=pcb doesn't exist
 FBlock        EQU -6*3, -> >D0.L=signal mask, <D0.L=signals received
 FFreeSignal   EQU -6*2, -> >D0.B=the number of the sigbit to clear
 FAllocSignal  EQU -6*1  ->< D0.B=the number of the allocated sigbit, <D0.L=-1=no free signal
-
CONST ROM     EQU 0,
 RAM          EQU $100000,
 SYSSP        EQU $140000,
 SSPSIZE      EQU $1000,
 USPSIZE      EQU $1000,
 MEMSIZE      EQU $40000,
 EM_SWAP      EQU 0, -> Ram and ROM are swapped
 EM_IRQINH    EQU 1, -> inhibit all interrupts
 EM_LFCINH    EQU 2, -> inhibit LF-clock interrupts
-
 ES_SWAP      EQU 0, -> swapped ROM<->RAM
 ES_SER       EQU 1, -> serial is attached
 ES_LFLED     EQU 2, -> LF-clock interrupt outputs and increments ENV_LED->LED
 ES_KERNEL    EQU 3, -> ENV_KERNEL is initiated.
-
 ENV_MCR      EQU 0,   -> *Copy of Master Control Register: EM_*
 ENV_LFC      EQU 1,   -> *Copy of LF-clock register
 ENV_LED      EQU 2,   -> *Copy of LED register (not always used)
 ENV_INT7     EQU 3,   -> *<>0 - int7 got data for int1 to process, =0 - no data for int1
 ENV_STATUS   EQU 4,   -> *Status: ES_*
-
 ENV_SRBINT3  EQU 15*4,-> *This will execute before the scheduler!
 ENV_SRAINT3  EQU 16*4,-> *Used by the timer
 ENV_SREXCEPT EQU 29*4,-> *Exceptions subroutine. On the stack a WORD with the vector number will be.
 ENV_TIMER    EQU 30*4,-> *Timerbase
 ENV_CONNECT  EQU 31*4,-> *Connect base
 ENV_PCB      EQU 32*4,-> *Process Control Block base
 ENV_KERNEL   EQU 33*4,-> *Kernel pointer
 ENV_CRBUCKET EQU 34*4,-> *Place where a supervisor crash can escape
-
 KRN_CURPCB   EQU 0, -> *.L ptr to currently running PCB
 KRN_SSPREF   EQU 4, -> *.L the highest point in the supervisor stack.
 KRN_TOPPC    EQU 8, -> *.L temp storage for the PC at SSPREF
 KRN_TOPSR    EQU 12,-> *.W temp storage for the PC at SSPREF
 KRN_STATUS   EQU 14,-> *.B KS_*
 KRN_QCNT     EQU 15,-> *.B counter from KRN_QUANTUM to 0. When it reaches 0 a rescheduling will occur.
 KRN_QUANTUM  EQU 16 -> *.B how many interrupt that should pass before a rescheduling should occur.
-
CONST DIP=$20001E, LED=$20001F, MCR=$20001D, LFC=$20001C
CONST NUM_COMMANDS=15, TAB=9, BACKSP=8, DEL=127,
      CTRL_A=1, CTRL_C=3, CTRL_D=4, CTRL_S=19, CTRL_W=23,
      DEF_STACK=1024, MYSELF=35000, TOKENS=6, MAXTOKEN=5
-
CONST PT_CODE=1, PT_DATA=2, PT_SCRIPT=3,
      PT_PROTCODE=$41, PT_PROTDATA=$42, PT_PROTSCRIPT=$43,
      PT_PROT=$40
-
SET  BYTE, WORD, LOONG, STR
ENUM PS_NEW, PS_SUSPENDED, PS_FINISHED, PS_READY, PS_RUNNING, PS_WAITING,
     PS_CRASHED=$80
-
OBJECT envvars
 mcr    :CHAR
 lfc    :CHAR
 led    :CHAR
 int7   :CHAR
 st     :CHAR
 ramlock:CHAR
 rxlock :CHAR
 txlock :CHAR
 pad    :INT
 freez  :INT
 freel  :LONG
 seriptr:PTR TO CHAR
 seribeg:LONG
 seriend:LONG
 seroptr:PTR TO CHAR
 serobeg:LONG
 seroend:LONG
 id     :LONG
 srbint1:LONG
 sraint1:LONG
 srbint2:LONG
 sraint2:LONG
 srbint3:LONG
 sraint3:LONG
 srbint4:LONG
 sraint4:LONG
 srbint5:LONG
 sraint5:LONG
 srbint6:LONG
 sraint6:LONG
 srbint7:LONG
 sraint7:LONG
 srarx:LONG
 sratx:LONG
 sraline:LONG
 sramodem:LONG
 srexcept:LONG
 timer:PTR TO timer
 connect:PTR TO connect
 pcb:PTR TO program
 kernel:PTR TO kernel
 crbucket:LONG
ENDOBJECT
-
OBJECT timer
 lastReg:INT
 timerEl:PTR TO CHAR -> [128]:timerElement
ENDOBJECT
-
OBJECT kernel
 curpcb:PTR TO program ->.L ptr to currently running PCB
 sspref:LONG  ->.L the highest point in the supervisor stack.
 toppc:LONG   ->.L temp storage for the PC at SSPREF
 topsr:INT    ->.W temp storage for the PC at SSPREF
 status:CHAR  ->.B KS_*
 qcnt:CHAR    ->.B counter from KRN_QUANTUM to 0. When it reaches 0 a rescheduling will occur.
 quantum:CHAR ->.B how many interrupt that should pass before a rescheduling should occur.
ENDOBJECT
-
OBJECT connect
 next:PTR TO connect
 device:CHAR
 type:CHAR
 unit:INT
 uAddr:LONG
 iAddr:LONG
ENDOBJECT
-
OBJECT program
 next:PTR TO program
 data:PTR TO CHAR
 size:LONG
 type:CHAR
 status:CHAR
 sr:INT
 pc:LONG
 regs[15]:ARRAY OF LONG
 sp:PTR TO LONG
 stack:PTR TO LONG
 stsize:LONG
 sigmask:LONG
 sigwait:LONG
 sigset:LONG
 name:PTR TO CHAR
ENDOBJECT
-
OBJECT loadS19
 data:PTR TO CHAR
 next:PTR TO loadS19
ENDOBJECT
-
OBJECT commands
 c:PTR TO CHAR
 h:PTR TO CHAR
 h2:PTR TO CHAR
ENDOBJECT
-
#ifdef EXOS
 #define GLOB1SIZE 18
 CONST GLOB4SIZE=GLOB1SIZE/4
#endif
#ifdef AMIGA
 DEF lastchar=-1, scrap, w:PTR TO window, task:PTR TO tc, sspref[40]:ARRAY
 #endif
-
DEF x, y, z, a, buf, doskey, tl:PTR TO LONG, s:PTR TO CHAR, t:PTR TO CHAR,
    ab, aw:PTR TO INT, al:PTR TO LONG, outOfMem, noEntry, invParams,
    prg:PTR TO program, envvars:PTR TO envvars, help:PTR TO commands
-
PROC main()
 DEF tl_[TOKENS]:ARRAY OF LONG, t_[20]:STRING, fail=TRUE
#ifdef EXOS
 DEF globs[GLOB1SIZE]:ARRAY OF LONG, self -> E uses A4 for global variables but as the AmigaOS startup
 MOVEA.L globs,A4                         -> code has been stripped in ExOS mode it must be setup here
 ADDA.L  #$200+GLOB4SIZE,A4               -> using a piece of the stack..
