Documentation

of



Version 1.1b

Oktober 19, 1998

Sunrise for MSX

License Agreement

Warranty Notice

Sunrise for MSX makes no warranty of any kind. In no event shall Sunrise for MSX be liable for any errors contained herein or for consequential damages with the use of WiOS or the documentation.

The information given in this document is subject to change without notice.

Copyright Notice

This document contains information which is only available for Alpha-Testers chosen by Sunrise for MSX. No part of this Alpha-Release (that includes WiOS, the Application Creation Toolkit incl. MAKDRV.COM and this document) may be copied, reproduced or translated without the written prior of Sunrise for MSX.

Any violation against the copyright results in exclusion from updates and information!

Author Credits

WiOS and this documentation was written by Michael Stellmann

Other Credits

Thanks to Erik Maas and Henrik Gilvad who provided assistance in answering questions about the MSX-interrupt handler.

Also thanks to Peter Burkhard for mental support.

Marcel Trütsch for printing.

Anyone else who gave assistance and waited that long for this version.

Trademarks

Any products mentioned in this documentation are trademarks, registered trademarks, service marks, or registered service marks of their respective companies or organizations.

Agreement

With the use of this documentation, WiOS or any WiOS-related utility from the Application Creation Toolkit you agree to these conditions mentioned above.

Table Of Contents

1 INT	RODUCTION TO WIOS	.3
1.1	ABOUT THIS MANUAL	. 3
1.2	ABOUT THIS VERSION	. 3
2 INS	TALLATION	. 4
2.1	REQUIRED EQUIPMENT	1
	PACKING LIST	
	INSTALLATION INSTRUCTIONS	
2.3.1	ASCII-C installation	
2.3.2	WiOS installation	. 4
3 PRC	OGRAMMING	.5
3.1	INTRODUCTION	5
3.1	PROGRAM STRUCTURE	
3.3	MULTITASKING	
3.3.1	Pre-emptive Multitasking	
3.3.2	Co-operative Multitasking	
3.3.3	Polling and Problems	. 8
3.4	MEMORY STRUCTURE	
3.4.1	Internal WiOS-part	
3.4.2	Code- and Stack-Segment	
3.4.3	Data-Segment	
3.4.4 3.5	Global Data Area ('GDA') DRIVERS	
3.6	HANDLES	
3.7	WINDOW INTERFACE	
3.7.1	Icons	
3.7.2	Parameters	12
3.7.3	Window Management	13
<i>3.7.4</i>	Window Types	
	EVENTS - COMMUNICATION BETWEEN WIOS AND TASKS	
	USER INPUT	
3.9.1 3.9.2	Mouse TriggersKey Presses	
	DRAG & DROP	
3.10.1		
3.10.2	3 /	
3.11	Programming Language	
3.12	C SPECIFICATION	20
3.12.1		
3.12.2		
3.12.3	= · · · · · · · · · · · · · · · · · · ·	
3.13	BASICS	
3.13.1 3.13.2	Accessing the Global Data Area	
3.13.2	,	
3.13.4	•	
3.13.5		
3.14	FUNCTION REFERENCE (WIOS DRIVERS)	
3.14.1	External Driver	
3.14.2	· · · · · · · · · · · · · · · · · · ·	
3.14.3	- · r - · · · · · · · · · · · · · · · ·	
3.14.4	- · r	
3.14.5	,	
3.14.6		
3.14.7 3.14.8		
3.14		
3.14		

WOS - Application Programming Documentation

3.14.9	Direct Calls	. 45
3.15 EVE	NT REFERENCE	.46
3.15.1	Event Block Structure	. 46
3.15.2	Event Definition	. 46
3.16 DAT	A EXCHANGE SPECIFICATION	.49
3.16.1	Execution of a data exchange	
3.16.2	Data Type Definition	
3.16.2.1		
3.16.2.2	Data sent with the 'E_USERMSG' event	51
3.17 GD.	A Reference	. 52
3.18 ME	NU REFERENCE	. 54
3.18.1	Menu types	. 54
3.18.2	Menu Features	. 55
3.18.3	General conventions about menus	56
3.18.4	Menu-Block Structure	. 57
3.19 PRO	GRAMMING TIPS	
3.19.1	Windows	
3.19.1.1	Opening Windows	
3.19.1.2	Redrawing Windows	
3.19.1.3	Scroll Requests	62
3.19.2	User Input	. 63
3.19.3	Using the Temporary Segment	63
3.19.4	Mouse Pointer	
3.19.5	Sending data to other program parts using the address block	64
3.19.6	Stacks	64
3.19.7	Program Termination	. 64
3.20 COM	MPILATION	. 65
3.20.1	Single Part Applications (up to 16k code)	
3.20.2	Multi Part Applications (more than 16k of code)	
3.21 TAS	K SPECIFICATIONS	
3.21.1	Headers	
3.21.2	Standard Header Structure	
3.21.3	Extended Header Structure	
3.22 SAM	IPLE PROGRAMS	
3.22.1	Pre-processor Directives	
3.22.2	Header Files	
3.22.3	External Variables	
3.22.4	Task Initialization	
3.22.5	Main Routine	
3.22.6	Window Creation	
3.22.7	Polling and Event Handling	
-	URITY	
	RTING TASKS WITH THE ALPHA-RELEASE	
	BLEMS, BUGS AND OTHER UNWELCOME STUFF	
3.25.1	Known Problems	
3.25.2	Planned Extensions	
4 FAQ - F	REQUENTLY ASKED QUESTIONS	.78
5 KEVWA	ORD DESCRIPTION	70
5 KEYWO	JRD DESCRIF HUN	. 19
6 LAST R	UT NOT LEAST	80

Introduction to WiOS

WiOS is a graphic-based, multitasking operating system for the MSX and the V9990 videochip, which is on the Graphics 9000 and the Yamaha E-VDP III Evaluation Board.

WiOS is a collection of routines that support the programming of applications in a graphical environment. Currently, WiOS allows a shared number of up to 252 drivers or tasks (which is a synonym for 'application'), the management of 3072 memory-segments and 256 windows.

1.1 About This Manual

This Manual is divided into 5 main chapters. Although my intention was to write a perfect documentation, with which every programmer should come from the understanding of why several parts of WiOS are as they are to the creation of a good application with this documentation as a valuable reference book, I'm almost sure that there are sections which you will not understand, not enough details in the description of complex structures and maybe even gaps in essential parts. Due to a lack of time (well, actually my meaning of life is not WiOS, although one could believe that, regarding the time and the effort WiOS took and will still take) and users (the next MSX user lives about 400 km away), this documentation has neither been read nor was it ever used for programming. If there are parts you don't understand or even missing, please inform us. We can only help you if we know what you need. Don't hesitate to write constructive criticism.

1.2 About This Version

This Alpha-Release is not bug-free. It is not optimized for speed and it is not tested with other computers, other hardware, resistant programs and it has not been tested with any application doing more than just using a few of the functions - just to test them. Each function has been checked carefully in single-steps, but not in co-operation in a bigger task. The only purpose of this version is to speed up the debugging. Only Alpha-Testers chosen by Sunrise for MSX are given this version under the condition of helping to find and reporting bugs.

This is NO public version and, on the contrary of the Beta-Version, NOT free.

2 Installation

2.1 Required Equipment

As a minimum you must have the following hardware and software to run this WiOS Alpha-Release.

Hardware An MSX turbo R is strongly recommended for getting an acceptable speed. A mouse is needed to be able to move the cursor.

Display A V9990-based graphic card at port 60h.

Memory At least 160k of free memory.

To make bigger applications and use memory for temporary data storage, memory mappers should be added. About 300k of free memory is good.

Disk For good programming, a harddisk is required.

Operating System MSX-DOS 2 or higher.

Language ASCII-C v1.2 is advisable, due to the fact that the function calls must have the same structure.

2.2 Packing List

See the READ.ME file on the Application Creation Toolkit for complete packing list

2.3 Installation Instructions

2.3.1 ASCII-C installation

There is no need to have ASCII-C in the same directory than WiOS.

To find the compiler files, you may either specify the directory in the PATH environment variable or put the complete drive and path before the COM-files in the compilation batch file. To find the standard C libraries outside of the current directory, include them with #include <heatfile.h>

and set up the INCLUDE environment variable with the drive and directory of the header files.

2.3.2 WiOS installation

Make a directory of your choice on a drive on your harddisk and copy all the files from the directory WiOS on the disk to the harddisk.

3 Programming

3.1 Introduction

If you have never worked with WiOS (and you sure did never work with WiOS;) it's important to read this section completely to better understand the basic structure of how WiOS works. This is not a detailed information of how anything is done internally, but a raw outline of what you need to know before you are going to make programs for WiOS.

The introduction contains the following parts:

```
Structure
```

Basic differences

Multitasking

Pre-emptive Multitasking Co-operative Multitasking Polling and Problems

Memory Structure

Internal WiOS memory

Task-segment Data-segment Global Data Area

Drivers

Difference: Task <-> Driver

Handles

Programs Graphic

Window Interface

Icons

Parameters

Window Management

Window Types

Events

Communication between WiOS and Tasks

User Input

Mouse

Keyboard

Drag & Drop

How to simulate non-existing parts in the Alpha-Release

Programming Language

C!

C?

C!!!

C specification

How do I come from the C source to an executable COM-file

Limits of C

Breaking the limits of C

3.2 Program Structure

Programming applications for WiOS is different from the way you are used to structure programs. The main difference - and that is why the structure is different - is the fact that a program is not any more the one and only thing the computer has to process - you are now in a multitasking system.

Single tasks normally work like this:

function1

calculate_something program is busy for a while

wait_for_user_input user-input function is repeatedly called until a certain key is pressed or the mouse was clicked or time elapsed or...

calculate something else let the user wait until this is done

wait for next user input

call_another_function or loop_something

exit well, finally return to DOS

Most programmers should be familiar with this linear structure.

Also with the process of what is happening when subroutines are called: The return-address is saved on the stack, and is restored when the subroutine returns. If other subroutines are called from subroutines, the stack will be filled with the returnaddresses and restored in the reversal order as they are stacked - i.e. the last address will be restored first.

To execute a second program parallel to 'function1' is a bit difficult. One way might be to prepare a special subroutine ('function2') which is some kind of loop that has a countervariable and returns every time it is called for the next loop. To achieve that, we would have to call this subroutine in many places in our program.

This loop-routine must be called from 'calculate something', 'wait for user input', 'calculate_something_else', 'wait_for_next_user_input' and 'call_another_function'. Then we have one 'active' function that calls a calculation-routine permanently - it seems as if the second task would be executed at the same time as the other function.

Our sample is not very effective, since the second task can be only one fixed function which has to be prepared for this special type of 'loop' (check counter, loop once and return). Also we cannot run a third task. There would have to be a table which indicates the number of tasks to call from 'function1'. But, what if we stop and remove 'function1' from memory? We would have to call the other tasks from 'function2'. Since it is only a special loop-routine, it is almost impossible.

3.3 Multitasking

Now we are going to examine what types of multitasking exists.

3.3.1 Pre-emptive Multitasking

Every application gets a certain amount of time. The control is given and taken from the system. One major problem is to preserve the status of what an application did when the control was taken. A simple demonstration: One task prints a text at the upper left, a second task at the lower right corner. The first task sets the text-pointer to its corner and starts printing. Now the system interrupts task 1 before is could print anything and task 2 starts. It sets the text-pointer to the lower right and starts printing. Again, the system interrupts and task 1 continues. Since task 1 has been interrupted, it could not be informed of the fact that task 2 was called. So it continues with the text-output. Since the text-pointer is still at the lower right corner, task 1 is unable to finish its text-output on the correct corner, and, even worse, corrupts the output of task 2.

Consequently actions must be done to either let the system preserve the text-pointer, or to let the task check a status-flag before every character output if it has been interrupted, and if yes, set the text-pointer to the new direction. That requires a text-pointer-counter for the task or a table of global variables for each task where the system can save the text-pointers. Since text-pointers are not the only thing one task might corrupt the actions of another task, you need huge tables for storing and restoring global settings, or you slow down the execution time of your task drastically with counters and flag-checking. Also, it's not easy to send a message to a task like 'user has moved your window, redraw!!!', or 'here comes a picture of another task'. So more status-flags must be used and checked. In every case, preemptive multitasking needs a very powerful processor and is very memory-consuming - and very hard to program.

3.3.2 Co-operative Multitasking

With this type, the speed of the system depends on the co-operation of the tasks. Every task has the total control of the execution time. The way co-operative multitasking works is a little bit of what we have done in the upper example, and a bit of something completely different. To offer the possibility of other tasks being able to do something in 'parallel', each task must leave the control somewhere in the program. This is the 'calling' of 'function2' in the example, with the difference that it's not 'function2' which is called but a routine whose position is fixed and independent of which tasks are running. This function is not a real 'subroutine', although, from the task's point of view, is does look like if it would be, because the task calls it and expects the function to return like from a normal subroutine. This unique function is called the 'Poll-Routine', and the process of calling it is called 'Polling'. The objective of the poll-routine is to 'return' to other tasks that have polled - one after each other.

WiOS uses co-operative multitasking - although it has to deal with similar problems like preemptive multitasking, it's easier, faster and more memory-preserving.

One major advantage is, referring to the example with the text-pointers, that the task can complete its output without being interrupted and without corrupting other tasks, because you are free to set the position of the poll-routine. If one routine runs two minutes without polling, the system will be 'blocked' for two minutes - that's co-operation!!! For a task, the 'returning' from poll-routine seems as if it was only a call to a subroutine.

For better understanding, let's focus on what the poll-routine does:

Since it is 'called', it must preserve the return-address for the task that polled. It has to check a list where all running tasks are registered and see which one is next in the list. This task's return-address is restored - the poll-routine does not return to it's caller, but to the position of where the next task has formerly called it. That allows to process complex programs parallel, not only special prepared subroutines, as 'function2', because polling can be done from every point. So every running task 'called' the poll-function and is waiting for 'return', except the one task which is running - the 'active' task.

3.3.3 Polling and Problems

That causes another problem. Since the polling can be done from every point, it can also be done in s sub-routine, in a sub-sub-routine, or even more interlocked. If one stack is shared among all the tasks, it would happen that if the active task returns from a subroutine, it returns to the caller of the previous task. So a stack for each task is needed. And there we are at the next point.

3.4 Memory Structure

To achieve several programs running at the same time, more than 64k memory must be accessible. Also, there must be a fixed area which is always present to guarantee that some functions, like the polling, may be called from everywhere. WiOS uses the following memory configuration:

memory page	address-space	description
Page 0	0000-3FFF	internal WiOS-part
Page 1	4000-7FFF	code- and stack-segment for tasks & drivers
Page 2	8000-BFFF	data-segment for tasks & drivers
Page 3	C000-FFFF	global data area and pointer

3.4.1 Internal WiOS-part

Here are the functions that must be accessible from any point and that processes switches in the code- and data-segment. The addresses of functions and variables may vary with different WiOS-versions.

3.4.2 Code- and Stack-Segment

Every task has 16k for code. To make things faster, the stack-pointer is also kept in this segment. So whenever the poll-routine has to switch to another segment to return to the next task, the task's stack is already there - only the stack-position must be corrected.