 LEA     a4(PC),A0
 MOVE.L  A4,(A0)
#endif
 putStr('Starting the terminal...\n')
 tl:=tl_
 t:=t_
 outOfMem:='Out of memory!\n'       -> saving precious RAM by using only one copy of these std strings..
 noEntry:='Unable to find entry!\n'
 invParams:='Invalid parameters!\n'
 putStr('Installing the kernel..\n')
 IF install()
  putStr('Allocating buffers..\n')
#ifdef AMIGA
  task:=FindTask(NIL)
  task.trapcode:={trapcode}
  IF (w:=OpenW(900,400,100,30,$200600,$100a,'input',NIL,1,NIL))=NIL THEN CleanUp()
  WriteF('scrap=$\h, \d\ntrapcode=$\h\n',{scrap},{scrap},task.trapcode)
#endif
  NOP
  NOP
  NOP
  IF s:=allocStr(400)
   stringf(s,'Expected startup address: $\h, main() is located at: $\h\n',
           RAM+$1244+FIFO_SIZE+FIFO_SIZE+MEM_DEFSIZE,{main})
   putStr(s)
#ifdef EXOS
   IF self:=allocMem(MYSELF) -> As the console doesn't alloc memory, the terminal has to alloc its own memory
#endif
    IF buf:=allocMem(256)
     IF doskey:=allocMem(256)
      doskey[0]:=NIL
      putStr('\nWelcome to the Terminal v1.0 in ExOS! (2004-04-27)\n' +
             'Type "help" or "?" to get a list of commands.\n\n')
      help:=['clear',NIL,NIL,'cls', NIL,NIL,'delete',NIL,NIL,'echo',NIL,NIL,
             'exit', NIL,NIL,'help',NIL,NIL,'list',  NIL,NIL,'load',NIL,NIL,
             'meminfo',NIL,NIL,'peek', NIL,NIL,'poke',NIL,NIL,'rename',NIL,NIL,
             'run',    NIL,NIL,'speed',NIL,NIL,'stop',NIL,NIL]:commands
      inithelp()
      terminal()
      fail:=FALSE
      putStr('Freeing buffers..\n')
      freeMem(doskey)
     ENDIF
     freeMem(buf)
    ENDIF
#ifdef EXOS
    freeMem(self)
   ENDIF
#endif
   freeStr(s)
  ENDIF
  putStr('Disengaging the kernel..\n')
  forbid()
  freeMem(envvars.pcb.name)
  freeMem(envvars.pcb)
  envvars.pcb:=NIL
#ifdef AMIGA
  CloseW(w)
#endif
 ENDIF
 IF fail THEN putStr('Could not allocate memory for myself, buffers or structures!\n')
ENDPROC
-
PROC ws(c) IS c=" " OR (c=13) OR (c="\n") OR (c="\t")
-
-> Ctrl codes:
-> l=cls, k=line up, g=bell, n=8-bit, o=7-bit
PROC terminal()
 DEF eko=0
 LOOP
  putStr('T> ')
  y:=z:=0  -> cursor, size
  REPEAT
   x:=getS()
   IF eko
    stringf(t,'\r\z\h[2] ',x)
    sendS(t,3)
   ENDIF
   SELECT x
   CASE CTRL_W -> "pil upp"
    a:=z-y
    WHILE y<z
     putS(" ")
     INC y
    ENDWHILE
    y:=StrLen(doskey)
    WHILE z
     IF eko=0
      IF a<z THEN sendS([BACKSP," ",BACKSP]:CHAR,3) ELSE putS(BACKSP)
     ENDIF
     DEC z
    ENDWHILE
    AstrCopy(buf,doskey,256)
    z:=y
    putStr(buf)
   CASE CTRL_A -> "pil vnster"
    IF y
     DEC y
     putS(BACKSP)
    ENDIF
   CASE CTRL_D -> "pil hger"
    IF y<z
     putS(buf[y])
     INC y
    ENDIF
   CASE CTRL_S -> "pil ner"
    IF z
     a:=z-y
     WHILE y<z
      putS(" ")
      DEC z
     ENDWHILE
     WHILE a
      putS(BACKSP)
      DEC a
     ENDWHILE
     WHILE y
      sendS([BACKSP," ",BACKSP]:CHAR,3)
      DEC y
     ENDWHILE
     z:=0
    ENDIF
   CASE BACKSP
    IF y
     IF y<z
      putS(BACKSP)
      sendS(buf+y,z-y)
      sendS([" ",BACKSP]:CHAR,2)
      DEC y
      DEC z
      FOR a:=y TO z-1
       buf[a]:=buf[a+1]
       putS(BACKSP)
      ENDFOR
     ELSE
      DEC y
      DEC z
      IF eko=0 THEN sendS([BACKSP," ",BACKSP]:CHAR,3)
     ENDIF
    ENDIF
   CASE DEL
    IF y<z
     DEC z
     sendS(buf+y+1,z-y)
     sendS([" ",BACKSP]:CHAR,2)
     FOR a:=y TO z-1
      buf[a]:=buf[a+1]
      putS(BACKSP)
     ENDFOR
    ENDIF
   DEFAULT
    IF z<256
     IF eko=0 THEN putS(x)
     IF x>=32 OR (x="\t")
      IF y<z
       IF eko=0 THEN sendS(buf+y,z-y)
       FOR a:=z TO y+1 STEP -1
        buf[a]:=buf[a-1]
        putS(BACKSP)
       ENDFOR
      ENDIF
      buf[y]:=x
      INC y
      INC z
     ENDIF
    ENDIF
   ENDSELECT
  UNTIL x=10 OR (x=13)
#ifdef AMIGA
 IF x=13 THEN putS("\n")
#endif
  IF z
   WHILE z=256 OR ws(buf[z-1]) AND z DO DEC z
   buf[z]:=0
  ENDIF
  IF z
```

```
   AstrCopy(doskey,buf,256)
   tokenize()
putStr('Tokenizer says..:')
y:=0
WHILE tl[y]
 sendS('\n"',2)
 putStr(tl[y++])
 putS("\q")
ENDWHILE
putS("\n")
   a:=tl[]
   LowerStr(a)
   IF StrCmp(a,'cls') ; putStr([$9b48,$9b4a,0]:INT)
   ELSEIF StrCmp(a,'exit') ; putStr('The Terminal is exiting...\n') ; RETURN
   ELSEIF StrCmp(a,'help') OR StrCmp(a,'?',x) ; hlp()
   ELSEIF StrCmp(a,'list')    ; list()
   ELSEIF StrCmp(a,'load')    ; load()
   ELSEIF StrCmp(a,'meminfo') ; meminfo()
   ELSEIF StrCmp(a,'speed')   ; speed()
   ELSEIF tl[1]
    IF     StrCmp(a,'clear')  ; clear()
    ELSEIF StrCmp(a,'delete') ; delete()
    ELSEIF StrCmp(a,'echo')   ; eko:=echo()
    ELSEIF StrCmp(a,'peek')   ; peek()
    ELSEIF StrCmp(a,'run')    ; run()
    ELSEIF StrCmp(a,'stop')   ; stop()
    ELSEIF tl[2]
     IF     StrCmp(a,'poke')   ; poke()
     ELSEIF StrCmp(a,'rename') ; rename()
     ELSE
      stringf(s,'Unknown command: "\s"\n',a)
      putStr(s)
     ENDIF
    ELSE
     stringf(s,'Unknown command or too few parameters: "\s"\n',a)
     putStr(s)
    ENDIF
   ELSE
    stringf(s,'Unknown command or too few parameters: "\s"\n',a)
    putStr(s)
   ENDIF
  ENDIF
 ENDLOOP
ENDPROC
-
PROC clear()
 DEF x, y, ab
 IF prg:=findprg(tl[1])
  IF prg.type AND PT_CODE
   IF prg.status=PS_SUSPENDED OR (prg.status=PS_FINISHED)
    prg.sr:=0
    prg.pc:=prg.data
    prg.sp:=NIL
    prg.status:=PS_NEW
    IF prg.stack
     freeMem(prg.stack)
     prg.stack:=NIL
    ENDIF
    prg.stsize:=DEF_STACK
    prg.sigmask:=0
   ELSEIF prg.status=PS_NEW
    putStr('No point to clear a newly downloaded or cleared program.\n')
   ELSE
    putStr('Can not clear a running program.\n')
   ENDIF
  ELSEIF prg.type AND PT_DATA
   IF prg.type=PT_PROTDATA
    putStr('Cannot clear a protected data area.\n')
   ELSE
    y:=prg.size-1
    ab:=prg.data
    FOR x:=0 TO y DO ab[]++:=0
   ENDIF
  ELSEIF prg.type AND PT_SCRIPT
   putStr('A script cannot be cleared.\n')
  ENDIF
 ELSE
  putStr(noEntry)
 ENDIF
ENDPROC
-
PROC delete()
 IF prg:=findprg(tl[1])
  IF prg.type AND PT_PROT
   putStr('This entry is protected against deletion.\n')
  ELSE
   putStr('Are you sure you want to delete this ')
   putStr(IF prg.type AND 3=2 THEN 'data block' ELSE 'program')
   putStr('? y/N: ')
   x:=getS()
   putS(x)
   IF x OR 32="y"
    IF prg.type=PT_CODE
     forbid()
     IF prg.status=PS_NEW OR (prg.status=PS_SUSPENDED) OR (prg.status=PS_FINISHED) OR (prg.status=PS_CRASHED)
      IF prg=envvars.pcb
       envvars.pcb:=prg.next
      ELSE
       x:=prg
       prg:=envvars.pcb
       WHILE prg.next<>x AND prg.next DO prg:=prg.next
       prg.next:=prg.next.next
       prg:=x
      ENDIF
      StrCopy(s,prg.name)
      IF prg.stack THEN freeMem(prg.stack)
      freeMem(prg.name)
      freeMem(prg.data)
      x:=1
     ELSE
      putStr('This program is running. You must stop it before you can remove it.\n')
      x:=0
     ENDIF
     permit()
    ELSE
     freeMem(prg.name)
     freeMem(prg.data)
     x:=1
    ENDIF
    IF x
     putS("\q")
     putStr(s)
     putStr('" has been deleted.\n')
    ENDIF
   ENDIF
  ENDIF
 ELSE
  putStr(noEntry)
 ENDIF
ENDPROC
-
PROC echo()
 IF x:=Val(tl[1])
  y:=bwl(tl[2])
  SELECT y
  CASE WORD
   putS(Shr(x,8))
   putS(x)
  CASE LOONG
   putS(Shr(x,24))
   putS(Shr(x,16))
   putS(Shr(x,8))
   putS(x)
  DEFAULT
   putS(x)
  ENDSELECT
 ELSE
  IF Long(tl[1]) AND $dfdfff00="ON\0\0"
   putStr('Global echo mode is now on. Type "echo OFF" to turn it off again.\n')
   RETURN TRUE
  ELSEIF Long(tl[1]) AND $dfdfdfff="OFF\0"
   putStr('Global echo has been turned off.\n')
   RETURN FALSE
  ELSE
   putStr(invParams)
  ENDIF
 ENDIF
ENDPROC
-
PROC hlp()
 IF tl[1]
  FOR y:=0 TO NUM_COMMANDS-1 DO EXIT StrCmp(tl[1],help[y].c,z)
  IF y<NUM_COMMANDS
   putStr(help[y].h)
   IF help[y].h2 THEN putStr(help[y].h2)
   putS(10)
   putS(10)
  ENDIF
 ELSE
  FOR x:=0 TO NUM_COMMANDS-1
   putStr(help[x].c)
   putS("\n")
   y:=getA()
   EXIT y=CTRL_C
   IF y>=32
    REPEAT
     y:=getS()
    UNTIL y=10 OR (y=13)
   ENDIF
  ENDFOR
  putStr('\nType "help <command>" to get help about a specific ' +
         'command. F.ex: "help help".\n\n')
 ENDIF
ENDPROC
-
PROC list()
 al:=['New','Suspended','Finished','Ready','Running','Waiting','Crashed','Undefined']
 aw:=['Code','Data','Script','PCode','PData','PScript','Unknown']
 IF tl[1]
  IF prg:=findprg(tl[1])
   y:=prg.type-1
   IF y AND PT_PROT THEN y:=y AND Not(PT_PROT)+3
   IF y<0 OR (y>6) THEN y:=7
   y:=Shl(y,2)
   x:=prg.status
   IF x>=PS_CRASHED
    x:=PS_WAITING+1
   ELSEIF PS_WAITING+1<x
```

```
    x:=PS_WAITING+2
   ENDIF
   stringf(s,'Name: "\s"\nSize: \d bytes\nLocation: \r$\z\h[6]\n' +
             'Type:   \s\nStatus: \s\nProgram counter: $\z\h[6]\n' +
             'Status register: $\z\h[4]\nStack location: $\z\h[6]\n',
             prg.name,prg.size,prg.data,Long(aw+y),al[x],prg.pc,prg.sr,prg.stack)
   putStr(s)
   stringf(s,'Stack pointer:  $\z\h[6]\nStack size: \d bytes\n' +
             'SigAlloc: $\z\h[8]\nSigSet:   $\z\h[8]\n' +
             'SigWait:  $\z\h[8]\nRegisters:\n',
             prg.sp,prg.stsize,prg.sigmask,prg.sigset,prg.sigwait)
   putStr(s)
   al:=prg.regs
   stringf(s,'\rD0:\z\h[8] D1:\z\h[8] D2:\z\h[8] D3:\z\h[8]\n' +
             'D4:\z\h[8] D5:\z\h[8] D6:\z\h[8] D7:\z\h[8]\n',
             al[0],al[1],al[2],al[3],al[4],al[5],al[6],al[7])
   putStr(s)
   stringf(s,'A0:\z\h[8] A1:\z\h[8] A2:\z\h[8] A3:\z\h[8]\n' +
             'A4:\z\h[8] A5:\z\h[8] A6:\z\h[8]\n',
             al[8],al[9],al[10],al[11],al[12],al[13],al[14],al[15])
   putStr(s)
  ELSE
   putStr(noEntry)
  ENDIF
 ELSE
  prg:=envvars.pcb
  a:=z:=0
  putStr('Name:                Size: Type:   Status:   Location: PC:     Stack: \n')
  WHILE prg
   y:=prg.type-1
   IF y AND PT_PROT THEN y:=y AND Not(PT_PROT)+3
   IF y<0 OR (y>6) THEN y:=7
   y:=Shl(y,2)
   x:=prg.status
   IF x>=PS_CRASHED
    x:=PS_WAITING+1
   ELSEIF PS_WAITING+1<x
    x:=PS_WAITING+2
   ENDIF
   IF x=6
    stringf(s,'\l\s[20] \r\d[5] \l\s[7] \rCrashed\h[2] $\z\h[6]   $\z\h[6] \d[5]  \n',
            prg.name,prg.size,Long(aw+y),prg.status AND $7f,prg.data,prg.pc,prg.stsize)
   ELSE
    stringf(s,'\l\s[20] \r\d[5] \l\s[7] \s[9] \r$\z\h[8] $\z\h[6] \d[5]  \n',
            prg.name,prg.size,Long(aw+y),al[x],prg.data,prg.pc,prg.stsize)
   ENDIF
   putStr(s)
   INC a
   z:=z+prg.size
   prg:=prg.next
   x:=getA()
   EXIT x=CTRL_C
   IF x>=32
    REPEAT
     x:=getS()
    UNTIL x=10 OR (x=13)
   ENDIF
  ENDWHILE
  stringf(s,'\n\d entries in \d bytes\n',a,z)
  putStr(s)
 ENDIF
ENDPROC
-
PROC load()
 DEF x, y, z, a, b, c, data, type, load, l:PTR TO loadS19
 data:=type:=b:=0
 IF a:=tl[1]
  z:=2
  IF strCmpNC(a,'S19') ; data:=0
  ELSEIF strCmpNC(a,'BINARY') ; data:=1
  ELSE
   data:=type:=0
   z:=1
   b:=a
  ENDIF
  IF z=2 THEN a:=tl[2]
  IF a
   b:=tl[z+1]
   IF strCmpNC(a,'PROGRAM') ; type:=0
   ELSEIF strCmpNC(a,'DATA') ; type:=1
   ELSEIF strCmpNC(a,'SCRIPT') ; type:=2
   ELSE
    IF b
     putStr(invParams)
     RETURN
    ENDIF
    b:=a
   ENDIF
  ENDIF
 ENDIF
 IF b=NIL
  prg:=envvars.pcb
  z:=0
  WHILE prg
   b:=prg.name[]
   IF b="P" OR (b="D") OR (b="S") THEN z:=Max(z,Val(prg.name+1))
   prg:=prg.next
  ENDWHILE
  stringf(t,'\c\d',Char('PDS'+type),z)
  b:=t
 ENDIF
 al:=['a program','data','a script']
 stringf(s,'Loading \s data as \s named "\s"..',
         IF data=0 THEN 'S19' ELSE 'binary',al[type],b)
 putStr(s)
 prg:=NIL
 IF data
  readS({z},4)
  IF x:=allocMem(z)
   putS(".")