3.4.3 Data-Segment

This is where the task can switch memory segments freely for storing data permanently or temporarily. There might even be program-code, but this has NOT been tested yet - in every case, the stack resides in page 1, and if a function in page 2 switches the segment of page 1, unpredictable behavior of the system will be the result.

3.4.4 Global Data Area ('GDA')

Here's the interrupt-handler and the fixed table of entry-points. Whenever WiOS' internal part is changed, the correct addresses are filled in the GDA at start-time. This indirect addressing allows other programs to access functions even if their position change. The list of pointers may be expanded in future, but NEVER may existing entries be changed. WiOS also uses this area for its own internal stack to be able to switch page 1.

3.5 Drivers

The total amount of 16k in page 0 is far too less for WiOS itself, and even if it would be enough for this time, it would run out in the future. Therefore only the necessary parts are placed in page 0 like the poll-routine, global variables, memory- and task-management. To offer more than this - WiOS was developed to make graphical applications easier, and yet we have seen nothing of this possibility - modules are used. Each module has to be seen as a 'container' with many functions in it. It is not required that these functions are accessed directly. For many functions, it's sufficient passing the parameters in a list and let this list be moved to the function. We differ between internal and external functions. Internal functions are accessed directly, external functions are accessed indirectly - via internal functions. These external functions also need to be mapped somewhere, and this is exactly the same area where the tasks reside: page 1. This has as a consequence that WiOS does a lot of segment-switching. But, in return, this enables the possibility of an almost unlimited number of external functions. These containers are called 'drivers'.

Difference of Tasks and Drivers

Drivers have two main differences to tasks. They

- * may not poll, so the routines in tasks block any other activity as long as they are executed
- * cannot be present on the screen with windows (which arises from the fact that they may not poll)

But: drivers may call other drivers (like subroutines)

To prevent exaggeration of indirect addressing, external functions are called using numbers, which have to be defined in the driver itself. Therefore every driver has one fixed entry point (per driver) to where the internal 'function-caller' sends the arguments from the task or driver. Since every driver may use the same function-numbers as other drivers - the function numbering normally starts with zero - they are separated by handles.

3.6 Handles

The memory-manager of WiOS needs to save the 'user' of a segment for preventing tasks (when speaking of 'tasks' in this paragraph, both, tasks and drivers are meant) of freeing segments of other tasks. To ensure this, every task must have its own ID-code. This ID is called 'handle' - like file-handles in DOS 2. Whenever a task is loaded, WiOS searches for the next free handle and applies it to the task.

Task handles are searched from start to the end, driver handles vice versa. The maximum number of handles is 252 - and the total amount of possible tasks and drivers depends on how many tasks or drivers are already installed.

As we have seen, the handle of a task or especially driver may vary. How does a task know which handle a certain driver has? WiOS has some standard-drivers, which are needed in every case. Their handles are stored in the GDA.

What if a new driver will become a 'standard' also, for example a printer driver? Every task and driver has its own identification, independent to the handle assigned by WiOS. This identification is a null-terminated string, which length and content may be chosen freely by the programmer. A task's or driver's handle can be searched by its name. If you are searching for a driver which has no entry in the GDA, just search for its name, and WiOS will return either the handle or an error-code so you can reload the driver if necessary at your task's initialization routine. The handle can then be stored in a variable and the new driver can be accessed like all the other ones.

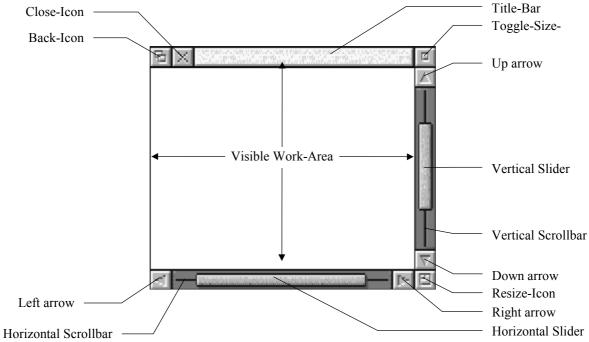
Also windows, standard-icons and fonts are accessed by these handles - the last two work in the same way as described above: You have to search icons and fonts by name once and receive the handle.

See: Init-Routine

3.7 Window Interface

3.7.1 Icons

Windows are almost completely controlled by WiOS. That means you do not have to care about user-activities outside the work-area. Please take a look at this little illustration to make yourself familiar with the designations.



The icons around the window are called window-icons. There are 7 'icons' (some are more than icons) and each can be set separately for every window using the 'icon-flags':

- Back-Icon
- Close-Icon
- Title-Bar
- Toggle-Size-Icon
- Vertical Scrollbar
- Resize-Icon
- Horizontal Scrollbar

3.7.2 Parameters

To create a window, you have to set up a parameter-heap with the icon flags as well as the following:

- absolute coordinate of upper-left corner on the screen
- size of the visible work-area (from now on called 'work-area' size)
- size of the total work-area ('virtual' size)
- scroll offsets
- minimum size of work-area (i.e. how small the window can be made by the user)
- maximum size of work-area (...how big...)
- window-area-flags
- icon-flags
- work-area-flags

That is not all you can set, but almost.

What is the work-area?

It is the place where the task can show its own information, graphics, whatever. But a more interesting fact is that the size of the area given as the work-area size is never changed by WiOS due to window-icons. If your window's work-area is 100x100 pixels and you add a titlebar, then it will not take space from your work-area but enlarge the total height of the window by the height of the title-bar. There are also some priorities with the flags, so some icons are 'shrinked' if there's not enough place around the window. So the best thing is you just try all of the 2^7=128 combinations of the window-flags and see by yourself.

By the way: WiOS does no checking if the icons fit around the window - if your window is too small, some icons will overhang others. Also WiOS does not allow the user to drag or resize windows totally or even partially out of the screen. It's up to you to set the minimum and maximum window-size in a way that this does not happen. There's a definition for sizes, so you don't have to try that out (we're still in the introduction, and I won't become too specific here).

Referring to what was said at the beginning, WiOS does the complete checking and handling of the window-icons. Nevertheless does the task still have the full control on whether to accept what the user has done, to make changes, or to reject it completely. WiOS sends the parameter heap to the task BEFORE doing any actions on the screen - it's up to the task to give the command to redraw the new window.

3.7.3 Window Management

WiOS only holds a parameter-list for each window in memory. Their identification is similar to the tasks and drivers, but they have only handles, no names. In contrary to conventional graphical-user-shells, windows contents - and the contents of the windows below - are not stored as bitmap-graphics, but only as coordinates on the screen. Due to the fact that all windows have one thing in common - they are rectangular - it is possible to calculate which parts are visible on the screen. Since many parts of a window can be hidden by upper windows, it is not easy to draw only the visible parts. WiOS offers two theoretically ways, whereas only one has been tested completely. When WiOS sends the list of rectangular fragments on the screen, you can either redraw

- * only the necessary parts (this method is not tested since it's very tricky and might be even slower yet than the other).
- * the full window at coordinates WiOS offers you (on a hidden page) and let WiOS copy the necessary parts of this window to the screen.

As long as it is only possible to auto-cut graphic commands to a rectangular area - not to a polygon - the first method is not advisable - although there are cases where the redraw of a full window takes much more time than the redraw of small parts. But that's part of an optimization of the window-driver and - as said before - WiOS is only 'optimized' for functionality, neither for speed nor for efficiency.

3.7.4 Window Types

Parent Windows

are conventional windows. If only single window is opened by a task, this window has the status 'parent'.

Pane Windows

are windows that are connected to a 'parent window'. They are always directly over their parent. If the parent is moved to the back, all its pane windows will go back as well. Also if the parent is put on front. They will be closed automatically if the parent is closed and - that's different from parent-windows - without the task being informed. It only gets to know if the parent window shall be closed.

Now, after having told you so much about WiOS sends information to the task, it's the best time to start talking about the interaction of both.

3.8 Events - Communication between WiOS and Tasks

Imagine the following situation: The task does a very complex calculation that lasts some minutes. Of course (it is co-operative!) it polls every 10th loop. Now the user closes a window which is in front of the calculating task's window. Although the task would be busy for some minutes, it should be able redraw its window on demand.

To permit WiOS to 'talk' to tasks, they must have an 'Event-Handler'. There's no better place to let WiOS send information to the task than after the polling. A number which is called the 'Event' is returned to the task. It has do deal with it, so every task must call its own event-handler and take action on the event before continuing with its calculation.

To guarantee an immediate screen update after a window-move-operation by the user, the events have a different priority. That means task which has to redraw a window should be called before he would have the turn in the normal poll sequence and before other tasks that are 'just' waiting for being returned to without the need to redraw.

This requires a differentiation between tasks that must be called to update the screen and tasks that are just doing some calculations where it does not matter if they continue one or two seconds later than usually.

It's not important for programming, but it may be interesting for you to have a better understanding of what WiOS does, that there are three priorities of events. Here they are (highest priority first):

- Open
- Redraw
- Scroll
- Close
- User-Message
- Mouseclick
- Pointer has left window
- Pointer enters window
- Pointer is over work-area
- End of a drag-operation (- planned: keyboard hit)
- Null-Event

The event with the lowest priority is only sent if nothing else is to do. And it's also the event that has to do with the wide spread (but wrong) idea of 'multitasking'. It is sent to tasks (when speaking of 'sending' something to tasks, it is meant that the poll-routine returns this information to the poller) if nothing else important has to be done and the task can continue its 'background'-calculation. One could translate it like 'now there's nothing else important to do, and you may continue with your normal activity'. The word 'background' is in quotes because there are no tasks working either in the background or foreground. Every task is waiting for the poll-routine to return. The only 'active' task is the one which has the turn - and that may change many times per second. Anyway, it is clear that a redraw-event is more important than a null-event and must be executed first.

But, on the other hand, every task must be co-operative not to take advantage of being called (i.e. returned) for a redraw-event to process its calculation after redrawing since that would cause other tasks waiting for the null-event even wait longer for it. Therefore tasks that receive a high-priority event should poll again immediately after this event has been handled and wait for the null-event to continue with a 'background' calculation.

WOS - Application Programming Documentation

Actually, not every task has to do complex calculations all the time. If a task just waits for the user clicking somewhere in the window or for a keyboard-entry, it is not necessary for him to be called all the time if nothing happens. Also would calculating tasks take profit of the time needed for calling tasks with the null-event. The result of this fact is the possibility to mask events.

The task may mask-out events for not being called all the time. The simplest case is the null-event. If a task waits for something, it is able to tell the poll-routine 'activate me only when a mouseclick is done or a window has to be redrawn, but not for drag-, null- and pointer-over-window-events'.

That speeds up the processing time because WiOS will not call all tasks if nothing of interest (for the task) happened.

3.9 User Input

3.9.1 Mouse Triggers

Traditionally, programs waiting for mouse-input have to scan the triggers all the time. If the program pauses for a while, it won't know if you have pressed and released the triggers in this pause. The same problem is for the keyboard-entry. But the MSX-system uses a trick to inform programs if a key was pressed during their 'absence'. This technique is called 'buffering'. Whenever you call the BIOS-routine to get the next key, a pointer in a buffer is incremented and returns the next key in this buffer. The buffering is only possible because the system scans the keys all the time - no matter if the program also scans the keys or is busy with something else - and fills the buffer if a key is pressed. To make sure that the keyboard-scanning is done constantly, the routine which does the check must be hooked in the interrupt.

[btw: Therefore all the keys pressed during a time when the interrupt is disabled are not put into the buffer - that's why most of the keys you press while something is loaded from disk are not buffered]

In a multitasking environment, it might be possible that your task does not have the turn for a while and it's impossible to scan what the user did one second ago. WiOS' mouse movement is done in interrupt, so it's possible to buffer mouseclicks - but only the last click is buffered. This has several reasons. The most important one is that the user may not click randomly over the desktop while one task is drawing, one after each other, the following windows: 'FORMAT - ABCDEFGH', 'FORMAT DRIVE A - YES - NO'. If you clicked (unfortunately) at position 'A' in the first window and at 'YES' in the second window before they appeared, and the task takes the data from a multiple-click buffer, your drive A will be emptied. You would blame WiOS for that behavior and demand me to pay your damages. One buffered click is warrantable, and if you are still afraid of this, your task may even accept a 'FORMAT - YES' only when the mouseclick is done during a direct trigger-scanning.

How does a task know if the trigger is hold or has been released and pressed again since the last polling?

As there are multiple tasks, WiOS could to update a table for every task which has requested the mouse-button state. But since that would be too slow, WiOS simply stores a timer in the buffer when a click is done. The task must use this timer for getting to know is the mouse has been released and pressed again during polling. If the mouse is still pressed, and the timer in the buffer is the same as before polling, the mouse has been hold. If the timer is different, the mouse has been pressed anew. Sure, there's a theoretical inaccuracy if the mouse is pressed exactly at the same timer-state as before, but this chance is 65536:1 - if the mouse is pressed exactly 65536 vsyncs afterwards. To know where the click was, the mouse-handler also stores the coordinates of the click so the task always knows the position where the mouse-pointer was when the trigger was clicked, not only where it is now. This assures all user-activities to be handled exactly, even if the task can first react some seconds later.

3.9.2 Key Presses

A bit more difficult is the buffering of the keyboard-input. If it would be the normal way, one task that's waiting for the 'K'-key always empties the keyboard-buffer as long as you press 'K', and another task that waits for the [RETURN]-key does not get any keys. Theoretically, every task should have its own keyboard-buffer. Of course, that would take too much time and space, but WiOS does something similar: it uses one buffer with several pointers for each task. Whenever a task calls the function to kill or to read the keyboard-buffer, only its pointer is reset or incremented, so one task may just have cleared his keyboard-buffer, while the next task will get the last 100 key-presses. The buffer is 256 bytes large and is updated at polling-time and whenever WiOS' getkey function is called. If a task starts to read the buffer after about 300 key-presses, if does only receive the last 300-256=44 keys.

In every case, there are two things to remember:

- * If a task reads the keys directly via the BIOS-getkey call, only this task will get the keys they will not be buffered for other tasks. But they should! Especially, when a task wants to react on 'hot-keys'.
- * If a task waits for a recent key-press, kill the buffer and start waiting for a key in a loop where the getkey- and poll-routine is called.

3.10 Drag & Drop

The multitasking structure of WiOS allows tasks to call routines from other tasks. That does not only make it unnecessary for programmers to code common routines again and again, that also saves a lot of memory and that even gives the possibility to use routines from other programmers, even if you don't know in which language the routine is written and how much memory the program uses.

A good example is the disk-menu. There exists only ONE task that makes all file-handling like deleting, moving, copying and even opening files. This disk-menu is a basic task of WiOS and delivered as the 'standard' application with every WiOS, so nobody has to write its own disk-menu again.

In fact, one problem is that the disk-menu is not yet written, since your WiOS-version comes directly from the development (if you put your hand on WiOS, you can feel that it's still warm :-)

To be able to write Alpha-Test tasks without the disk-menu, you have to understand how 'drag & drop' works.

Normally, all interactions between tasks are done by the events. And if you look back to the events, you will see that there's a 'End-Of-Drag-Operation' event. With this event, there is always coming a data-field which identifies the incoming data for the receiving task. The structure of the data-field is not yet of importance and described later. Now if the user drags files from the file-manager to a task's window or to its icon on the toolbar, the task will receive this event with the ID that there are filenames coming and where they are stored - nothing else. And that's exactly the point on how you can write WiOS-tasks without the disk-menu.