   prg:=envvars.pcb
   WHILE prg.next DO prg:=prg.next
   forbid()
   prg.next:=allocMem(StrLen(b)+1+SIZEOF program)
   IF prg.next
    prg:=prg.next
    prg.next:=NIL
    prg.status:=PS_NEW
    permit()
    readS(x,z)
    putStr('download finished!\n')
   ELSE
    permit()
    prg:=NIL
    putStr(outOfMem)
   ENDIF
  ELSE
   putStr(outOfMem)
  ENDIF
 ELSE
  load:=l:=NIL
  REPEAT
   c:=getS()
   IF c=CTRL_C THEN RETURN
  UNTIL c="S"
  x:=TRUE
  z:=0
  REPEAT
   readS(t,3)
   a:=Shl(IF t[1]>="A" THEN t[1]-"A"+10 ELSE t[1]-"0",4) OR (IF t[2]>="A" THEN t[2]-"A"+10 ELSE t[2]-"0")
#ifdef AMIGA
WriteF('a=$\h\n',a)
#endif
   y:=t[]
   IF y>="0" AND (y<="9") AND x
    IF c:=allocMem(a*2+4+SIZEOF loadS19)
     IF load
      l.next:=c
      l:=l.next
     ELSE
      load:=l:=c
     ENDIF
     l.next:=NIL
     l.data:=l+SIZEOF loadS19
     l.data[]:="S"
     l.data[1]:=y
     l.data[2]:=t[1]
     l.data[3]:=t[2]
     readS(l.data+4,a*2)
     z:=z+a-4
    ELSE
     putStr(outOfMem)
     x:=FALSE
     MOVEA.L SYSLIST,A0
     JSR     FFlushRx(A0)
    ENDIF
    putS(".")
   ENDIF
   IF y<"7" OR (y>"9")
    REPEAT
     c:=getS()
    UNTIL c="S" OR (c=CTRL_C)
    IF c=CTRL_C THEN x:=FALSE
   ENDIF
  UNTIL y>="7" AND (y<="9") OR (x=FALSE)
  IF x=FALSE
   l:=load
   WHILE l
    load:=l
    l:=l.next
    freeMem(load)
   ENDWHILE
   RETURN
  ENDIF
  IF load
#ifdef AMIGA
WriteF('z=$\h \d\n',z,z)
#endif
   putStr('finished! Now decoding..')
   MOVE.L load,A0
   MOVE.L SYSLIST,A6
   JSR    FStoreProg(A6)
   MOVE.L A0,data
   MOVE.L D0,z
#ifdef AMIGA
WriteF('z=$\h \d\n',z,z)
#endif
   IF data
    prg:=envvars.pcb
    WHILE prg.next DO prg:=prg.next
    forbid()
    prg.next:=allocMem(StrLen(b)+1+SIZEOF program)
    IF prg.next
     prg:=prg.next
     prg.next:=NIL
     prg.status:=PS_NEW
     permit()
     putStr('.done!\n')
    ELSE
     permit()
     prg:=NIL
     putStr(outOfMem)
    ENDIF
   ELSEIF z=0
    putStr(outOfMem)
   ELSEIF z=-1
    putStr('Checksum error in S19 data!\n')
   ELSEIF z=-2
    putStr('No data in S19 file.\n')
   ENDIF
   l:=load
   WHILE l
    load:=l
    l:=l.next
    freeMem(load)
   ENDWHILE
  ELSE
   putStr('finished! No data was transfered!\n')
   RETURN
  ENDIF
 ENDIF
 IF prg
  prg.name:=prg+SIZEOF program
  AstrCopy(prg.name,b,ALL)
  prg.data:=data
  prg.size:=z
  prg.stsize:=0
  SELECT type
  CASE 0
   prg.type:=PT_CODE
   prg.stsize:=DEF_STACK
  CASE 1
   prg.type:=PT_DATA
  CASE 2
   prg.type:=PT_SCRIPT
  ENDSELECT
 ENDIF
ENDPROC
-
PROC meminfo()
 DEF a, b, c, d
 x:=1
 a:=b:=c:=d:=y:=0
 WHILE tl[x]
  IF strCmpNC(tl[x],'MEMHDL') ; a:=1
  ELSEIF strCmpNC(tl[x],'VERBOSE') ; b:=1
  ELSEIF strCmpNC(tl[x],'COMPRESS') ; c:=1
  ELSEIF strCmpNC(tl[x],'CHECK') ; d:=1
  ELSE
   y:=1
  ENDIF
 ENDWHILE
 IF y
  putStr(invParams)
  RETURN
 ENDIF
 IF a=0 THEN b:=0
 IF c
  MOVE.L SYSLIST,A6
  JSR    FCompFreeList(A6)
 ENDIF
 MOVE.L SYSLIST,A6
 MOVEQ  #MI_TOTALFREE,D0
 JSR    FMemInfo(A6)
 MOVE.L D0,x
 MOVEQ  #MI_LARGEST,D0
 JSR    FMemInfo(A6)
 MOVE.L D0,y
 stringf(s,'Total free memory: \d bytes, largest free block: \d bytes\n',x,y)
 putStr(s)
 IF a
  MOVE.L SYSLIST,A6
  MOVEQ  #MI_NUMALLOC,D0
  JSR    FMemInfo(A6)
  MOVE.L D0,x
  MOVEQ  #MI_NUMFREE,D0
  JSR    FMemInfo(A6)
  MOVE.L D0,y
  stringf(s,'\d allocations have been made.\n' +
            '\d additional allocations can be made before the current free list is full.\n' +
            '\d is the size of the current free list.\n',x,y,x+y)
  putStr(s)
  IF x+y<>envvars.freez
   stringf(s,'Oops! The calculated size of the current free list does not ' +
             'match the maximum size in envvars which is \d!\n',envvars.freez)
   putStr(s)
  ENDIF
  IF b
   putStr('Status:   Start: End:   Size:\n')
   aw:=envvars.freel
   b:=aw[]++
   al:=aw
   FOR a:=0 TO b
    IF al[2]
     stringf(s,'Free:     \r\z\z\z\z\z\h[6] \z\z\z\z\z\h[6] \z\z\z\z\z\h[6]=\d\n',
             al[]++,al[]++,al[],al[]++)
    ELSE
     c:=al[1]-al[]
     stringf(s,'Occupied: \r\z\z\z\z\z\h[6] \z\z\z\z\z\h[6] \z\z\z\z\z\h[6]=\d\n',
             al[]++,al[]++,c,c)
     al[]++
    ENDIF
    putStr(s)
    c:=getA()
    EXIT c=CTRL_C
    IF c>=32
     REPEAT
      c:=getS()
     UNTIL c=10 OR (c=13)
    ENDIF
   ENDFOR
  ENDIF
 ENDIF
 IF d
  MOVE.L  SYSLIST,A6
  MOVE.L  #131072,D0
  JSR     FMemCheck(A6)
  MOVE.L  D0,x
  MOVE.L  #131072,D0
  JSR     FMemCheck(A6)
  ADDQ.L  #1,A0
  MOVE.L  D0,y
  IF x
   putStr('Atleast one byte in the memory on the most significant ' +
          'part of the data bus was erroneous!\n')
  ELSE
   putStr('No errors were found in the most significant memory IC.\n')
  ENDIF
  IF y
   putStr('Atleast one byte in the memory on the least significant ' +
          'part of the data bus was erroneous!\n')
  ELSE
   putStr('No errors were found in the least significant memory IC.\n')
  ENDIF
 ENDIF
ENDPROC
-
PROC peek()
 LowerStr(tl[1])
 IF StrCmp(tl[1],'dip')
  ab:=DIP
  stringf(s,'DIP: $\r\z\h[2] \d "\c"\n',ab[],ab[],IF ab[]<32 THEN " " ELSE ab[])
 ELSEIF StrCmp(tl[1],'mcr')
  stringf(s,'MCR: $\r\z\h[2] \d\n',envvars.mcr,envvars.mcr)
 ELSE
  x:=BYTE
  y:=Val(tl[1])
  z:=1
  IF tl[2]
   x:=bwl(tl[2])
   IF x=0
    x:=BYTE
    z:=Val(tl[2])
   ELSEIF tl[3]
    z:=Val(tl[3])
   ENDIF
  ENDIF
  IF x=STR
   sendS(y,z)
  ELSE
   ab:=aw:=al:=y
   IF z=1
    SELECT x
    CASE BYTE
     stringf(s,'\r\z\z\z\z\z\h[6]: $\z\h[2] \d "\c"\n',
             y,ab[],ab[],IF ab[]>=32 THEN ab[] ELSE " ")
    CASE WORD
     stringf(s,'\r\z\z\z\z\z\h[6]: $\z\z\z\h[4] \d "\c\c"\n',
             y,aw[],aw[],IF ab[]>=32 THEN ab[] ELSE " ",IF ab[1]>=32 THEN ab[1] ELSE " ")
    CASE LOONG
     stringf(s,'\r\z\z\z\z\z\h[6]: $\z\z\z\z\z\z\z\h[8] \d "\c\c\c\c"\n',
             y,al[],al[],IF ab[]>=32 THEN ab[] ELSE " ",IF ab[1]>=32 THEN ab[1] ELSE " ",
             IF ab[2]>=32 THEN ab[2] ELSE " ",IF ab[3]>=32 THEN ab[3] ELSE " ")
    ENDSELECT
   ELSE
    y:=Shr(z,4)
    FOR x:=0 TO y
     stringf(s,'\r\z\z\z\z\z\h[6]: ',ab)
     putStr(s)
     FOR a:=0 TO 15 DO buf[a]:=IF ab[a] AND $7f>=32 THEN ab[a] ELSE "."