3.10.1 How to simulate non-existing parts in the Alpha-Release

You can simulate the file-manager sending files to a task by writing a second task sending an 'E_EDRAG' and an 'E_USERMSG' event (for both, see chapter 3.16 - Data Exchange Specification) to your own task with the needed file(s). On first view, this is a stupid way: Why not opening the file directly? Well, you *can* do this, but if you do it the other way, it's easier to adapt your application to the complete public WiOS-version with the file-manager included - to be more precise, if you do it this way, there is at least no modification needed later!

3.10.2 File Management

It is still one weak point of WiOS that no checking is done if you open one file with multiple instances of one task or with different tasks, make changes to the file and save it. Although it's not yet tested what DOS 2 does, there is no handling in WiOS. Since this is part of the file-driver, this can be handled correctly or prevented in future versions of it.

Disk-errors are also planned to be handled by WiOS itself, so, for Alpha-Testing, just don't produce them >;->

3.11 Programming Language

Now comes a tough section for all hard-core assembler guys:

Forget assembler! Here comes C!!!

Note: When speaking of C, only ASCII-C 1.2 is referred to, not BDS, Hi-Tech or any other C language for MSX!

Why C?

WiOS is not developed for those little tiny baby-application-tsr's that nobody can use except for some presentation like 'Look, I enter basic and the music is still playing in interrupt!' With WiOS, it is not only possible, we even we expect programmers to make bigger applications. It does not have to be a Graphic-Program with a balloon-logo. But WiOS is developed for big programs, no matter if they need one, two or even three megabytes of memory. That also means that the program does not only consist of a separate special-adapted-and-handoptimized routine that rotates the screen while scrolling a text and flashing the colors until the next space-key is pressed. Speed does not have the highest priority for applications. Every program with which the user can do many things need a complex handling-routine - and lots of time for debugging. And there are many standard-things like loops and calculations that have to be done ever and ever again. Although it's not that hard to code this is assembler, you will come into trouble if you have several loops in each other or if you want to make a variable step-size. As you all know from basic, this is no problem. But in assembler, programs tend to get bigger than the screen, then bigger than 10 screens, and at the latest it will take you a lot of time if the program does not what you want. Then you will spend a lot of time debugging these standard-routines. And yet, you didn't even come to the complex stuff although applications are full of them. How easy would it be if you could just insert a print command to get the contents of the variables. This is easy with so-called 'higher programming languages', and most of the programmers just left basic because of its slowness and lack of commands. Since C offers 100% assembler-routine implementation, you can make a good mix of speed, efficiency, and at least development time. C is also recommended due to the way parameters are sent to WiOS. Since C always creates an assembler listing, it is, of course, possible to write complete WiOS-programs in assembler, but believe me, without C, WiOS would not have been finished in ten years.

Now take this rigorous advice: Begin to learn C (if you still haven't) or forget programming applications for WiOS. This is not only of use for the MSX but also for your future if you are continuing in the software-market - no matter if you change to the PC, UNIX or whatever.

And even more rigorously: You will interest no employer if you tell him that you write only Z80-assembly and MSX-Basic fluently!!!

3.12 C specification

Congratulations, if you're still reading, the you are 'with us'!!!

Next comes a small introduction of how C on the MSX works:

3.12.1 How to come from the C source to an executable COM-file

C consists of two programs for compiling the source-code, one for assembling and one for linking an executable COM-file.

First of all, the Parser:

The program 'CF.COM' takes your C-file and checks all syntax-errors. Its output is a file with the extension 'TCO' - the so-called T-code file. If your program passed the parser, you can be 99% sure that your program does not contain fatal errors that prevent the compilation of it. This step is very important because the next step requires this T-code file.

The Code-Generator:

It takes the T-code file and generates pure assembly-code. The result is put into a MAC-file. It's plain ASCII-text and this is the point where you can still hand-optimize or use XelaSoft's optimizer to optimize the compiled code. For test-runs, it's sufficient to let the code 'as is'.

The M80 (macro-)assembler:

The code-generator produces an assemblable (stupid word;) file for the M80 assembler. If you take a look at this file using a text-editor, you'll find some directives that allow the usage or variables and routines outside of this file - and these non-standard commands are not supported by other assemblers. M80 creates a REL-file with your code - which means 'relocatable'. This file contains only relative jumps and calls, and stores labels for 'external' functions and variables. This file can NOT be executed. It must be linked first.

The L80 linker:

It takes all the REL-files you have given in the command-line and calculates all absolute addresses. On this step, you MUST give all the files that contain the functions and variables, because an executable COM-file is generated. If there are some labels missing, L80 will inform you of that. The COM-file is always generated, but if there are error messages, you have a good chance of you program hanging somewhere

Why these many steps?

Due to the limit of 64K on the MSX, the Code-Generator is not able to handle too complex functions and store that many labels which are needed for writing complex programs. Normally, if a function exceeds 50 lines of code, the code-generator capitulates. Then the routine has to be split up into small routines.

Also, the compiler is very slow. If you write bigger programs, it will take minutes to compile them. Therefore, it is advisable to separate 'good' (i.e. working) routines from the code into a separate file, so it's no longer necessary to compile them ever and ever again with the non-working routines.

Now you have seen that, in every case, bigger programs consist of many small source-files. As soon as REL-files of all parts are produced, you only need to compile the part which contains bugs and link it with all the other parts again. By the way: Linking REL-files is many times faster than to concatenate all MAC-files and assemble this big file with another assembler that produces an executable directly. Also linking is necessary since parts of the C runtime-library are copied. For example, if your program uses the 'printf' command, the executable is about 1 kilobyte larger than if you don't use it - only the necessary functions are linked from the so-called standard-I/O-library ('STDIO').

3.12.2 The limits of C

C is strictly limited to 16-bit addressing-space - no built-in memory-mapping, no bigger programs. I actually don't know WHICH part limits to 64KB, but I know that the last action preventing programs of getting too large is the linking-process. There exists the L80 version 2, but use version 1 for linking C-programs: your program can be larger than in version 2! Actually, I don't know what's different in the new version, but I could compile anything until now with the old one. Anyway, the linker also creates a COM-file and expects any program to start at 100h. If you write an ORG 4000h, you will receive 3F00h bytes with zeroes in your COM-file.

3.12.3 Breaking the limits of C

Now we have a big problem: I first wrote that it's no problem writing programs that have 40MB of code-size, and now I tell you that C is limited for 64KB. Well, both is true, but it has to be done in a non-standard, but easy way. You won't even notice when your program is larger than 64KB.

Before continuing with the description on how this works, please remember the code-area of your program: it if fixed to 16k for code-size and stack - one memory segment. Therefore the memory 'limit' is not even 64k, as in non-mapped C, but only 16k - but with a difference: 16k per function!

Got it?

That's exactly the same limit when you are programming memory-mapped programs in assembler. And you have to use the same old tricks for accessing data outside of a segment.

To create WiOS-'executables', the linker has to be told that the task has to start at 4000h. This can be achieved with the option '/P:4000' as the first argument in the L80 command-line. We get a COM-file that has about 16k of zeroes. This file must be left AS IS. With the Application-Creation-Toolkit (the disk and the documentation you just received;) there comes the most important program for creating tasks and drivers, MAKDRV.COM, with which it is possible to make real BIG applications.

Although execution speed has never had the highest priority while developing WiOS, efficiency was. Therefore tasks and drivers do not contain a single file for each 16k-block, but all code is stored in one file. That makes loading time faster, saves a lot of harddisk space (since every file takes at least 8, 16 or 32 kilobytes, even if it's size is only one byte!) and looks just nicer than 'TASK.001', 'TASK.002', ..., 'TASK.9999999' - finally it's also easier to spread the program in mailboxes. MAKDRV creates this from all your COM-files, cuts the 16k-zeroes-header and fills some important information in the file-header, like the number of blocks the file contains and how large each part is - since it would be a waste of resources if every code-segment would always take 16k or harddisk-space in the file. Also, there is the possibility of making crypted files. WiOS would decrypts them on-the-load, but this is not yet supported (it depends on the inquiry;)

Returning on how to compile bigger programs: If you write larger programs, you split it into 16k blocks. For that, you have to think a little bit about the structure of your program, because it would be a waste of memory, if you just say 'Okay, one segment per function, even if it's only ADD HL,HL and RET'. Since for 'sending' variables to other segments, you need to store them in your data-segment and restore them in the called routine in the other segment, it's advisable to make 'bundles' of functions that need to interact a lot with each other, and put them into one 16k-segment. That saves execution time, because switching segments always needs a lot of time. Also with functions in one segment, you may use the 'standard' C variable-accessing structure. Calling functions in other segments bring up one more problem: If you change a routine in another block, the address of the following functions are not the same.

WOS - Application Programming Documentation

How does the caller know the address of a routine in another segment? It's the same conventional way as programming non-WiOS-inter-segment functions. The keyword is, again, 'indirect access'. Each part has to have a fixed table where the current addresses are stored. So tasks do not call functions like 'go to function at address 4056h' but 'go to the first function in segment x'. How this table is created, and where it stands, is written later in this documentation.

3.13 Basics

This section explains exactly what is needed to use the full possibilities of WiOS and how it is done. All explanations focus on writing programs in C - if you want to create WiOS-applications in pure assembler, please refer to chapter ?? where you can look up how to emulate the C-structure. No special programming-course is given for assembler, so you will have to adapt it yourself with the information given on the following pages for C.

The Basics contain the following parts:

An explanation, how the variables of the Global Data Area must be interpreted and used in own applications, how maximum compatibility to future versions of WiOS is guaranteed, a detailed information about how WiOS functions can be accessed from other programs, how simple data can be sent and what parts of WiOS can be called, and finally what data WiOS returns to the tasks.

3.13.1 Accessing the Global Data Area

As for the linker, all external functions you want to call from a task must be linked to your executable. It is, of course, not possible to link the complete WiOS with your task, since WiOS itself is larger than 64KB and this would be in opposite of the possibility of using functions from other tasks. Another way of accessing functions without linking was developed for WiOS.

For all tasks and drivers, you need to link the GDA-file which contains the addresses of the globals which allow tasks to access WiOS functions and variables. This file only contains labels that point to a specific address-table in page 3, the area above C000h, so it does not take a single byte of memory in your program! The linker just takes the addresses and fills them at the respective calls in your routines.

ALL GDA-ENTRIES ARE POINTERS.

So NEVER call or access them directly! Always use the address where the entry points to.

Always include the file 'GDA.H' if you want to have access to WiOS (which is always the case!)

3.13.2 Compatibility

WiOS will not always be the same. It will be changed ever and ever again. To keep maximum compatibility, please take care of the following points:

- Never write down the address in the GDA and call it directly. Always use indirect calling! Look up the pointer in the GDA and jump to it.
- Never use the area above the GDA! WiOS might expand. There are global storage-areas for your data. Use them.

In return, it is guaranteed that the existing GDA will NEVER change - only expand. If you call the first pointer in the GDA, it will lead you to the same function in WiOS version 1.0 Alpha as well as in WiOS '98;) If a new function is written, a new GDA-entry at the end of the list will be added. The old pointers will always exist!

3.13.3 Calling WiOS-Functions

In C, functions are normally called using a syntax like

function();

where the code generated for this function always calls the address 'function' directly. In WiOS, functions and variables also have labels. All pointers in the GDA are named after their function they point to with a leading underscore.

To call at an address where a variable points to, the syntax looks a bit different. Since we have indirect addressing, our example-label would not point to the function itself but to an address in the GDA which contains the address of the function. This label would be named with a leading underscore: '_function'. In C, the contents of a pointer is accessed with an asterisk before the label-name. To call the address of where a pointer points to, the label and the asterisk must be surrounded by brackets.

To call to the address where the label points to, write

(*_function)();

Easy, eh?

There are only very few functions that are called directly via the GDA. Many parts of WiOS are not mapped in at calling time. Since they are 'normal' drivers, they are also in page 1, from 4000h to 7FFFh - where the calling task is. Another indirection is used: Tasks do not call the driver itself. They call a function which maps in the driver, calls it, restores the segment and returns to the task. This function is called 'caldrv' - which means 'call driver'. The referring GDA-entry is named '_caldrv'. Whenever a WiOS-function is called, it's always called with

(* caldrv)(...);

where '(...)' stands for the arguments you are sending to the 'caldrv' function.

WiOS needs to know which driver you want to call. Normally, you can search the driver by its name and receive the handle. But there are some standard drivers which are always loaded when WiOS is started. These handles can also be found in the GDA.

The standard-drivers with their correct names are in alphabetical order: (for a detailed list of included functions in the drivers, see chapter ??)

_hgiodry Graphic I/O driver (loads fonts and icons)

hgrpdrv Graphic driver (handles graphic-output to the screen)

_hwindrv Window driver (guess what!)

Always remember that the GDA is a pointer list. These variables do not contain the handle itself but the address where the handle is stored. In C, again, this is very simple. The first argument for the 'caldry'-function is the handle of the driver. So just write

(*_caldrv)(*_hwindrv); does work

to access the window-driver. Indirect addressing looks a bit strange, since you need brackets for calling the function, but not for getting the handle - but as brackets for the functionarguments are needed, it looks twice the same, the function and the argument. For comparison, here's the corresponding command in direct addressing:

caldrv(hwindrv); does NOT work

Each driver can have more than one function (of course), so we need to tell the task which function we want to call. These functions are numbered in a possible range from 1 to 32767. The 'standard' drivers (i.e. all 8 drivers mentioned before) are numbered linear, beginning with 1.

As function 4 in the window driver does something completely different than function 4 in the graphic driver, take care not to access the wrong driver.

For making things easier, the function-numbers of all drivers can be accessed via text. Each driver has its own Header-file where all functions are defined. If you want to get the current data of a window at address 5000h, the source-code looks like this:

(* caldrv)(* hwindrv,GET WIN STATE,0x5000);

where 'hwindrv' means ' \underline{h} andle of \underline{win} dow \underline{driv} er'.

For accessing the driver's functions by names, you have to include the header files - 'WINFNC.H' if you need the window-driver functions.

There are some functions that can be called directly (via the indirect way through the GDA). They can be found in chapter ?? where all GDA pointers are listed.

3.13.4 Return Values

Every driver-function returns a 16-bit unsigned integer. But not all functions return a reasonable value (because they do not need to return anything). Functions that return one value return it directly from the function. Functions that return several values return a pointer to an address where the return values are stored. See the description of the driver functions in chapter ??

3.13.5 Definitions

If you encounter variables or structures which are non-standard C types, you will find their definition in one of those files in most cases. These header files contain the following definitions:

GDA.H global data area variables

DEF.H type and global definitions

DEFSTR.H structure definitions of blocks

EVENTFNC.H event definitions

3.14 Function Reference (WiOS Drivers)

This chapter contains a complete documented list of all functions of the drivers which are available when WiOS has been started. Each header contains:

- * the description of the driver
- * the driver-name (for searching purpose, although this is not necessary since the handles of all standard-drivers can be found in the GDA)
- * the label-name of the pointer to the driver's handle in the GDA
- * the header-file with the definitions

This is only for referral use. You will not find programming techniques or fields of application.

Important:

- * The function number is always 16-bit unsigned integer!
- * The return value is always 16-bit unsigned integer! Sometimes you need to convert it to 8 bit!
- ★ The return value -1 means 65535!
- * Functions marked as 'used internally' are only for internal use of WiOS and are NOT (yet) for universal use. These functions are NO dummies using them may result in unpredictable actions or unstability of WiOS.

3.14.1 External Driver

Description: External driver Driver-Name: "External Driver"

GDA: _hextdrv Header-File: EXTFNC.H

Function Name	Number	Function
-	1	used internally
Arguments	-	
Return	-	

Function Name	Number	Function
RESKEY	2	clear keyboard-buffer of calling task
Arguments	-	
Return	-	

Function Name	Number	Function
GETKEY	3	get next keyboard-character in buffer of calling task
Arguments	-	
Return	0	if no key has been pressed since last GETKEY
	1255	character-code of the key

3.14.2 File-System Driver

Description: File-System driver
Driver-Name: "File-System"
GDA: _hfsdrv
Header-File: FSFNC.H

All disk-functions have the same numbers and names as their corresponding DOS 2-functions.

These functions were not complete at the time when this documentation was printed, so for a description look in the Application Creation Toolkit.

If you want more DOS functions to be supported please let me know.

Function Name	Number	Function
_SELDSK	0x0e	set current drive
Arguments	-	
Return	-	

Function Name	Number	Function
_CURDRV	0x19	get current drive
Arguments		
Return		

Function Name	Number	Function
_RDABS	0x2f	absolute sector read
Arguments		
Return		

Function Name	Number	Function
_WRABS	0x30	absolute sector write
Arguments		
Return		

Function Name	Number	Function
_FFIRST	0x40	find first entry
Arguments	char *	pointer to filename / file info block
	char *	pointer to new file info block
	char	search attributes
Return		

Function Name	Number	Function
_FNEXT	0x41	find next entry
Arguments	char *	pointer to file info block
Return		

Function Name	Number	Function
_OPEN	0x43	open file handle
Arguments	char *	pointer to filename
	int	mode
Return	0	error
	1255	file handle

Function Name	Number	Function
_CREATE	0x44	create file and open handle
Arguments		
Return		

WOS - Application Programming Documentation

Function Name	Number	Function
_CLOSE	0x45	close file handle
Arguments	int	file handle
Return	0	done
	1255	error

Function Name	Number	Function
_READ	0x48	read from file handle
Arguments	int	file handle
	char *	destination address (4000hBFFFh)
	unsigned	len (016384)
	T_SEG	destination segment (03071)
	char	dummy (must be 0!) mode (CRC, CRYPT)
Return	0	done
	1255	error

Function Name	Number	Function
_WRITE	0x49	write to file handle
Arguments		
Return		

Function Name	Number	Function
_SEEK	0x4a	seek (position file pointer)
Arguments	int	file handle
	unsigned	offset (065535)
	char	mode 0 from start of file
		1 from end of file
		2 relative to current position
Return	0	done
	1255	error

Function Name	Number	Function
_DELETE	0x4d	delete file or subdirectory
Arguments		
Return		

Function Name	Number	Function
_RENAME	0x4e	rename file or subdirectory
Arguments		
Return		

Function Name	Number	Function
_MOVE	0x4f	move file or subdirectory
Arguments		
Return		

Function Name	Number	Function
_ATTR	0x50	change file or subdirectory attributes
Arguments		
Return		

Function Name	Number	Function
_FTIME	0x51	get/set file date and time
Arguments		
Return		

Function Name	Number	Function
_VERIFY	0x58	get verify flag setting
Arguments		
Return		

WOS - Application Programming Documentation

Function Name	Number	Function
_GETCD	0x59	get current directory
Arguments		
Return		

Function Name	Number	Function
_CHDIR	0x5a	change current directory
Arguments		
Return		

Function Name	Number	Function
_PARSE	0x5b	parse pathname
Arguments		
Return		

Function Name	Number	Function
_FORMAT	0x67	format disk
Arguments		
Return		

Function Name	Number	Function
_D2V	0x80	read directly from file handle to port-address (e.g. VRAM)
Arguments	int	handle
	unsigned	len
	char	port
Return	0	done
	1255	error

3.14.3 Graphic I/O Driver

Description: Graphic I/O driver "Graphic-I/O Driver"
_hgiodrv
GIOFNC.H Driver-Name:

GDA: Header-File:

Function Name	Number	Function
-	1	used internally
Arguments	-	
Return	-	

Function Name	Number	Function
-	2	used internally
Arguments	-	
Return	-	

Function Name	Number	Function
-	3	used internally
Arguments	-	
Return	-	

Function Name	Number	Function
GET_VHANDLE	4	get handle of a file in Videoram-Directory (used for finding font- and icon-handles)
Arguments	char	ICON for icon FONT for font
	char *	filename
Return	065534	handle of icon/font
	-1	file not found

Function Name	Number	Function
ADDFONTLIB	5	load a font from disk
Arguments	char *	filename
Return	065534	handle of loaded font
	-1	file not found
Note	Don't overload memory!!! No checking is done yet!	

Function Name	Number	Function
-	6	used internally
Arguments	-	
Return	-	

3.14.4 Graphic Driver

Description:
Driver-Name:
GDA:
Header-File:
Graphic driver
"Graphic Driver"
"Graphic Driver"
"Graphic Driver"
"Graphic Driver"
"GRAPFNC.H

Function Name	Number	Function
MOUSEBOX	1	set the box where the mouse can move
Arguments	absolute co	ordinates - corners must be in the order upper-left / lower-right!
	int	x coordinates of upper left corner
	int	у
	int	x coordinates of lower right corner
	int	y
Return	-	

Function Name	Number	Function	
PPOINTER	2	put the mouse-pointer at absolute coordinates on the screen	
Arguments	absolute coordinates		
	int	x coordinates of mouse-pointer	
	int	у	
Return	-		
Note	If the coordinates are outside of the mousebox, it will 'jump' into them at the next interrupt!)		

Function Name	Number	Function
MOUSEMOVE	3	move mouse according to the user-movement (not necessary!)
Arguments	-	
Return	-	

Function Name	Number	Function
STOREBOX	4	store current VDA-coordinates
Arguments	-	
Return	-	
Note	also used by	y set of VDA-coordinated to be stored! Since this functions is y WiOS, the function RESTOBOX might not restore the you stores after a polling or calling the window-driver!

Function Name	Number	Function
RESTOBOX	5	restore the last set of VDA-coordinates stored with STOREBOX
Arguments	-	
Return	-	
Note	see note at STOREBOX	

Function Name	Number	Function	
SETBOX	6	set valid display area (VDA)	
Arguments	absolute co	ordinates	
	int	x coordinates of first corner	
	int	у	
	int	x coordinates of opposite corner	
	int		
Return	-		
Note	corners may be set freely - they are adjusted to upper-left / lower-right		

WOS - Application Programming Documentation

Function Name	Number	Function	
COPY	7	copy graphic area corresponding to VDA	
Arguments	int	sx coordinates of upper left corner from source-area	
	int	sy	
	int	nx width of area to be copied	
	int	ny	
	int	dx destination coordinates	
	int	dy	
Return	0	area was fully copied (destination is in VDA)	
	1	nothing was copied (destination is totally out of VDA)	
	2	area was cutted (destination is partially out of VDA)	
Note	copy-direct	ction is corrected automatically if source and destination intersect	

Function Name	Number	Function			
PUTICON	8	puts icon on	screen corresponding to VD		
Arguments	int	032767	handle of standard-icon from VRAM		
		-1	icon graphic-data comes from memory and '*_adrblk' must contain the following information: offset len +0 2 address, +2 2 segment of icon-graphic-data (32768 color mode) +4 2 width, +6 2 height of icon		
	int	x	coordinates of destination		
	int	У			
Return	0	icon was fully put (destination is in VDA) nothing was put (destination is totally out of VDA)			
	1				
	2	icon was cu	tted (destination is partially out of VDA)		

Function Name	Number	Function	
EXPICON	9	expand icon - fill complete VDA with icon	
Arguments	int	032767	handle of standard-icon from VRAM
		-1	icon graphic-data comes from memory (see PUTICON for data of '*_adrblk')
	int	X	coordinates
	int	у	
Return	-		
Note	x,y-coordinate set the destination of the 'master'-icon. The area around this icon is tiled with the same icon		

Function Name	Number	Function	
SETFONT	10	set current font	
Arguments	int	font-handle	
Return		height of font	
Note	no error-checking for the font-handle is done. If a non-existing font-handle is sent, the height of the current font is returned and nothing is changed		

Function Name	Number	Function	
SETCOL	11	set the font-color	
Arguments	int	foreground color	
	int	background color	
Return	-		
Note	color 0 is transparent		

WOS - Application Programming Documentation

Function Name	Number	Function	
SETPOS	12	set current text-position	
Arguments	int	x coordinates of text-position	
	int	у	
	int	distance in pixels between characters	
Return			

Function Name	Number	Function		
WRTTXT	13	write text to the screen according to VDA		
Arguments	char *	address of string		
	T_SEG	segment of string		
	int	len 0 all characters until byte '0' are written		
		132767 characters are written		
	char	flags not used yet must be 0		
Return		total len of text in pixels (NOT the length of displayed text)		
Note	distance between characters is set with function 'SETPOS' font color 0 is			
	transparent			
	every underscore will underline the next character			

Function Name	Number	Function
GETTXTLEN	14	get len of null-terminated string in pixels
Arguments	char *	address of string
	T_SEG	segment of string
Return	-	
Note	distance between characters is set with function 'SETPOS' every underscore will underline the next character	

Function Name	Number	Function
GETCHARS	15	get number of characters fitting in given width
Arguments	char *	address of string
	T_SEG	segment of string
	int	width of area in pixels
	char	direction 0 forward (increment text-pointer)
		1255 backward
Return		number of characters

Function Name	Number	Function
BOX	16	draw colored box on screen according to VDA
Arguments	int	x coordinates of upper left corner
	int	у
	int	nx width of box
	int	ny
	int	032767 color of box
	int	logical operation (see V9990 technical manual)
Return	-	

Function Name	Number	Function	
GETBOX	17	get current VDA-coordinates	
Arguments	-		
Return		address of data-block offset len +0 2 x-coordinate of upper left corner +2 2 y +4 2 x-coordinate of lower right corner +6 2 v	
Note	use this function to store current VDA-coordinates permanently in your own memory-area (better than STOREBOX when polling or calling the window driver)		

3.14.5 Memory Driver

Description: Driver-Name: Memory driver

GDA: _hmemdrv MEMFNC.H Header-File:

This driver is in the internal WiOS-part and has no name!

Function Name	Number	Function	
ALLSEG	1	allocate one new segment	
Arguments	T_TASK	0252 task's handle	
Return	-1	no more free segments	
	032767	number of the segment	

Function Name	Number	Function	
ENASEG	2	enable segment in page 1 / 2	
Arguments	T_SEG	032767 number of the segment	
	char	1 page 1: 4000h-7fffh	
		page 2: 8000h-bfffh	
Return	0	segment was enabled	
	-1	segment could not be enabled (out of range)	
	-2	segment was not allocated	

Function Name	Number	Function			
FRESEG	3	free allocated segment			
Arguments	T_SEG	032767 number of the segment			
Return	0	segment was freed			
	-1	segment could not be freed (out of range)			
	-2	segment was not allocated by this task			
Note	After freeing a segment should be paid attention NOT to write in the area				
	where the segment was if no other segment has been enabled at this page				

WOS - Application Programming Documentation

Function Name	Number	Function	
GETSEG	4	get status of segment	
Arguments	T_SEG	032767 number of the segment	
Return	0252	task/driver's handle which allocated the segment	
	253	used by a non specified application (was allocated when WiOS	
		started)	
	254	too slow memory for CPU (e.g. Sony mappers in turbo mode)	
	255	segment is free	

Function Name	Number	Function
-	5	used internally
Arguments	-	
Return	-	

Function Name	Number	Function	
SEGCPY	6	copy a given number of bytes from one segment to another	
Arguments	relative add	esses in segments: range is from 03fffh	
	T_SEG	032767 source segment	
	T_SEG	032767 destination segment	
	unsigned	016383 source address	
	unsigned	016383 destination address	
	unsigned	016384 number of bytes to be copied	
Return	0	data was copied	
	-1	out of range:	
		- src is > 16383 or	
		- dest is > 16383 or	
		- length of the block exceeds the src- or dest-segment	
Note	no checking	ng is done whether the segments are allocated or to which task	
	they belong		

Function Name	Number	Function		
SEGNCPY	7	copy bytes f	rom one segment to another until a given byte is found	
Arguments	relative add	relative addresses in segments: range is from 03fffh		
	T_SEG	032767	source segment	
	T_SEG	032767	destination segment	
	unsigned	016383	source address	
	unsigned	016383	destination address	
	unsigned	016384	number of bytes to be copied	
	char	0255	termination byte	
Return	-1	see SEGCPY		
	116384	number of	bytes copied	
Note	no checking is done whether the byte exists. SEGNCPY will copy data until			
	the termination byte is found - with all consequences! Make sure that it is in			
	a valid range!!!			

3.14.6 Standard Driver

Description: Standard driver

Driver-Name: -

GDA: _hstddrv Header-File: STDFNC.H

This driver is in the WiOS-kernel and has no name!

Function Name	Number	Function	
ADDINT	1	add an interrupt	
Arguments	T_TASK	0252 task/driver's handle	
	T_SEG	032767 segment	
	unsigned	4000h7fffh address of interrupt-routine	
Return	0MAXINT	interrupt handle ('MAXINT' is in DEF.H)	
	255	no more interrupts	
Note	interrupt-routine may not EI		

Function Name	Number	Function	
DELINT	2	remove an interrupt	
Arguments	T_INT	interrupt handle	
Return	0	interrupt was removed	
	255	interrupt could not be removed	

Function Name	Number	Function
DRAG	3	starts a drag-operation
Arguments	-	
Return	-	
Note	under construction! Do not use yet!!!	

3.14.7 Task Driver

Description: Task driver

Driver-Name: -

GDA: _htaskdrv Header-File: TASKFNC.H

This driver is in the internal WiOS-part and has no name!

Function Name	Number	Function
-	1	used internally
Arguments		
Return		

Function Name	Number	Function
LOADTASK	2	load a task from disk
Arguments	char *	pointer to drive+path+filename string
Return	-1	if task could not be loaded
	0252	handle of task

Function Name	Number	Function
-	3	used internally
Arguments	-	
Return	-	

Function Name	Number	Function	
KILLTASK	4	remove a task from memory and free all segments and interrupts	
Arguments	T_TASK	task handle	
Return	0	event was sent	
	-1	task does not exist	
Note	not yet supported!!!		

Function Name	Number	Function
S_NAME	5	search by name if a task/driver is currently present (loaded)
Arguments	char *	pointer to name-string of task to be searched
Return	0	not found
	<>0	address of data block
		offset len
		+0 2 handle of found task/driver
		+2 2 version number
Note	String must	be between 4000h and 7fffh

Function Name	Number	Function		
ADD_EVENT	127	send an event to a task		
Arguments	unsigned	address of event-block		
		member	function	
		task	handle of the destination task	
		event	event	
		array[]	12 info-bytes for the receiving task	
	char	handle of th	e sending task (filled by WiOS)	
Return	-			
Note	event-block must be outside of page 1 & 2 so the receiving task can access			
	the block without segment switching. It is recommended to use '_eventb from GDA to store events			

3.14.8 Window Driver

Description: Window driver Driver-Name: "Window Driver"

GDA: _hwindrv Header-File: WINFNC.H

The function argument-explanation require the definition of struct WINSTR windat;

as a global variable in the C source-code to have access to the window-structure via names.