     buf[a]:=NIL
     SELECT x
     CASE BYTE
      stringf(s,'\z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] ',
              ab[]++,ab[]++,ab[]++,ab[]++,ab[]++,ab[]++,ab[]++,ab[]++)
      stringf(s,'\z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] \z\h[2] ',
              ab[]++,ab[]++,ab[]++,ab[]++,ab[]++,ab[]++,ab[]++,ab[]++)
     CASE WORD
      stringf(s,'\r\z\z\z\h[4] \z\z\z\h[4] \z\z\z\h[4] \z\z\z\h[4] ' +
                '\z\z\z\h[4] \z\z\z\h[4] \z\z\z\h[4] \z\z\z\h[4] ',
              aw[]++,aw[]++,aw[]++,aw[]++,aw[]++,aw[]++,aw[]++,aw[]++)
      ab:=aw
     CASE LOONG
      stringf(s,'\z\z\z\z\z\z\z\h[8] \z\z\z\z\z\z\z\h[8] \z\z\z\z\z\z\z\h[8] \z\z\z\z\z\z\z\h[8] ',
              al[]++,al[]++,al[]++,al[]++,al[]++,al[]++,al[]++,al[]++)
      ab:=al
     ENDSELECT
     putStr(s)
     putStr(buf)
     putS("\n")
     x:=getA()
     EXIT x=CTRL_C
     IF x>=32
      REPEAT
       x:=getS()
      UNTIL x=10 OR (x=13)
     ENDIF
    ENDFOR
   ENDIF
  ENDIF
 ENDIF
ENDPROC
-
PROC poke()
 DEF y:REG
 LowerStr(tl[1])
 y,z:=Val(tl[2])
 IF StrCmp(tl[1],'LED')
  MOVE.B y,LED
 ELSEIF StrCmp(tl[1],'MCR')
  MOVE.B y,MCR
 ELSEIF StrCmp(tl[1],'LFC')
  MOVE.B y,LFC
 ELSE
  IF tl[3] THEN x:=bwl(tl[3]) ELSE x:=BYTE
  IF x<>STR AND z
   ab:=aw:=al:=tl[1]
   SELECT x
   CASE BYTE  ; ab[]:=y
   CASE WORD  ; aw[]:=y
   CASE LOONG ; al[]:=y
   ENDSELECT
  ELSEIF x=STR
   x:=tl[2]
   WHILE x[]
    y[]++:=x[]++
   ENDWHILE
  ELSE
   putStr(invParams)
  ENDIF
 ENDIF
ENDPROC
-
PROC rename()
 IF prg:=findprg(tl[1])
  IF findprg(tl[2])
   putStr('Name already exists!\n')
  ELSEIF x:=allocMem(StrLen(tl[2]+1))
   AstrCopy(x,tl[2],ALL)
   freeMem(prg.name)
   prg.name:=x
  ELSE
   putStr('Could\at allocate memory for the new name!\n')
  ENDIF
 ELSE
  putStr(noEntry)
 ENDIF
ENDPROC
-
PROC run()
 IF prg:=findprg(tl[1])
  IF prg.type AND PT_CODE
   x:=prg.status
   SELECT x
   CASE PS_NEW
    IF tl[2] THEN y:=Val(tl[2]) ELSE y:=DEF_STACK
    IF y>=100
     prg.stack:=allocMem(y)
     IF prg.stack
      prg.pc:=prg.data
      prg.sr:=0
      prg.sp:=prg.stack+prg.stsize-4 -> 4=last RTS
      prg.sp[]:={finalReturn}
      prg.sigmask:=0
      prg.sigwait:=0
      prg.sigset:=0
      prg.status:=PS_READY
     ELSE
      putStr('Couldn\at allocate memory for the stack!\n')
     ENDIF
    ELSE
     putStr('Too small stack! An absolute minimum of 100 bytes is required.\n')
    ENDIF
   CASE PS_READY
    putStr('This program is already running.\n')
   CASE PS_RUNNING
    putStr('Unless you tried to run the terminal from itself ' +
           'there is a serious error in the system!\n')
   CASE PS_WAITING
    IF prg.sigwait=0
     putStr('This program is active but waiting forever.\n')
     IF prg.sigset
      putStr('It has signals set. Do you want to make it accept them? y/N: ')
      IF getS() OR 32="y"
       prg.sigwait:=prg.sigset
       prg.status:=PS_READY
      ENDIF
     ELSE
      putStr('Do you want to give it a CTRL-C signal? y/N: ')
      IF getS() OR 32="y"
       prg.sigwait:=SIG_CTRL_C
       prg.sigset:=prg.sigset OR SIG_CTRL_C
       prg.status:=PS_READY
      ENDIF
     ENDIF
    ELSE
     putStr('This program is already active and waiting for signals.\n')
    ENDIF
   CASE PS_SUSPENDED
    prg.status:=PS_READY
   CASE PS_FINISHED
   ENDSELECT
  ELSE
   putStr('This entry is not executable.\n')
  ENDIF
 ELSE
  putStr(noEntry)
 ENDIF
ENDPROC
-
PROC speed()
 DEF a, b, c, d, e
 a:=b:=c:=d:=e:=0
 WHILE tl[x]
  IF strCmpNC(tl[x],'MIPS')
   a:=TRUE
  ELSEIF strCmpNC(tl[x],'CPU')
   c,b:=Val(tl[x+1])
   IF c<=0 OR (c>5) AND b
    putStr(invParams)
    RETURN
   ENDIF
   b:=TRUE
  ELSEIF strCmpNC(tl[x],'SERIAL')
   IF tl[x+1]
    INC x
    d,e:=Val(tl[x])
    IF e<=0 OR (e>65535) AND d
     putStr(invParams)
     RETURN
    ELSEIF strCmpNC(tl[x],'SER_16X2400') ; e:=SER_16X2400
    ELSEIF strCmpNC(tl[x],'SER_16X4800') ; e:=SER_16X4800
    ELSEIF strCmpNC(tl[x],'SER_16X9600') ; e:=SER_16X9600
    ELSEIF strCmpNC(tl[x],'SER_16X19200') ; e:=SER_16X19200
    ELSEIF strCmpNC(tl[x],'SER_16X38400') ; e:=SER_16X38400
    ELSEIF strCmpNC(tl[x],'SER_16X57600') ; e:=SER_16X57600
    ELSEIF strCmpNC(tl[x],'SER_16X115200') ; e:=SER_16X115200
    ELSE
     e:=0
     DEC x
    ENDIF
   ELSE
    e:=0
   ENDIF
   d:=TRUE
  ENDIF
 ENDWHILE
 IF a
  putStr('MIPS meassurement is currently not implemented.\n')
 ENDIF
 IF b AND c
  SELECT c
  CASE 6
   putStr('This requires an external clock signal to be connected ' +
          'to the clock card. Do you want to proceed? y/N: ')
   IF getS() OR 32="y"
    envvars.mcr:=envvars.mcr AND $1f OR $82
    ab:=MCR
    ab[]:=envvars.mcr
    putStr('\nThe clock speed has been set to "external".')