If there are problems with some window-specific words in the following list, refer to the illustration of chapter 3.7.1 - *Icons*.

Since this is the most complex driver, there are some functions which are not yet implemented. Still there is enough functionality to write normal applications and most of the missing functions (some of them are not mentioned by name, so you won't get to know what was planned and therefore you might probably not miss them >;->) can be simulated with some 'work-arounds'. Also remember that although the functions have been tested intensively there are cases which have never been tested before - especially working with pane windows.

3.14.8.1 The Window Structure

Туре	Name	Desc	ription	Note		
int	handle	windo	w-handle	Identifies each window - no windows have the		
				same handle		
int	Х	absol	ute position	on screen		
int	У			T		
int	nx	work-		The visible work-area is always guaranteed to be free for the task's use. Window-icons do NOT use		
int	ny	heigh work-	t of visible area	this area since WiOS puts them 'around' the window - except for the following cases, where the icons are copied into the user-area: Back-, Close- or Toggle-Size-Icon, if there's no title-bar Resize-Icon, if there is no scrollbar		
int	VX	virtua	I width of	Is only needed for calculation of the scrollbars and may be 0 if you have none		
int	vy	virtua windo	I height of			
int	scrx	horizo		always contains the absolute offset in pixels to the upper left corner of the window (like SET SCROLL		
int	scry	vertic	al scroll-	in basic) and is not needed without scrollbars		
int	minx	min. v	width of ow	contains the minimum size of the window, which can be scaled by the user - does only prevent the		
int	miny	windo		user to create smaller windows using the mouse, NOT changes made manually from a task!		
int	maxx	max. windo	width of ow	see minimum-size		
int	maxy	max. windo	height of ow			
int	behind			ndow in front (see below) heck the level of the window		
T_TASK	task	task v create windo	ed this	used internally to determine which task has to be informed if the user drags a window and to close all related windows if a task is closed		
char	winflag	bitma	pped windo	ow-area flags		
		bit		function		
		7	unused (m	nust be 0)		
		6	not yet imp	plemented (must be 0)		
		5	not yet imp	plemented (must be 0)		
	1	4		plemented (must be 0)		
	1	3		at on arrow-icons		
		2		pane-window (see 'parent' below)		
		1		n be dragged (via title bar)		
		0	1: WiOS c	from task needed can redraw the whole window (not yet implemented)		
char	iconflag		• •	ow-icon flags		
	1		fies the WiOS-controlled icons of a window			
		bit	function			
		7	unused (must be 0)			
			6 horizontal scrollbar			
			5 adjust-size icon			
			4 vertical scrollbar			
			3 toggle-size icon			
			2 title-bar			
		0	1 close icon			
	<u> </u>	U	back icon			

WOS - Application Programming Documentation

Туре	Name	Descr	ription	Note
char	workflag	bitma	pped windov	w-work-area flags (see below)
		bit		function
		7	unused (m	ust be 0)
		6	unused (m	
		5		k notifies task
		4		er work-area notifies task (for drag & drop)
		3		s task (once)
		2		s task (always)
		1		continually while pointer is over work area
		0	ignore all c	
int	parent	-1: 0255 Pane paren stay d If a pa be clo	 handle of windows are t window is lirectly in fro arent window sed. 	s a parent window (i.e. can have pane windows) the parent window, if window is a pane window e 'connected' to their parent windows, i.e. if a opened on front or on back, all pane-windows will nt of the parent window. v is closed, all connected pane windows will also ndows are NOT moved in the x- and y-position, if
char	statflag	bitma	pped windov	w-status flags
		bit		function
		7		
		6		
		5		unused (must be 0)
		4		,
		3		
		1	window io r	ninimized (not yet implemented: 01)
		0		minimized (not yet implemented: 0!) maximized (not yet implemented: 0!)
int	realnx			dow on screen
int	realny			
char	dummy[25]	'real' height of window on screen These bytes are to fill the window data block up to 64 bytes and are		
Gilai	Gaining[20]			e use, so always fill them with 0

3.14.8.2 Window Driver Functions

Function Name	Number	Function		
CREATE_WIN	1	add window to list and send open-event		
Arguments	unsigned	address of window-data-block		
		needed members:		
		x,y		
		nx,ny		
		vx,vy		
		scrx,scry		
		minx,miny		
		maxx,maxy		
		behind -2 back		
		-1 front		
		0255 specified window		
		if window is a pane window, 'front' always means on top of the		
		pane window block, and 'back' means directly above the		
		parent window (on back of pane window block)		
		winflag		
		iconflag		
		workflag		
		parent -1 window is parent		
		0255: handle of parent window		
		statflag must be 0		
		dummy[25] all 25 bytes must be 0		
Return	-1	window could not be created		
	0255	handle of the new window		
Note		the members handle, task, realnx and realny are filled by WiOS, no matter		
		valid data or not! This function always creates a new window		
		ATE_WIN' the window is NOT drawn on the screen! It is only		
	registered a	nd an 'OPEN ' event is sent to the task after the next polling!		

Function Name	Number	Function	
OPEN_WIN	2	change data of existing window in list, calculate new position and send redraw-events to all affected tasks	
Arguments	unsigned	pointer to window-data (see above)	
Return	-1	window could not be opened	
	1255	number of windows opened (incl. pane windows)	
Note	Window-data must be valid and complete so call GET_WIN_STATE before!!! If the behind value changes, all pane-windows (if window is a parent) are also changed so they are always directly above the parent window be sure use this function only on windows belonging to your task!!!		

Tr.				
Function Name	Number	Function		
CLOSE_WIN	3	remove window and all connected pane-windows from list and send redraw-events lower window's tasks		
Arguments	unsigned	0255 handle of window to be closed		
Return	-1	window could not be deleted		
	1255	number of windows closed (incl. pane windows)		
Note	be sure use this function only on windows belonging to your task!!!			

Function Name	Number	Function
GET_WIN_STATE	4	copy window-data from list to task's memory
Arguments	unsigned	address where the data has to be copied member 'handle' must contain the window-handle
Return	0h-FFFEh	address of window-data (=the address sent)
	-1	window could not be found

Function Name	Number	Function	
DRAWFRAME	5	draw window frame	
Arguments	int	destination x-coordinate	
	int	destination y-coordinate	
	unsigned	pointer to address of window structure	
		member 'handle' must contain the window-handle	
	char	must be '0'	
Return	-		
Note	no validity checking for the window-handle is done is called automatically by 'GETWIB'		

Function Name	Number	Function		
GETWIB	6	get window-information-block for redraw		
Arguments	unsigned	0255 handle of window		
Return	0h-FFFEh	address of WIB		
	-1	window could not be found		
Note	For the structure of the WIB (window-information-block), see chapter			
	3.19.1.2 - Redrawing Windows			

Function Name	Number	Function
COPYWIN	7	copy graphic area to display-screen using the redraw-stack
Arguments	unsigned	address of WIB
Return	-	
Note	VDA is changed WIB must be outside page 1 and 2 ('GETWIB' always returns a correct address)	

Function Name	Number	Function
-	8	used internally
Arguments	1	
Return	-	

Function Name	Number	Function
DRW_MENU	9	draws a menu corresponding to the task's data
Arguments		
Return		
Note	for detailed information on menus see chapter 3.18 - Menu Reference	

Function Name	Number	Function
CHK_MENU	10	check coordinates and return the item number
Arguments		
Return	-1	no item at these coordinates
	032767	number of item
Note	for detailed information on menus and return values see chapter 3.18 -	
	Menu Reference	

WOS - Application Programming Documentation

Function Name	Number	Function
-	11	used internally
Arguments	-	
Return	-	

Function Name	Number	Function
-	12	used internally
Arguments	-	
Return	-	

Function Name	Number	Function
-	13	used internally
Arguments	-	
Return	-	

3.14.9 Direct Calls

WiOS has calls which can not be accesses via the drivers. They must be called directly.

The variables are declared in 'GDA.H'. The list shows the usage of the commands in C-code.

```
(* poll)((unsigned)mask)
```

Calls the poll routine with an event mask (events whose bits are set in the mask are not sent to the task). It returns the address of the 'event block'. This block is defined in 'DEFSTR.H' and has the structure described in chapter 3.15.1 - Event Block Structure.

```
(* caldrv)( (char)d handle, (D FNC)func, (unsigned)arg1, (unsigned)arg2, ,...)
```

Calls the driver-caller, a routine which calls the driver with the handle 'd_handle', and sends the function number as the first parameter, then the arguments.

```
(*_cal_seg)( (unsigned)addr , (T_SEG)seg , (char *)block )
```

Only for tasks!!! Switch to the multi part segment 'seg', call the address which is pointed to by 'addr' and send 'block' as the address of 'adrblk'. The stack pointer of the old segment is saved and the one of the new segment is restored. The address of the routine to save the stack, restore the caller's segment and restore the old stack is put on the stack of the called function, so the return from a multi part function is simply done with a 'RET' instruction or with normal end-of-function in C. This function always returns a 16-bit value (in HL) to the caller function. Multi part functions can be nested as long as the stack does not underflow the 16k of the task or (which is more likely) the task's code is overwritten. No checking is done for that.

```
(*_dcal_seg)( (unsigned)address , (T_SEG)segment , (char *)block )
```

Only for drivers!!! Switch to the multi part segment 'seg', call the address which is pointed to by 'addr' and send 'block' as the address of 'adrblk'. The stack pointer is not changed, since drivers use the WiOS-stack. The address of the routine to restore the caller's segment is put on the stack of the called function, so the return from a multi part function is simply done with a 'RET' instruction or with normal end-of-function in C. This function always returns a 16-bit value (in HL) to the caller function. Multi part functions can be nested as long as the stack does not underflow the upper barrier of the GDA - else strange things may happen and the system will be instable. Since there's more than 4k of free stack size, this should never happen (I know, I know, 10 years ago, somebody said '640K should be enough for everyone!')

3.15 Event Reference

The following events are defined as the 'standard' events and event handlers should be able to deal with them.

The following list gives their name, number and the needed arguments if you send the data - and this is also the structure for the task receiving the data. In contrary to subroutine calls, these events do never return a value since the program receiving the events is not called immediately. An event is just a block of data which is put on a stack, telling the destination task after the polling that it should do something. Due to this fact, you should care about sending the correct data. Keep in mind that it is normally not your program receiving the data you send, and tasks from other people might not have such a powerful error-handling routine like your tasks have. Also, the events are sent one by one after each polling - so if you put 50 events on the stack, it will take a while to 'clear' it.

Event names with an asterisk (*) are sent by WiOS and should <u>NOT</u> be sent by any task. They should be handled as 'receive-only'!!!

3.15.1 Event Block Structure

Every event contains a block of 16 bytes for data storage:

Туре	Name	Description
T_TASK	task	handle of destination task
D_FNC	event	event number
unsigned	array[6]	for data storage
char	sender	handle of source task which sent this event

This structure is defined as 'EBSTR' in 'DEFSTR.H'

3.15.2 Event Definition

Event names are in 'EVENTFNC.H'

Function Name	Number	Function
E_NULL	0	Null-Event
Arguments	-	

Function Name	Number	Function
E_REDRAW	1	window redraw request
Arguments	array[0]	handle of window to be redrawn
Note		

Function Name	Number	Function
E_SCROLL	2	window scroll request
Arguments	task	sender (this task)
	array[0]	handle of window to be scrolled
	array[1]	horizontal scroll offset (absolute)
	array[2]	vertical scroll offset (absolute)
Note	An 'E_WOP	EN' event is sent immediately after the scroll event to the task.
	The task on	ly has to determine whether the new scroll offsets are accepted
	or not and, if yes, update the window data.	
	Tasks which send a scroll request to windows of other tasks should take	
	care to send	I the 'E_WOPEN' event <u>immediately</u> after it.

Function Name	Number	Function
E_WOPEN	3	window open request
Arguments	task	sender (this task)
	array[0]	handle of window to be opened
	array[1]	behind value
	array[2]	upper left x,y-coordinates of window position
	array[3]	
	array[4]	width and height of work-area
	array[5]	
Note	Width and h	eight are the work-area dimensions, not the real window size

Function Name	Number	Function			
E_WCLOSE	4	vindow close request			
Arguments	task	sender (this task)			
	array[0]	handle of window to be closed			

Function Name	Number	Function	
E_PNTOUT	5	pointer has left window	
Arguments	array[0]	handle of left window	

Function Name	Number	Function	
E_PNTIN	6	pointer has entered window	
Arguments	array[0]	handle of entered window	

Function Name	Number	Function			
E_MCLICK	7	mouseclick			
Arguments	array[0]	handle of window which has been clicked			
	array[1]	coordinates of pointer where click was (absolute)			
	array[2]				
	array[3]	bitmapped click-type (bit is set if clicked)			
		bit function			
		0 left button is pressed			
		1 left button is held			
		2 right button is pressed			
		3 right button is held			
	array[4]	the time (16-bit-counter) when the click was performed			
Note		do not give the actual coordinates of the pointer but the			
	coordinates	of where the pointer was on click-time.			

Function Name Number Function	Function Name	Number	Function
-----------------------------------	---------------	--------	----------

WOS - Application Programming Documentation

E_EDRAG	8	8 end of a drag operation		
Arguments	task	sender (this task)		
	array[0] data-type			
	array[1]	detailed information on data		
Note	For the data-type see chapter 3.16.2 - Data Type Definition			

Function Name	Number	Function	
E_WKAREA *	9	pointer is over work-area (only if bit 1 of work-area flag is set)	
Arguments	array[0]	handle of window below pointer	

Function Name	Number	Function
E_MENU *	10	not yet supported
Arguments	-	
Note	does not need to be handled!!!	

Function Name	Number	Function	
E_SHUTDOWN	11	requests the task to be closed	
Arguments	-		
Note	does not need to be handled!!!		

Function Name	Number	Function			
E_USERMSG	16	user definable			
Arguments	task	sender (this task)			
	array[0]	user-message code			
	array[15]	user definable			
Note	must be handled even if the task shall not receive external data!!!				

3.16 Data Exchange Specification

These events are the so-called 'user-events'. They are sent by the E_USERMSG event and, although the data has no effect on WiOS itself, you should use the following definition for standard data transfer to other tasks so every programmer who read this documentation can write applications which can handle this kind of data (well, I'm quite sure there will be no programmers writing programs for WiOS without having read this;)

3.16.1 Execution of a data exchange

To send data to other tasks, a special procedure is needed. First, an end-of-drag-event ('E_EDRAG') with the basic data-type must be sent to the destination task. Although it could be possible that the destination task masked out this event, it is strongly recommended not to do that. Normally, the destination task receives this event and has to answer. As soon as the source task (sender) receives the message 'SEND', it can send the destination task the 'DATA' message with detailed information about where the data is and what type and size this data has.

If a task shall not receive any data, it should also answer the end-of-drag event with the message 'NOSEND' so any source task which wants to send data will know that this task will not receive data.