   ENDIF
   putS("\n")
  CASE 5
   putStr('This will set the CPU to 20MHz which may cause it to hang.\n' +
          'Do you want to proceed? y/N: ')
   IF getS() OR 32="y"
    envvars.mcr:=envvars.mcr AND $1f OR $84
    ab:=MCR
    ab[]:=envvars.mcr
    putStr('\nThe clock speed has been set to 20MHz.')
   ENDIF
   putS("\n")
  DEFAULT
   envvars.mcr:=envvars.mcr AND $1f OR Shl(c-1,5)
   ab:=MCR
   ab[]:=envvars.mcr
   putStr('A new CPU clock speed has been set.\n')
  ENDSELECT
 ELSEIF d=0
  b:=Shr(envvars.mcr,5)
  IF b=4 OR (b=7) THEN b:=0
  IF b=6 THEN b:=4
  al:=['1.25','2.5','5','10','20','Unknown/external ']:LONG
  stringf(s,'The current speed of the CPU is \b (\sMHz)\n',b+1,al[b])
  putStr(s)
  b:=0
 ENDIF
 IF d
  IF e
   putStr('The serial port will now be set to the new speed.\n' +
          'Press Enter when you are done.\n')
   MOVE.L SYSLIST,A6
   JSR    FFlushTx(A6)
   MOVE.L e,D0
   JSR    FSetSerSpeed(A6)
   REPEAT
   UNTIL getS()="\n"
   putStr('If you see this text it means the new speed settings work.\n')
  ELSE
   putStr('Calculate the 16X divisor value like this: divisor=960000/bps.\n' +
          'Predefined speeds (divisor values) that can be used are:\n' +
          '"SER_16X2400" for 2400bps\n' +
          '"SER_16X4800" for 4800bps\n' +
          '"SER_16X9600" for 9600bps\n' +
          '"SER_16X19200" for 19200bps\n' +
          '"SER_16X38400" for 38400bps\n' +
          '"SER_16X57600" for 57600bps (this may not work as the value is not exact)\n' +
          '"SER_16X115200" for 115200bps (this may not work as the value is not exact)\n')
  ENDIF
 ELSEIF b=0
  a:=[SI_SPEED,{x},NIL]:LONG
  MOVE.L a,A0
  MOVE.L SYSLIST,A6
  JSR    FGetSerInfo(A6)
  SELECT x
  CASE SER_16X2400 ; a:='2400'
  CASE SER_16X4800 ; a:='4800'
  CASE SER_16X9600 ; a:='9600'
  CASE SER_16X19200 ; a:='19200'
  CASE SER_16X38400 ; a:='38400'
  CASE SER_16X57600 ; a:='57600'
  CASE SER_16X115200 ; a:='115200'
  DEFAULT
   a:=0
  ENDSELECT
  IF a
   stringf(s,'The speed of the serial port is set to \sbps which equals ' +
             'a 16X divisor of \d @ 15.360MHz.\n',a,x)
  ELSE
   stringf(s,'The serial port is currently set to a custom speed with a ' +
             '16X divisor of \d which equals ' +
             'a speed of \dbps @ 15.360MHz.\n',x,Div(960000,x))
  ENDIF
 ENDIF
ENDPROC
-
PROC stop()
 IF prg:=findprg(tl[1])
  IF prg.type AND PT_CODE
   IF tl[2]
    UpperStr(tl[2])
    IF StrCmp(tl[2],'FORCE') THEN prg.status:=PS_SUSPENDED ELSE putStr('Illegal parameter!\n')
   ELSE
    prg.sigset OR SIG_CTRL_C
   ENDIF
  ELSE
   putStr('Can not stop a script or data.\n')
  ENDIF
 ELSE
  putStr(noEntry)
 ENDIF
ENDPROC
-
/************************ ***************** *************************
************************ support and system *************************
************************ ***************** ************************/
-
PROC install()
#ifdef EXOS
 MOVEA.L SYSLIST,A6
 JSR     FGetEnvToA6(A6)
 MOVE.L  A6,envvars
#endif
#ifdef AMIGA
 NEW envvars
 PutLong({syslist},{syslist})
 WriteF('syslist=$\h\n',{syslist})
#endif
 FOR x:=0 TO 0
  IF envvars.kernel=0
   y:=allocMem(SIZEOF kernel)
   envvars.kernel:=y
  ENDIF
  EXIT envvars.kernel=NIL
#ifdef AMIGA
  envvars.kernel.sspref:=sspref
#endif
#ifdef EXOS
  envvars.kernel.sspref:=RAM+MEMSIZE
#endif
  envvars.kernel.status:=0
  envvars.kernel.qcnt:=DEFQUANTUM
  envvars.kernel.quantum:=DEFQUANTUM
  prg:=allocMem(SIZEOF program)
  EXIT prg=NIL
  prg.next:=NIL
  prg.data:={main}
  prg.size:=MYSELF
  prg.type:=PT_PROTCODE
  prg.status:=PS_RUNNING
  prg.stack:=RAM+MEMSIZE-SSPSIZE-USPSIZE
  prg.stsize:=USPSIZE
  prg.sigmask:=0
  prg.sigwait:=0
  prg.sigset:=0
  prg.name:='Terminal'
  envvars.pcb:=prg
  envvars.kernel.curpcb:=prg
 ENDFOR
 IF x=0
  freeMem(envvars.kernel)
  freeMem(envvars.pcb)
 ENDIF
ENDPROC x
-
PROC tokenize()
 x:=0
 FOR y:=0 TO MAXTOKEN
  tl[y]:=NIL
  EXIT buf[x]=NIL
  WHILE buf[x]=" " OR (buf[x]="\t") OR (buf[x]="\n") OR (buf[x]=13) AND buf[x] DO INC x
  IF buf[x]
   IF buf[x]="\q"
    INC x
    tl[y]:=buf+x
    WHILE buf[x]<>"\q" AND buf[x] DO INC x
    IF y<MAXTOKEN AND buf[x]
     buf[x]:=NIL
     INC x
    ENDIF
   ELSE
    tl[y]:=buf+x
    WHILE buf[x]<>" " AND (buf[x]<>"\t") AND buf[x] DO INC x
    IF y<MAXTOKEN AND buf[x]
     buf[x]:=NIL
     INC x
    ENDIF
   ENDIF
  ENDIF
 ENDFOR
ENDPROC y
-
PROC bwl(st)
 IF st
  UpperStr(st)
  IF StrCmp(st,'BYTE') ; RETURN BYTE
  ELSEIF StrCmp(st,'WORD') ; RETURN WORD
  ELSEIF StrCmp(st,'LOONG') ; RETURN LOONG
  ELSEIF StrCmp(st,'STR') ; RETURN STR
  ENDIF
 ENDIF
ENDPROC 0
-
PROC findprg(n)
 DEF pcb:PTR TO program
 pcb:=envvars.pcb
 WHILE pcb
  EXIT strCmpNC(pcb.name,n)
  pcb:=pcb.next
 ENDWHILE
ENDPROC pcb
-
PROC strCmpNC(s,t,strlen=-1)
 DEF x, y, z, a, b
 IF strlen=0 THEN RETURN TRUE
 z:=Min(StrLen(s),StrLen(t))
 IF strlen>0 THEN z:=Min(strlen-1,z)
 FOR x:=0 TO z
  a:=s[x]
  b:=t[x]
  IF a>=65 AND (a<=90) THEN a:=a OR 32
  IF b>=65 AND (b<=90) THEN b:=b OR 32
  EXIT y:=a<>b
 ENDFOR
ENDPROC y=0
-
PROC allocStr(z)
 DEF x:PTR TO INT
 IF x:=allocMem(z+5)
  x[0]:=z
  x[1]:=0
  x[2]:=0
  z:=x+4
  z[x[0]]:=0
  RETURN z
 ENDIF
ENDPROC NIL
-
PROC freeStr(z) IS freeMem(z-4)
-
#ifdef AMIGA
-
PROC allocMem(z) IS New(z)
PROC freeMem(z) IS Dispose(z)
PROC putStr(s) IS WriteF('\s',s)
PROC sendS(a,z) IS Write(stdout,a,z)
PROC putS(x) IS WriteF('\c',x)
-
PROC forbid()
 MOVEQ  #TRUE,D0
 MOVE.L SYSLIST,A6
 JSR    FSingleTask(A6)
ENDPROC
-
PROC permit()
 MOVEQ  #FALSE,D0
 MOVE.L SYSLIST,A6
 JSR    FSingleTask(A6)
ENDPROC
-
PROC getA()
 DEF ms:PTR TO intuimessage
 IF ms:=GetMsg(w.userport)
```

```
   x:=ms.code
   ReplyMsg(ms)
   RETURN x
  ENDIF
ENDPROC -1
-
PROC readS(a,z)
 WHILE z
  a[]++:=getS()
  DEC z
 ENDWHILE
ENDPROC
-
PROC getS()
 DEF x
 WHILE (x:=WaitIMessage(w))=$400
  x:=MsgCode()
  IF x>=80 AND (x<=89) -> using F-keys in the input window for TRAPs :)
   x:=x-80
   SELECT x
   CASE 0 ; TRAP #0
   CASE 1 ; TRAP #1
   CASE 2 ; TRAP #2
   CASE 3 ; TRAP #3
   CASE 4 ; TRAP #4
   CASE 5 ; TRAP #5
   CASE 6 ; TRAP #6
   CASE 7 ; TRAP #7
   CASE 8 ; TRAP #8
   CASE 9 ; TRAP #9
   ENDSELECT
  ENDIF
 ENDWHILE
 SELECT x
 CASE $200
  CloseW(w)
  CleanUp()
 CASE $200000
  lastchar:=-2
  RETURN MsgCode()
 ENDSELECT
ENDPROC
-
trapcode:
 SUBI.L #32,(A7)
 ADDQ.L #4,A7
 CMPI.L #0,-4(A7)
 BNE.S  tcNot0
  MOVE.L  D0,x
  MOVEM.L D0-D7/A1-A6,-(A7)
  x:=allocMem(x)
  MOVEA.L x,A0
  MOVEM.L (A7)+,D0-D7/A1-A6
  RTE
tcNot0:
 CMPI.L #1,-4(A7)
 BNE.S  tcNot1
  MOVE.L  A0,x
  MOVEM.L D0-D7/A0-A6,-(A7)
  freeMem(x)
  MOVEM.L (A7)+,D0-D7/A0-A6
  RTE
tcNot1:
 CMPI.L #3,-4(A7)
 BNE.S  tcNot3
  JMP   tReSchedule(PC)
tcNot3:
 CMPI.L #4,-4(A7)
 BNE.S  tcNot4
  JMP   tStop(PC)
tcNot4:
 CMPI.L #6,-4(A7)
 BNE.S  tcNot6
  JMP   tDebug(PC)
tcNot6:
 CMPI.L #7,-4(A7)
 BNE.S  tcNot7
  JMP   tUserTrap(PC)
tcNot7:
 CMPI.L #8,-4(A7)
 BNE.S  tcNot8
  JMP   incLed(PC)
tcNot8:
 CMPI.L #9,-4(A7)
 BNE.S  tcNot9
  JMP   int3(PC)
tcNot9:
 RTE
-
tReSchedule:
tStop:
tDebug:
tUserTrap:
int3:
RTE
-
delay:
event:
connect:
initSerial:
setSerSpeed:
getSerInfo:
_sendS:
_sendA:
_getS:
_getA:
_readS:
```

```
flushTx:
flushRx:
strCmp:
_strCmpNC:
strLen:
strCopy:
str2Int:
int2Dec:
compFreeList:
memInfo:
memCheck:
_storeProg:
_singleTask:
signal:
block:
freeSignal:
allocSignal:
RTS
-
_putStr:
 MOVEM.L D0-D7/A0-A6,-(A7)
 MOVE.L  A0,x
 putStr(x)
 MOVEM.L (A7)+,D0-D7/A0-A6
RTS
-
_putS:
 MOVEM.L D0-D7/A0-A6,-(A7)
 MOVE.L  D0,x
 putS(x)
 MOVEM.L (A7)+,D0-D7/A0-A6
RTS
-
int2Hex:
 MOVEM.L D0-D2/A0,-(A7)
 MOVE.L  D0,D1
 MOVEQ   #7,D2
i2hLoop:
 ROL.L   #4,D1
 MOVE.B  D1,D0
 ANDI.B  #$f,D0
 CMPI.B  #9,D0
 BGT.S   i2hAPlus
  ADDI.B #"0",D0
  BRA.S  i2hDone
i2hAPlus:
  ADDI.B #"A"-10,D0
i2hDone:
  CMPA.L #0,A0
  BEQ.S  i2hSerial
   MOVE.B D0,(A0)+
   DBRA   D2,i2hLoop
i2hSerial:
  BSR    _putS
 DBRA    D2,i2hLoop
 MOVEM.L (A7)+,D0-D2/A0
 RTS
-
incLed:
 BCHG.B #1,$BFE001
RTS
-
getEnvToA6:
 MOVE.L envvars,A6
RTS
-
 JMP delay(PC)          ; NOP
 JMP event(PC)          ; NOP
 JMP connect(PC)        ; NOP
 JMP initSerial(PC)     ; NOP
 JMP setSerSpeed(PC)    ; NOP
 JMP getSerInfo(PC)     ; NOP
 JMP _sendS(PC)          ; NOP
 JMP _sendA(PC)          ; NOP
 JMP _getS(PC)           ; NOP
 JMP _getA(PC)           ; NOP
 JMP _putS(PC)           ; NOP
 JMP _readS(PC)          ; NOP
 JMP flushTx(PC)        ; NOP
 JMP flushRx(PC)        ; NOP
 JMP _putStr(PC)         ; NOP
 JMP strCmp(PC)         ; NOP
 JMP _strCmpNC(PC)       ; NOP
 JMP strLen(PC)         ; NOP
 JMP strCopy(PC)        ; NOP
 JMP str2Int(PC)        ; NOP
 JMP int2Dec(PC)        ; NOP
 JMP int2Hex(PC)        ; NOP
 JMP compFreeList(PC)   ; NOP
 JMP memInfo(PC)        ; NOP
 JMP memCheck(PC)       ; NOP
 JMP _storeProg(PC)      ; NOP
 JMP getEnvToA6(PC)     ; NOP
 JMP incLed(PC)         ; NOP
 JMP _singleTask(PC)     ; NOP
 JMP signal(PC)         ; NOP
 JMP block(PC)          ; NOP
 JMP freeSignal(PC)     ; NOP
 JMP allocSignal(PC)    ; NOP
syslist:
 LONG $DEADBEEF
```

```
-
#endif
-
#ifdef EXOS
_a4: LONG 0
-
PROC allocMem(z)
 MOVE.L z,D0
 TRAP   #TAllocMem
 MOVE.L A0,z
ENDPROC z
-
PROC freeMem(z)
 MOVE.L z,A0
 TRAP   #TFreeMem
ENDPROC
-
PROC forbid()
 MOVEQ  #TRUE,D0
 MOVE.L SYSLIST,A6
 JSR    FSingleTask(A6)
ENDPROC
-
PROC permit()
 MOVEQ  #FALSE,D0
 MOVE.L SYSLIST,A6
 JSR    FSingleTask(A6)
ENDPROC
-
PROC putStr(s)
 MOVE.L s,A0
 MOVE.L SYSLIST,A6
 JSR    FPutStr(A6)
ENDPROC
-
PROC sendS(a,z)
 MOVE.L a,A0
 MOVE.L z,D0
 MOVE.L SYSLIST,A6
 JSR    FSendS(A6)
ENDPROC
-
PROC readS(a,z)
 MOVE.L a,A0
 MOVE.L z,D0
 MOVE.L SYSLIST,A6
 JSR    FReadS(A6)
ENDPROC
-
PROC getS()
 MOVE.L SYSLIST,A6
 JSR    FGetS(A6)
ENDPROC D0
-
PROC getA()
 MOVE.L SYSLIST,A6
 JSR    FGetA(A6)
ENDPROC D0
-
PROC putS(x)
 MOVE.L x,D0
 MOVE.L SYSLIST,A6
 JSR    FPutS(A6)
ENDPROC
-
#endif
-
PROC inithelp()
 x:=0
 help[x++].h:='"clear <entry>": If <entry> is a program its stack will\n' +
              'be deallocated, program counter reset, signals deallocated\n' +
              'and its status set to "New". If <entry> is a data block its\n' +
              'contents will be set to zero. Scripts cannot be cleared.'