Between these steps, it is required that these tasks poll. And, as if it wouldn't be hard enough already, each task should also be prepared that the next event after polling is not the answer but could also be a open or redraw request.

The program flow looks like this:

Source Task	Destination Task
send 'E_EDRAG' event with data-type	
poll	
	answer with 'SEND' or 'NOSEND' message
	poll
check for 'standard' events (and if yes, poll again)	
check for 'SEND' or 'NOSEND' message	
these steps are only necessary if the 'SEND'	message was sent
if 'SEND', send 'DATA' message with details	
poll	
	check for 'standard' events (and if yes, poll again)
	handle reception of data

From this flowchart you can see that this process is not too hard to handle. When viewing each task as separate, the sender has to send the 'E_EDRAG' event and poll as long as the 'SEND' or 'NOSEND' message comes, then send the data (if destination task wants to) and poll again.

For the destination task, it's even simpler: it evaluates the 'E_EDRAG' event and sends either 'SEND' or 'NOSEND' message, then polls. If it wants to receive the data, it polls until the 'DATA' message comes, else it's finished. Nothing else than sending the 'NOSEND' event must be done if the task doesn't need this kind of data.

From this, it is obviously that answering the 'E_EDRAG' event is essentially, or else the source task will 'hang' in an endless loop until an 'E_SHUTDOWN' event comes.

3.16.2 Data Type Definition

This is a weak point. Since I have never been programming on existing multitasking systems, the data type definition is in no way perfect nor were the data types created from their need in real applications - there has not been any application for WiOS yet! If you experience any unusability or if you can write a documentation for an existing, well-working data type definition, please let me know that <u>as soon as possible</u>. Since this means a change of the 'standard' definition given in this documentation, it can only be done in the next weeks during the alpha-testing phase. If several applications already exist, it will be impossible to define another standard. But in every case can new types be added to the existing definition - like in the GDA.

3.16.2.1 Data sent with the 'E EDRAG' event

Data type names in brackets are defined in 'DTDEF.H' and must be sent in array[0], the specifier in array[1].

Data type: 0 ('TEXT')
Specifier: 0 ('STRING')

Description: One or more text strings with no special function

Data type: 0 ('TEXT') Specifier: 1 ('FILENAME')

Description: One or more text strings with filenames to be handled / edited / opened

Data type: 1 ('GRAPHIC')

Specifier: 0..255 bits per pixel

Description: One rectangular bitmapped image (resolution is defined in the data itself)

3.16.2.2 Data sent with the 'E_USERMSG' event

Whenever sending user messages, array[0] holds the user-message code. That is, in this case, code 'DATA', which is defined in 'DTDEF.H'

Data	STRING				
Arguments	array[1]	ac	address of data block		
	array[2]	se	segment of data block		
	Data Blo	ck			
	Offset	Len	Valid Entries	Description	
	+0	2	065535	number of strings	
	+2	2	13071	number of segments used for data	
	+4,6,	2	03071	segment numbers with strings	
	after seg	after segment numbers			
		2	0	null-terminated strings	
			1255	number of fields	
	- if strings are null-terminated, the strings come directly after the '0'				
	- if strings are in fields of fixed length, there follows the length of the field				
				igth. After the length data comes the field-	
	formatted string data				
Note	If the stri	ngs ex	ceed 16k, it's	up to the program to switch the segments	

Data	FILENAME						
Arguments	array[1]		address of data block				
	array[2]	Se	segment of data block				
	Data Block						
	Offset	Len	Valid Entries	Description			
	+0	2	065535	number of filenames			
	after number of filenames						
				null-terminated strings			
Note	The filename data structure is exactly like the one of null-terminated strings - with the difference that the destination task does not have to insert this data somewhere but to open these filenames. In practice, that means to load these files – either in new windows for each file or for a slide show, one after each other, depending on the program. Also it's up to the destination task to parse the filename extensions or the files whether they are valid files for this task!						

Data	GRAPHIC						
Arguments	array[1]		address of data block				
	array[2] se		segment of data block				
	Data Block						
	Offset	Len	Valid Entries	Description			
	+0	2		unused			
	+2	2	13071	number of segments used for image			
	+4,6,	2	03071	segment numbers with image-data			
	after segment numbers						
		2	065535	width of image			
		2	065535	height of image			
				15-bit-mapped image data (see VDP-spec)			
Note	If an image exceeds 16k, it's up to the program to switch the segments						

This description <u>may</u> be insufficient since it has been developed only in theory. If you have detailed information on a better and more efficient data exchange standard, please let me know as soon as possible!

3.17 GDA Reference

This section describes the list of needed GDA variables.

The GDA area starts at C030h. It's end is open since it can be expanded in future. Do not change reserved variables. They are not protected, but changing them may result in system instability or a crash.

IMPORTANT: All GDA variables are pointers to the address of the variable or field.

Type definitions are in 'DEF.H'

Туре	Name	Offset	Len	Description		
T SEG		+0	2			
T SEG	*_tot_seg *_free_seg	+0	2	total number of segments in computer current number of free segments		
T SEG		+4	2	<u> </u>		
	*_tmp_seg			segment of temporary segment		
<unused></unused>		+6	2			
<internal use=""></internal>		+8	2	a consent le conflicte de la consent		
T_TASK	*_p1_task	+10	2	current handle of task in page 1		
T_SEG	*_p1_seg	+12	2	current segment in page 1		
char	*_p1rseg	+14	2	mapper segment number of current segment in page 1		
char	*_p1_slt	+16	2	slot of current segment in page 1		
<unused></unused>		+18	2			
T_SEG	*_p2_seg	+20	2	current segment in page 2		
char	*_p2rseg	+22	2	mapper segment number of current segment in page 2		
char	* p2 slt	+24	2	slot of current segment in page 2		
<internal use=""></internal>		+26	2			
unsigned	(*_caldrv)(.)	+28	2	WiOS-function call entry address		
T TÄSK	* hmemdrv	+30	2	handle of memory driver		
T TASK	* hstddrv	+32	2	handle of standard driver		
T TASK	* htaskdrv	+34	2	handle of task driver		
T TASK	* hfsdrv	+36	2	handle of file system driver		
T TASK	*_hgiodrv	+38	2	handle of graphic I/O driver		
T TASK	*_hgrpdrv	+40	2	handle of graphic driver		
T TASK	* hwindrv	+42	2	handle of window driver		
T TASK	* hextdrv	+44	2	handle of external driver		
unsigned	*_wiosver	+46	2	current version of internal WiOS part		
VOID	(*_wiosend)()	+48	2	WiOS shutdown call (only in Alpha-version!)		
unsigned	(* poll)()	+50	2	poll entry address		
<internal use=""></internal>	(+52	_			
internal acc		to				
		+106				
VOID	(*_dump)()	+108	2	entry call to make a memory dump on V9958		
				(argument is address)		
struct EBSTR	* eventblk	+110	2	address of event-block holding the data after		
				polling		
struct	* wibblk	+112	2	address of window information block		
WIBSTR			_			
struct	* winblk	+114	2	address of window data block		
WINSTR			-			
unsigned	* act font	+116	2	handle of current font		
unsigned	* std font	+118	2	handle of standard ('system') font		
<internal use=""></internal>		+120	2	Table 1 of the case of the cas		
char	* busyflag	+122	2	status of direct-write VDP status		
unsigned	* adrblk	+124	2	address of global, segment-switch		
			-	independent address block		
L	1	l	1	macpondont address blook		

WOS - Application Programming Documentation

Туре	Name	Offset	Len	Description			
char	*_mb	+126	2	current state of mouse buttons (bitmapped)			
				bit description			
				0	0 left trigger state (1=pressed)		
				1 right trigger state (1=pressed)			
char	*_cltype	+128	2	type of last mouse click (bitmapped)			
				bit description			
				0	left single click (1=clicked)		
				1	left double click (1=clicked)		
				2	5 7		
				_	3 right double click (1=clicked)		
unsigned	*_cltime	+130	2	time of mouse click (array of 2)			
						description	
				+0		time of last left-click	
				+2		time of last right-click	
int	*_coord	+132	2	current mouse coordinates (array of 2)			
unsigned	(*_cal_seg)()	+134	2			I for tasks to call multiple-part	
				functions			
<internal use=""></internal>		+136	2				
unsigned (*_dcal_seg)() +		+138	2	entry call for drivers to call multiple-part			
	functions			3			
<internal use=""></internal>		+140	2				
<internal use=""></internal>	nternal use> +142 2						

... to be expanded in future

3.18 Menu Reference

WiOS' menus and icons have to be handled by each task itself. Yet, there's no automatic handling of mouse-clicks in defined areas, but WiOS offers helpful functions to get the most of menus with an ease of usability.

There are two main functions you need to handle menus: 'DRW_MENU' to put the draw the menu on the screen and 'CHK_MENU' to check whether specific coordinates are in one of the defined points or not.

When talking about items in this section, a single menu point with a possible graphic icon is meant.

NOTICE:

The function CHK_MENU returns the number of the item on which the coordinates (normally the pointer) are, which are counted linear from the beginning of the menu-list. Each textentry, button, and box is one item. If the menu consists of a list with 5 entries and two options (in this order), the latter entries will return the item numbers 5 and 6, if the coordinates of the pointer are over them, since the first 5 entries have the numbers 0 to 4. A special type is the scrollbar: it contains of 5 items in the following order:

arrow up/left / page up/left / slider / page down/right / arrow down/right

Depending on whether the scrollbar is vertical or horizontal. To get to know more exactly how the items are evaluated, create a menu with different types in one list and print the return-values of the CHK_MENU-function (with the mouse-coordinates the coordinate-agrument) on the screen.

3.18.1 Menu types

The menu functions offer 5 different types of menus:

Option Text strings with an option box on the left.

Options are a block of one or more items where each item can be enabled or disabled separately.

Radio Button Text strings with a radio button on the left.

Radio buttons are a block of one or more items where only one item can be enabled. If another item in this block is selected, all other items will be disabled

List Plain text strings

Normally, lists are used for pull-down or pop-up menus

Scrollbar Graphical display of a given scale using a user defined scale with the possibility to select a certain area in the given scale.

Box Rectangular area

Boxes are <u>not</u> visible on the screen. They can be used for checking any rectangular area, e.g. for menus that cannot be handled with the above menu types.

The complete menu data of a window is normally placed in one big data block. Such a block uses the basic idea of a macro language, but at a very low level, i.e. word-, byte- or bit-mapped. The idea behind this is to have one structure where each item has its own number. The coordinate checking has to be called only once to get the item number of the desired coordinates, no matter if you have only a pull-down menu or a complex structure with many options, radio buttons and scrollbars.

3.18.2 Menu Features

Option / Radio Button Items can be printed in both directions, from top to bottom, and from left to right. Independent from the direction will the coordinate checker validate coordinates beginning from the left - at the icon - to the right end of the <u>text</u> of each separate item, i.e. if you check an area right to the text, you will receive no item number.

When writing items in the destination from left to right, WiOS will calculate the width of the longest string in the option / radio button block and use this as the standard distance between each item. Although the distance is equal, the items begin with the icon and end with the right end of the text of each item.

Disabled items will be printed in grey (instead of black)

Whenever a item is selected, it is up to the task to set (and for radio buttons: reset all other) icon-enabled-bits.

Lists Items can only be printed from top to bottom. The coordinate checker calculates the width of the longest string and will - in contrary to the option / radio button - use this width for each item in the list block.

Disabled items: see option / radio button.

Items can have an arrow icon on the right border to indicate the existence of a sub-menu - which must be opened as a new window by the task.

Scrollbar They can be adjusted in horizontal and vertical direction, and they are always separate (no combination in one scrollbar block possible, but there can be more than one scrollbar in a menu block!!!)

Disabled scrollbars will not have any visible difference in graphic - the task has to indicate that separately on the screen.

The scale of the 'virtual size' (similar to windows), the size on the screen in pixels and the size of the area represented by the slider (see 'work-area') can be chosen freely. Yet, there are some problems drawing very small sliders.

Slider-handling, scrolling and page up/down must be completely processed by the task - the step-size is not pre-defined. If the user clicks on the slider and drags it around, it is up to the task to trace the mouse and show the slider on the screen. This might be fixed, if wanted.

Box It can be freely defined with a coordinate pair (upper-left / lower right)

3.18.3 General conventions about menus

If items overlap, the checker will return the item number of the first valid area of the coordinates. If they overlap, it will return, not as drawn on the screen, the number of the 'lowest' area (e.g. if two boxes intersect, the number of the box drawn first will be returned although only the second box is visible since it has 'overwritten' the first box).

Every item can be disabled separately. Disabled items are still displayed on the screen but are not checked by 'CHK_MENU'. The item numbers of the following items stay the same, so for example if you disable the first item of a list box (which has number '0'), you will not be able to check it any more, but the second item still has the number '1'.

Border styles can be chosen for each item block except for the scrollbars. At the moment, only two border styles are valid: no border and filled box border with standard background color (which can not yet be altered). The border distance in pixels is doubled for x-direction because in 512x212 resolution, a pixel is twice as high as it is wide.

3.18.4 Menu-Block Structure

The menu block is one big field of data where one or more menu types can be defined.

The following symbols are used in the structure definition:

```
prefix
```

- b Byte (1 byte) w Word (2 bytes)
- t Text (variable byte length)
- # definition of valid values for 'D' (see suffix)

<u>suffix</u>

- N any number
- D number from a pre-defined list
- S and character string 0 must be this value

Valid border styles:

- 0 no drawn border
- 1 filled box with background color

Definition

bN number of item blocks

type

bĎ

#0 Option

Once for each option block:

wN font-handle wN x-offset wN v-offset

w0 width of option-block (filled by WiOS)w0 height of option block (filled by WiOS)

bD border style

bN distance of border in pixels

bD Direction #0 top to bottom #1 left to right

For each entry:

entry status

bD Entry Flags (bitmapped)

bit reset (0) set (1) 0 end of list valid entry

valid 1 disabled enabled (selectable)

selection 2 not selected selected

3..7 unused (must be reset)

tS Null-Terminated String

continued at Entry Flags (until bit 0 is reset)

WOS - Application Programming Documentation #1 Radio Once for each radio block: $w \\ N$ font-handle x-offset wN wN y-offset w0 width of option-block (filled by WiOS) height of option block (filled by WiOS) w0 border style bD distance of border in pixels bΝ Direction bD #0 top to bottom #1 left to right For each entry: Entry Flags (bitmapped) bD reset (0) bit set (1) entry status 0 end of list valid entry valid 1 disabled enabled (selectable) selection 2 not selected selected unused (must be reset) 3..7 **Null-Terminated String** continued at Entry Flags (until bit 0 is reset) #2 List Once for each list block wN font-handle wN x-offset wN y-offset w0 width of option-block (filled by WiOS) w0 height of option block (filled by WiOS) bD border style bN distance of border in pixels For each entry: Entry Flags (bitmapped) bD reset (0) bit set (1) entry status 0 end of list valid entry valid 1 disabled enabled (selectable) sub list no sub list sub list (arrow on the right) 2 3..7 unused (must be reset) **Null-Terminated String** continued at Entry Flags (until bit 0 is reset) #3 Scrollbar Entry Flags (bitmapped) bD reset (0) bit set (1) 0 not valid valid

bD Entry Flags (bitmapped) bit reset (0) set (1) 0 not valid valid WN x-offset WN y-offset WN visible size of field in pixels WN virtual size of field in pixels WN real size of field in pixels WN scroll offset of scrollbar in pixels

WOS - Application Programming Documentation

#4 Box bD Entry Flags (bitmapped) bit reset (0) set (1) 0 not valid valid entry status wN x-offset y-offset wN width wN height wN border style bD bΝ distance of border in pixels

If you miss any menu structure and think it would be worth implementing in WiOS, please let me know of your concrete ideas.