-
 help[x++].h:='"cls": Clear screen. This command is supposed to clear\n' +
              'the shell in the other end. This command is equal to\n' +
              '"echo $9b489b4a LONG".'
-
 help[x++].h:='"delete <entry>": Remove an entry from the list.'
-
 help[x++].h:='"echo <data|ON|OFF> [BYTE|WORD|LONG]": Echoes the binary\n' +
              'value of <data> directly back to the serial port. Practical\n' +
              'if you need to see how the shell in the other end reacts to\n' +
              'a specific byte or set of bytes.\n' +
              '<data> is a number of size BYTE, WORD or LONG.\n' +
              'The keywords ON or OFF can be used instead of a value to\n' +
              'turn global echo on or off. In global echo mode everything\n' +
              'written will be echoed back as hex numbers instead of the\n' +
              'actual text typed in. Practical if you want to know exactly\n' +
              'what is typed in. Commands will be parsed as usual,\n' +
              'including "echo".'
-
 help[x++].h:='"exit": Exit the terminal. You will most likely end up in\n' +
              'the primitive console. No global resources like the list of\n' +
              'programs and data will be freed so if the terminal is\n' +
              'restarted it will resume management of its data where it\n' +
              'left off.'
-
 help[x].h:=  '"help|? [command]": Displays this text. Type "help >command<"\n' +
              'to get help about a specific command. [option] means that\n' +
              'this is an optional parameter. <param> means that this is a\n' +
              'mandatory parameter (not to be confused with >this< which\n' +
              'is only an emphasis of a non-keyword option in the text.\n\n'
 help[x++].h2:='UPPERCASE means that it is a keyword that should litterary\n' +
              'be in the command line. lowercase means that it should be\n' +
              'substituted with a name or number.\n' +
              'Numbers can be either decimal, $hexadecimal or %binary.\n' +
              'BYTE means 8 bits, WORD means 16 bits, LONG means 32 bits,\n' +
              'STR means a string of characters than can be of variable\n' +
              'length depending on if it is used as an input or output.\n' +
              '<parameters> may not be swapped around but need to be in\n' +
              'the specific order the command template shows.\n' +
              '[KEYWORDS] may be swapped with other [KEYWORDS] but not\n' +
              'with [options].'
_
 help[x++].h:='"list [entry]": List all programs and data blocks currently\n' +
              'in memory and their status. If the parameter >entry< is\n' +
              'given only this entry will be listed. In this case more\n' +
              'information will be given too.'
_
 help[x].h:=   '"load [S19|BINARY] [PROGRAM|DATA|SCRIPT] [name]": Put the\n' +
              'terminal in a ready mode to accept a load command.\n' +
              'Option S19 is default and makes "load" wait for an "S"\n' +
              'which signals the first record of an S19 file.\n' +
              'Option BINARY will make load wait for 4 bytes of binary\n' +
              'length and then this number of binary bytes to be\n' +
              'downloaded. Be careful not to enter text when the four\n' +
              'size-bytes are requested as this will result in a very big\n' +
              '32-bit number and the terminal will not stop reading until\n' +
              'it has received this very large amount of data!\n\n'
 help[x++].h2:='The option PROGRAM is default and will make "load" treat\n' +
              'the downloaded data as an executable program. If the\n' +
              'option DATA is used, the loaded file will be treated as\n' +
              'non-executable data usable by some other program.\n' +
              'The option SCRIPT will make the terminal treat the data as\n' +
              'a shell script executable by the terminal.\n\n' +
              'The option >name< will give a name to the data, otherwise\n'+
              'a default Pnum, Dnum or Snum name will be given.'
_
 help[x++].h:='"meminfo [MEMHDL] [VERBOSE] [COMPRESS] [CHECK]": Display\n' +
              'memory information. Without any parameter "meminfo" will\n' +
              'display the amount of free memory and the largest free\n' +
              'block. With the parameter MEMHDL "meminfo" will display\n'+
              'details about the memory handler, such as how many\n'+
              'allocation have been made and how many more can be made\n'+
              'using the current buffer. Also the address of the buffer.\n'+
              'If VERBOSE is added a complete list of the memory buffer\n'+
              'will be printed.\n' +
              'The keyword COMPRESS will cause the memory handler to\n' +
              'attemt to optimize the free list by grouping all adjacent\n' +
              'free blocks together. If COMPRESS is given together with\n' +
              'any other keyword the compression will be done first.\n' +
              'CHECK will perform a memory check and tell you about eny\n' +
              'possibly physical errors found.'
_
 help[x++].h:='"peek <address> [BYTE|WORD|LONG|STR] [numbytes]": Look at\n' +
              'the contents of a memory address. Specify the format you\n' +
              'want the contents displayed in using BYTE, WORD or LONG.\n' +
              'BYTE is default. Use >numbytes< to specify how many number\n' +
              'of bytes you want to display rounded up to the nearest word\n' +
              'size. When >numbytes< is not specified one word will be\n' +
              'displayed in all number systems, else a hex display with\n' +
              'addresses will be displayed. <address> can also be one of\n' +
              'the keywords DIP or MCR to make it easier to address those\n' +
              'registers on the motherboard. Size and >numbytes< will be\n' +
              'ignored and BYTE is the word size that will be used.'
_
 help[x++].h:='"poke <address> <data> [BYTE|WORD|LONG|STR]": Change a word\n' +
              'in the memory space. Use one of the keywords BYTE, WORD,\n' +
              'LONG or STR to specify the word size of >data<. BYTE is\n' +
              'default. If STR is used as word size the actual length of\n' +
              'the string typed will be written to the address given\n' +
              'excluding a null-termination.\n' +
              '>address< can also be one of the keywords LED, MCR or LFC\n' +
              'to make it easier to write to the corrsponding specific\n' +
              'motherboard registers. Size and >numbytes< will then be\n' +
              'ignored and BYTE is the word size that will be used.'
_
 help[x++].h:='"rename <oldname> <newname>": Rename an entry in the list.'
_
 help[x++].h:='"run <programname> [stack]": Start the execution of a\n' +
              'program that has not yet been executed, has been stopped or\n' +
              'finished executing. The option >stack< is a positive number\n'+
              'used as stack size if you want to override the default 1kB\n'+
              'of stackspace. This option can not be used when restarting\n' +
              'a stopped program, only for new and finished programs.'
_
 help[x].h:=   '"speed [MIPS] [CPU [value]] [SERIAL [div]]": Display and/or\n' +
              'set the speed of the CPU and the serial port or meassure\n' +
              'the speed of the CPU in Million Instructions Per Second.\n\n' +
              'Without any arguments "speed" will display the current\n' +
              'speed settings for the CPU and the serial port. If the\n' +
              'keyword MIPS is used the performance of the CPU will be\n' +
              'meassured. With the keyword CPU the speed of the CPU clock\n' +
              'can be set by giving it a value between 1 and 6 where 5 is\n' +
              'the fastest, 20MHz and 6 is an external clock.\n\n'
 help[x++].h2:='The keyword SERIAL will allow you to set the speed of the\n' +
              'serial port. The speed is set by specifying a divisor value\n' +
              'that is equal to 15360000/16/<your preferred speed>. You\n' +
              'can also use one of the predefined constants that will be\n' +
              'printed if you leave out the >div< parameter.'
-
 help[x++].h:='"stop <programname> [FORCE]": Stop a running program.\n' +
              'Use FORCE to halt a program that doesn\at respond to Ctrl-C.\n' +
              'A program stopped with FORCE can be restarted at the exact\n' +
              'point it was stopped by using run again. Without FORCE\n' +
              '"stop" will just send a Ctrl-C signal to the process.'
ENDPROC
-
finalReturn:
 MOVE.L  A6,-(A7)
 MOVE.L  4,A6
 JSR     FGetEnvToA6(A6)
 MOVEA.L ENV_KERNEL(A6),A6
 BSET.B  #KS_INHIBIT,KRN_STATUS(A6)
 MOVEA.L (A6),A6
 MOVE.B  #PS_FINISHED,PCB_STATUS(A6)
 MOVE.L  (A7)+,A6
 TRAP    #TReSchedule
 RTS ->The same process will never return
```