3.19 Programming Tips

It is not only of importance to know which functions do exist, but also in which order and, that's even more important, <u>when</u> they should be used. Although there are sample programs in this documentation, you can look up the way how special things are managed in this section.

For programming applications in a graphic-based multitasking system, you often need more than just calling one function to do common procedures. This can not be derived from the functions itself. You need to know how these procedures which can only be done with the cooperation of two or more functions are handled.

During your first programs, it is advisable to use the structure in the sample-programs as-is, no matter if you understand why it is done like this or not, since there are other things you should focus on first - like working with the ASCII-C compiler or testing the graphic functions. If you think you understand the basics of WiOS and want to do real applications (or if you just want to know anything), go on reading.

3.19.1 Windows

To put windows on screen, there are two major steps necessary. One step does the 'virtual' stuff for a window, that is the calculation and storing of the window in memory so WiOS is able to handle all user-entries concerning the modification of the window using the mouse. The second step is changing the window 'physically', i.e. draw and move the window on the visible desktop.

3.19.1.1 Opening Windows

First of all, you can only draw windows in the event-handling routine - after polling. There is no difference whether the user, your task or another task wants to redraw a window of a task.

User

After polling, the task receives an open-window-event whenever a window was dragged, rescaled,... by the user. The task has to call the 'OPEN_WIN' function which adds the redraw-event. After polling, the task receives this event and must redraw the window on the screen.

Your Task

Creating a new window can be done with the 'CREATE_WIN' function of the window driver. This will add your window to the window list and fills in some variables. To draw the window on the screen, you have to arrange that you will receive a redraw-event after polling. This event is sent by the 'OPEN_WIN' function. To get the complete data of the new window, call 'GET_WIN_STATE'. This data can be sent to the 'OPEN_WIN' function, which adds the redraw-event.

Moving an existing window is done as described in the steps before without the 'CREATE_WIN' function.

Other Tasks

Forcing other tasks to move their windows can be done exactly by sending them the open-window-event - which is exactly the same way WiOS informs the tasks after the user has moved a window. The corresponding task will then call the 'OPEN_WIN' function which sends the redraw-event to itself, and after polling, it redraws its window.

Why sending the open-event, let the task call 'OPEN_WIN' before the redraw-event is sent? Although there can already be given some restrictions for the user not to move or rescale the window as he wants, it is of desirable to have more control over the window. The basic idea behind this is to 'ask' the task before giving the command to redraw the window. The task may then determine whether or not to accept the changes. These changes are sent by the 'OPEN_WIN' function. It's up to the task to send the changes to the window driver and update the window entry in the list or to reject these changes by ignoring this event.

Why cannot the task redraw the window directly after the open-event has been sent? There is more than only one window on the screen. Changes on one window can affect the others as well. These changes can not be predicted before the open-event sends the final window-data to the 'OPEN_WIN' function. If a window is changed (i.e. moved, rescaled,...) on the screen, other windows may be visible now. These windows also have to be redrawn. The window-driver does not send extra open-events to the tasks of these windows, but only redraw-events - since there is nothing changed with these windows. This requires a separate redraw routine which must be independent of the open-routine.

Regarding these facts, a task using windows has to have the following routines:

- * window-creation-routine for the 'standard' windows (called only at the beginning)
- ★ window-redraw-routine (handles the redraw of the windows on screen)
- ★ window-open-routine (checks window-parameters and calls 'OPEN WIN')
- ★ event-handling-routine (calls window-open- or window-redraw-routine)

3.19.1.2 Redrawing Windows

The redraw-routine has to know a bit more than just the data in the window structure. This data can be ordered by the window driver and is called the <u>window-information-block</u> (WIB). It has the following structure:

T_SEG stackseg segment of the redraw-stack unsigned elements number of rectangular areas in redraw-stack struct WINSTR *windat pointer to window-data (outside page 1 or 2)

needed window-members are:

x,y nx,ny realx,realy

unsigned sx,sy start coordinates of window-copy (always 0,212)

unsigned offx,offyoffset for work-area

int moffx,moffy move-offset relative to last position

unsigned toffx,toffy title offset unsigned t_nx,t_ny title width

The redraw stack has the following structure:

unsigned x1,y1 upper left coordinate (absolute) unsigned x2,y2 lower right coordinate (absolute)

The task's redraw-routine has to look like this:

- call 'GETWIB' and store return-address
- if elements is equal 0, nothing is to do
- draw your own data at coordinates (sx+offx , sy+offy) that come with the 'GETWIB' function
- call 'COPYWIN'

For examples, look in chapter 3.22 - Sample Programs.

3.19.1.3 Scroll Requests

Whenever an 'E_SCROLL' event is sent to another task, must be taken care to send an 'E_WOPEN' event immediately afterwards!!! No polling should be done between the sending of the two events!!!

If there are protests, I might change WiOS to send an open window request automatically if a scroll event is sent.

3.19.2 User Input

One (yet) weak point in WiOS is that there is no built-in routine which allows the user to edit one or more characters. That has to be simulated by your program (i.e. you have to write your own routine) until the routine will be present in a future version of WiOS.

Getting the next pressed key:

- call the external driver with function 'GETKEY'

Wait for a current keypress:

- call 'RESKEY'
- loop until 'GETKEY' returns not equal 0

Only ASCII codes will be returned. Yet, you cannot check whether any control key has been pressed if there is no separate ASCII-code for the key with and without the control key (e.g. SHIFT+'A' returns a different code that 'A', but SHIFT+CURSOR returns the same code as CURSOR only).

This might be fixed in a future version if there's need for that.

3.19.3 Using the Temporary Segment

Whenever you need to store 16k or less data temporarily, for example for calculation purpose or for data transfers, you may use the system's temporary segment. Its number is in the GDA ('_tmp_seg').

This segment is also used by the system as a temporary page (e.g. whenever data from disk is loaded) and it might be used from other tasks. So do not store your data there for too long, or it might be altered. If you want to use this page, be sure not to load from disk or poll - in these cases, your data is lost for sure!!! Instead, you can allocate a segment temporarily and free it later, if you need it to hold the data for several pollings (which is strongly recommended, since frequent polling is an essential part of co-operative multitasking).

The temporary segment is always in the main-slot.

3.19.4 Mouse Pointer

The mouse pointer is moved in interrupt, so every 1/60 second, a function which reads the new mouse coordinates and puts the pointer to the screen is called. The sprite data must be written to VRAM, and therefore it is needed to set up the write address and send the coordinates to the VRAM. From this fact arises the problem that there can't be another program which writes data linear to VRAM through P#0 or sets up a VDP command. Linear data would be written to the correct place as long as the interrupt function is called, then any other data would be written behind the sprite coordinates. If a VDP command is set up, the register pointer would be changed to R#0 from the interrupt routine and all other register data would be written to R#3 and up. One possibility is to disable the interrupts, but if there are other tasks running in interrupt like a music replayer, they would halt as well.

This is handled with the variable 'busyflag', which can be found in the GDA as '_busyflag'. Whenever this flag is set to a value not equal 0, the pointer update on the screen will not be processed. If you want to perform a direct write to VRAM via P#0 or set up a graphic command, set the busyflag, and if you finished, reset it. When writing a lot of linear data, you could as well set up a graphic box and send the data through P#2 to prevent long phases where she pointer does not move.

3.19.5 Sending data to other program parts using the address block

Sometimes, it is necessary to send other programs more than just one or two arguments. This works transparently with the 'caldrv' function, bus there are cases where the data must be stored in a place which is not affected by segment switching by the programmer. One of these cases is the calling of multi-part tasks. Since you know that it is taboo to write data in the GDA or somewhere outside of a task's own segments, there is a place where a task is allowed to store data: this area is pointed to by the GDA variable '_adrblk'. This area has place for 16 bytes which can be defined freely. This is not too much, but it is enough for most cases.

Another use for the address block is to eliminate a weak point of C: the fact that only one parameter can be returned. Sometimes, it is necessary to return more than one value. These values could be stored in this area.

Remember that this is not a place to store data permanently since other tasks may use it as well. So, similar to the temporary segment, store only very immediate data at this address, for example if you call a function in another segment of a multi part application.

3.19.6 Stacks

Every task has its own stack-pointer inside the actual segment. Whenever a task is loaded, this stack is set to the top of the 16k page (i.e. to 8000) Whenever calling WiOS-functions, the is stored data on the stack - also whenever a task is called from WiOS. Since the code size of every part in an application may vary, there's no limitation or internal checking whether the stack overwrites areas of the code. If you experience problems with applications which code size is 15k or more, try to move some functions to a new part. Normally, this should not happen, but if you nest 100 inter segment calls to multi part functions, your stack might come dangerously near your program code.

3.19.7 Program Termination

For WiOS applications, it is important never to end a task by returning from the main function which is called after loading. Also, you may never use the 'exit()' routine, CTRL+STOP or any other DOS abort routine, since the memory pages are altered and WiOS' interrupt driver is still active. The task must be closed with the 'KILLTASK' function of the task driver. Since this function does not work in this Alpha-Release, you should use the 'ESC' key which is scanned at each polling. This routine shuts down WiOS, restores the original memory segments, hooks out the interrupt driver and returns to DOS.

3.20 Compilation

The creation of a WiOS task requires several steps, which differ slightly for single and multi part applications.

First, a compilation batch file is needed (let's call it 'COMP.BAT'). If should basically contain the following commands:

The option '%2' at 'cg' should normally be '-r2000' to have enough space for functions. This can be set permanently to this value so you don't have to type it for each compile, but if changes are needed, you have to edit it separately. For C files which require a special pool:symbol:hash proportion for the parser and memory set-up (-rnnnn) for the code generator it is advisable to create separate batch files for them (for example TASK1.BAT to compile the file TASK1.C) where the '%n' arguments can be replaced by concrete names and options, so you have only to type 'TASK1' to compile TASK1.C.

For advanced C users, it is preferable to have two compilation batch files. If the code generator wants a different '-rnnnn' value, it is not needed to create the .TCO file again. Therefore, here are my personal batch files:

COMP.BAT

```
cf -r7:5:2 %1 fpc -u %1 comp2 %1 %2
```

COMP2.BAT

```
cg -r2000 %2 %1
m80 =%1/m
del %1.tco+%1.mac
```

The first batch file is called with 'COMP filename' (without extension!!!) and creates only the .TCO file and checks the function parameters. If no errors occur, it calls COMP2 with the arguments from the command line.

In the second batch file, the code generator and assembler is called, and the non used files are deleted.

The advantage of using two files is to be able to 'continue' the compilation process at the code generation, since the parsing is not needed is the code generator has a memory error. Also, you can override the '-r2000' option by writing a second argument in the command line like:

```
comp2 task1 -r1500
```

The code generator receives the line 'comp2 -r2000 -r1500 task1' and uses the later options rather than preceding options.

3.20.1 Single Part Applications (up to 16k code)

They consist (normally) of several C files. To link the .REL files, a line similar to the following is needed:

```
c:\c\l80 /p:3ff6,taskhead,task1a,task1b,task1c,task1d,...,gda,c:clib/s,c:crun/s,c:cend,task/n/e
```

This creates a single .COM file with about 16k of zeroes and the code beginning at 3ff6h - so that the real application (without the file-header) starts at 4000h.

Then MAKDRV must be called to cut the preceding 16k and to fill the correct values in the header like:

```
makdrv task.com task.tsk
```

This creates the WiOS-executable file TASK.TSK. Its file size should be smaller than 16k. If the .COM file has more than >16k of code (i.e. is larger than 32k), MAKDRV will not proceed.

After the task is created, the .COM file is no longer needed and may be deleted.

To make things easier, this process should also be done in a batch job. For that purpose, let's suppose the task file shall be called TASK.TSK, the main source code TASK.C and the standard header file TASKHEAD.MAC. Then a file called LT.BAT (which means Link Task) has to be written:

LT.BAT

```
c:\c\l80 /p:3ff6,%1head,%1,%2,%3,%4,%5,%6,%7,%8,%9,gda,c:clib/s,
c:crun/s,c:cend, %1/n/e
makdrv %1.com %1.tsk
del %1.com
```

If TASK.C is the only file needed for the task, just write

LT TASK

to link TASKHEAD and TASK, create the .TSK file and remove the .COM file. If you have another part called 'TASKB.C' which has to be linked, just add 'TASKB' at the end, as well as up to 9 other parts. If you have more parts, it's advisable to write an extra batch file for the linking process.

The creation of your task is then simply done with two commands:

```
COMP TASK
LT TASK
```

If no errors occurred, start WiOS (W.COM) and watch what your program does.

3.20.2 Multi Part Applications (more than 16k of code)

The first part (with the init routine) is created in exactly the same way as a single part application until the .COM file is done. Every part (normally) also consists of several C files.

All the following code parts are compiled with the same steps mentioned above - completely independent of the other parts- but they differ in the way they are linked. They do not have the standard header file for tasks, since it is only needed at the beginning of the .TSK file. Their link line looks like this:

```
c:\c\l80 /p:3ffe,etskhead,task2a,task2b,task2c,task2d,..,gda, c:clib/s,
c:crun/s,c:cend, task2/n/e
```

For a multi part application, you should have two or more .COM files.

The .COM files of multi part applications should never be deleted, although they use a lot of disk space, because they are always needed to create the .TSK file. This is done by:

makdry task.com task2.com task3.com ... task.tsk

Write as many parts as you want into this line and the name of the .TSK file at the end.

One problem is, of course, the limited size of the command line, but if there's really somebody writing such a huge application that suitable filenames (1.com, 2.com, 3.com,...) do not fit into the line, I promise to write an extended MAKDRV program ;-)

For this kind of task, it's also easier to have some batch files for compilation. For that purpose, let's suppose the task file shall be called TASK.TSK, the main source code TASK.C and TASKB.C (for part 1) and TASK2.C and TASK2B.C (for part 2), the standard header file TASKHEAD.MAC and the extended header file ETSKHEAD.MAC. Then a file called LT1.BAT to link the first part has to be written:

LT1.BAT

```
c:\c\l80 /p:3ff6,%1head,%1,%2,%3,%4,%5,%6,%7,%8,%9,gda,c:clib/s,
c:crun/s,c:cend, %1/n/e
```

This batch file is the same as LT.BAT without the calling of MAKDRV and without deleting the .COM file, which is needed for later creation of the task.

LT TASK TASKB

links TASKHEAD, TASK and TASKB, assuming you already compiled TASK and TASKB with the COMP batch file.

To create the other parts, you need another batch file because the header file is different. So create a file called LT2.BAT which is used to link all other parts.

LT2.BAT

```
c:\c\180 /p:3ffe,e%1head,%2,%3,%4,%5,%6,%7,%8,%9,gda,c:clib/s,c:crun/s,c:cend, %2/n/e
```

To link the sample files TASK2, TASK2B and ETSKHEAD, type

LT2 TSK TASK2 TASK2B

Now if you have the .COM files, if could be handy to write a new batch file to create the task. This file, called MAKETASK.BAT could look like:

MAKETASK.BAT

makdrv task.com task2.com task.tsk

3.21 Task Specifications

Multi part tasks do - regarding the fact that each part consists of a .COM file - not know anything of each other since they have all been compiled and linked separately. The 'interface' between them is the address table at the beginning of the page (about 4000h) and the WiOS functions which offer the possibility of transporting data and jumping and returning to and from the multi part segments. Therefore, when planning applications, you should take care of writing functions that have to call each other very often or use the same 'internal' data (i.e. in the task segment from 4000h to 7fffh) in one part - if possible - since every function which has to be called from outside must be declared in the address table. The best (theoretical) way is to write each function-complex which calculates a lot of stuff or does something special in an own part and have only one address in the address table - well, this is theoretical, but what I want to make clear is that it would be stupid to write all little functions which are called frequently in another segment since it would become very unhandy to call them via their address table entry and to access data not using the C variables but sending it with the address block or accessing them by a fixed address in the data segment. Anyway, any way is possible, and it's up to you to decide which one is best for you.

Important!!! The first part or a multi part application does <u>not</u> have a standardized address table. If other parts want to call a function in the first part, you have to set up an own address table in this part and give this address to the 'cal_seg' function. This address table may be anywhere at address 4008h to 7fffh and may be included in the standard header file as it's done in the extended header file.

3.21.1 Headers

A task's 'standard' header must have a list of information about the task. Every task must have this header. For multi part applications, the first part contains the header.

An 'extended' header is only for extended parts, i.e. part 2 and higher of a multi part task.

The headers are .MAC files to warrant the correct order of variable and pointer positions.

All names with a '@' are declared public and can be accessed as a normal variable in the C code. Labels which are not pointed to in a 'ds' or 'dw' instruction in another line can be left out. They are just here to make clear what this line defines.

Header files must be assembled with

M80 =filename/m

to create a linkable .REL file.

You can set the number of needed segments in the header file, so it is guaranteed that <u>if</u> your task runs, it <u>will</u> have enough segments, or else WiOS does not load it. The number of needed segments must be at least as big as the number of program parts because each part needs one segment. But this number can also be larger if a task needs more segments to store data permanently. You can add the number of segments to the number written in 'PPARTS' and write the result at the constant 'NEEDSEGS'.

Remember that this is only for the number of segments which are needed permanently to <u>run</u> the task. Theoretically, you can temporary allocate all available memory later if needed to speed some things up or to store huge picture data.

3.21.2 Standard Header Structure

normally, you only need to change these two lines (and the task name)!!!

NEEDSEGS equ 1 minimum number of segments needed for

this task (incl. data segments)

PPARTS equ 1 program parts (for multiple part tasks)

external header

WiOS_ver:dw0WiOS version needed to run driversegs:dwNEEDSEGSneeded segments to load driverp_parts:dwPPARTSnumber of program-parts in file

t_main: dw taskinit@ task's start-address

lastadr: dw @endx@-4000h len-info for loader, filled by L80

internal header (4000h)

tskname:dwtasknamepointer to task's namep_stable:dwsegtable@pointer to segment-numbersstack:dwtaskmain@Internal Stack Pointer of task

(must point to function taskmain in file!!!)
usedsegs: ds 2 total number of used segs (filled by WiOS)

variable part (4008h)

segtable@: ds NEEDSEGS*2 space for segment numbering during

allocation (filled by WiOS)

taskname: dw 0 task's version

db "Test-Task" Task's name

db 0 taskname termination

public tsknam@
public segtable@
extrn taskmain@
extrn taskinit@
extrn @endx@

end

3.21.3 Extended Header Structure

external header

lastadr: dw @endx@-4000h len-info for loader, filled by L80

internal header (4000h)

mp@seg@: dw 0 segment of 1st part with all task-infos

(filled by WiOS)

my@seg@:dw0segment of this part (filled by WiOS)stack:dw0Internal Stack Pointer of task (filled by

WiOS)

variable part (4006h)

func_1:dwtest1@address of routine 'test1' (4006)func_2:dwtest2@address of routine 'test2' (4008)func_3:dwtest3@address of routine 'test3' (400a)

.

extrn test@

public mp@seq@, my@seq@

extrn @endx@

end

3.22 Sample Programs

On the next three pages, you'll find a little example of a complete Desktop Publishing package including printer driver...

Well, not exactly, this section is more a reference for inexperienced WiOS programmers who just want to look how common routines are used and a look-up possibility for not-yet-C-programmers:-)

More examples can be found in the Application Creation Toolkit and, if there's demand, in a separate printed sheet.

What is essential for an application?

3.22.1 Pre-processor Directives

To get the optimum performance with the C compiler, the following lines should be on the top of each C file:

```
#pragma optimize time which has, as far as I know, no effect on the code to enable register allocation, if possible not to store all variables for every function.
```

If a function is recursive, you may write the same line and replace 'nonrec' by 'recursive' immediately before the function and add the line with 'nonrec' again after it.

Although this is the theoretical optimum, it's often better to disable register allocation so the code generator does not try to optimize the code. The program runs slightly slower and uses a few bytes more, but the compilation process will be up to twice as fast without optimization. As practice shows, a program has to be started several hundred or thousand times before the programmer thinks that it works without bugs. During this development phase, you can save a lot of time without optimization, and when the program works perfectly, it can be enabled in all source files for the final compilation to create an optimized code.

These lines are more likely for the real programmer:

```
#pragma optimize time
#pragma noregalo
#pragma nonrec
```

3.22.2 Header Files

The following header files are essential to write tasks in C:

```
#include <stdio.h>
#include "def.h"
#include "gda.h"
```

and one or more of the following are optional:

```
#include "eventfnc.h" events
#include "defstr.h" structures
#include "winfnc.h" ... see chapter 3.14 - Function Reference (WiOS Drivers)
#include "taskfnc.h"
#include "stdfnc.h"
#include "memfnc.h"
#include "grpfnc.h"
#include "giofnc.h"
#include "fsfnc.h"
#include "extfnc.h"
```

3.22.3 External Variables

There are variables defined by you (task's name) and WiOS (segment numbers) in the header file of each task. If you want to have access to them, they should be declared in the C file:

```
extern char *tsknam;
extern unsigned segtable[0];
```

The number in brackets can be set freely, since C does no boundary checking either. This just means that the variable 'segtable' is an indexed 16-bit variable.

With the variable 'segtable' you know where your task, all the other parts and the data segments are. To get the number of the first non-code segment, just use the number of parts of your task as the index. For example if you have a multi part task with $\underline{2}$ parts, the index 0 and 1 is used for the code segments and the index $\underline{2}$ and higher contains the numbers of the data segments.

When writing the task's name, remember to add 2 to the pointer, since the first two bytes of the task name contain the version number of the task.

3.22.4 Task Initialization

Whenever a task is loaded, WiOS calls the address pointed to by 'stack' in the first part of the task (see chapter 3.21.2 - Standard Header Structure). This can be used for task initialization, for example to check whether another instance of this task is already running or to check the presence of required drivers and, if necessary, to load them.

Important!!! This routine is not the real start of the task. It may not call the poll routine nor does any event have to be checked nor should windows be opened. Normally the init sequence is only for checking if the task finds everything which is required to run. This routine may return immediately without doing anything! The task 'officially' starts at the address pointed to by 't_main'. The handle of the task is passed as an argument of type char. This value may be either ignored or saved (to prevent a 'search for name'-call later to determine the handle of the task).

```
VOID taskinit(handle) char handle;
{
    printf("task init-routine: task has handle %d\n",(int)handle);
}
```

This function is necessary, but it's not necessary for the function to do something.

3.22.5 Main Routine

This is the function where the task begins. The function should never return. If you want to stop the task, use the 'KILLTASK' function of the task driver. For the Alpha-Release, see chapter 3.19.7 - Program Termination. This address is pointed to by 't_main' in the standard header file.

```
VOID taskmain()
{
    printf("task has started!!!\n");
    ...create_and_open_windows...
    ...poll...
    ...do_something...
    ...format_harddisk...
    ...print_virus_message...
}
```

It's up to your imagination to create a usable program. This is the function which has to set up anything which can be seen on the screen, handle user events, doing anything - the 'main()' function from 'normal' C.

3.22.6 Window Creation

Before you can see anything on the screen and let WiOS do all the work with user, you have to set up proper window data.

For this, and also for some functions later in the redraw and open section, the 'windat' variable should point to the task's own window block which is used to create and change windows, so somewhere in the C file, there should be a line like:

```
struct WINSTR windat;
```

Here comes an example of a perfect set-up of a window (so at least I can say <u>one</u> thing is perfect with WiOS;)

```
VOID creatwin()
int whandle, i;
       windat.x=50;
       windat.y=60;
       windat.nx=100;
                             /* work-area size */
       windat.ny=75;
       windat.vx=100*2;
                              /* virtual size (twice as much as work-area)*/
       windat.vy=75*2;
       windat.scrx=100/2; /* scroll offset (to the middle) */
       windat.scry=75/2;
       windat.minx=0;
                              /* no minimum limit */
       windat.miny=0;
       windat.maxx=0; /* maximum size is virtual size or screen */
      windat.maxy=0; /* size */
windat.behind=-1; /* open window on top */
windat.winflag=0; /* hmmm, see in the referring chapter */
windat.workflag=32+8+4+2; /* allow all clicks */
       windat.parent=-1; /* window has no parent -> it \underline{is} parent */
       windat.statflag=0; /* ... */
       for (i=0; i<25; i++) windat.dummy[i]=0;/* reset dummies */</pre>
       whandle=windat.handle=(*_caldrv)(*_hwindrv,CREATE_WIN,&windat);
       printf("created window has handle %d\n", windat.handle);
       (*_caldrv)(*_hwindrv,GET_WIN_STATE,&windat); /* get win data */
       (*_caldrv)(*_hwindrv,OPEN_WIN,&windat);/* send data for opening
                                                           the window */
}
```

There should not be any problems in understanding the code. One important thing is that 'whandle' should be stores in a global variable because it is the only point where the task gets to know the handle of the created window. This is important to redraw the correct window if a redraw request is sent. The 'GET_WIN_STATE' function before opening the window is necessary because WiOS filled some variables to the window structure like the window's task and the real window size including the icon frame.

3.22.7 Polling and Event Handling

Here's an example of what should an event handling routine be capable to deal with. We use the simplest type of main loop:

or the last two lines of code (for C cracks) so 'block' is not needed at all:

```
procevent((struct EBSTR *)(*_poll)(0));
```

For the description of the event block, see chapter 3.15 - Event Reference. The function 'procevent' should then handle all the necessary events and could look like:

```
VOID procevent(block)
struct EBSTR *block;
char c;
      printf("processing event %d\n",block->event);
      switch(block->event)
      {
            case E_NULL:
                  printf("receiving the NULL-event...");
                  break;
            case E_REDRAW:
                  printf("REDRAW event for window %d received\n",
                        block->array[0]);
                   redraw(block);
                  break;
            case E_SCROLL:
                  block->array[0]);
                   wscroll(block);
                  break;
            case E WOPEN:
                   printf("OPEN event for window %d received\n",
                         block->array[0]);
                   wopen(block);
                  break;
            case E_WCLOSE:
                  printf("CLOSE event for window %d received\n",
                        block->array[0]);
                   (*_caldrv)(*_hwindrv,CLOSE_WIN,block->array[0]);
                  break;
            case E_PNTOUT:
                  printf("pointer has left window %d\n",
                         block->array[0]);
                  break;
            case E_PNTIN:
                  printf("pointer has entered window %d\n",
                        block->array[0]);
                  break;
```

The example requires the existence of the functions 'redraw', 'wscroll' and 'wopen'.

The 'redraw' function redraws the window physically on the screen. The following example simply would draw a white box in the work-area, assuming we have a box-draw routine to draw a box with the arguments (x,y,nx,ny,colour).

After GETWIB was called, the window frame exists at the coordinates returned with the members 'sx' and 'sy' and the window data is copied to the global area (i.e. outside of page 1 and 2) pointed to by the member 'windat'. If the window is visible (it normally should... but the window driver is not fully optimized for that), it is copied from the hidden area to the screen.

The 'wscroll' routine simply has to determine whether the scrolling performed by the user (or requested by another task) is valid and confirm it by updating the scroll-offset of the window, poll again to receive the 'E_WOPEN' event and return to the normal task to poll (so the 'E_REDRAW' event can be sent). Since this routine also has to handle the open event, I use one common function for the scroll and for the open event. This function is called 'do wopen':

The 'OPEN_WIN' function changes the window in the memory list and sends the redraw event to the task (besides, it does more, like rearranging pane windows and all other windows which might be visible now, and sends a lot of open window requests, and... but this is not important for your task - just for the ones who want to know why to call this 'stupid' function first instead of just drawing all the window stuff right now on the screen;)

Now for the function:

Although it looks a bit dirty because there is no event-checking done after the poll routine, it is correct. Every window scroll request from WiOS is followed by a 'E_WOPEN' event. If a task sends a scroll request to another window, is must also send an open window request immediately afterwards.

The event-handling is almost complete now:

```
VOID wopen(block)
struct EBSTR *block;
{
          windat.handle=block->array[0];
          (*_caldrv)(*_hwindrv,GET_WIN_STATE,&windat); /* get win data */
          do_wopen(block);
}
```

This function gets a copy of the window data in the task's own window block since to be able to change the size without setting up the data of the whole window itself.

3.23 Security

Once a task has been created with 'MAKDRV' it should not be changed via a disk-editor. To protect tasks against changes (or viruses;-))), they contain a checksum which is checked by WiOS before executing the task. If the checksum is incorrect, the task will not be started. Yet there is no info-window for that. This checksumming is not a protection against hard-core crackers, nor is it a guarantee for the task being unaltered! It is some kind of 'guarantee' that if the task starts, it will be the same version the programmer compiled, and there will not be any bugs due to disk errors or bad modem lines.

3.24 Starting Tasks with the Alpha-Release

Whenever WiOS is loaded, two tasks are loaded and started. They must be named

TASK.TSK and TSK2.TSK

If these files are not present, WiOS will not start correctly. For Alpha-Testing, one file contains the normal code, and one file contains the code to send data or files, just as WiOS' file manager will do in future. Of course, if you don't need to send any data, you can also create a dummy task which does nothing or even start the same task twice by just copying TASK.TSK to TSK2.TSK;)

3.25 Problems, Bugs and Other Unwelcome Stuff

WiOS is not completely finished. It contains - I hope - the most necessary functions to create a wide variety of applications. This is important for debugging it. Besides the removing of bugs, there are other standard functions to be made in the next weeks and months. They are important for the public release, but not for testing the stability and suitability in making and running applications. The following problems are often only a

3.25.1 Known Problems

Scrollbars

Since slider in a scrollbars (both, in windows and menus) are dynamic (i.e. their size changes because it represents the proportion of the visible area to the virtual area), they have to be calculated in a special way. For this calculation, the double number of bits of the largest possible virtual size should be used (which is 16 bits). Therefore, long integer handling should be used. This is, of course, not a problem, but since ASCII-C does not support values >16 bit and I did not yet implement the routines of long integer handling, there might be wrong sliders with virtual sizes >8 bit (i.e. 256). This is already a problem when displaying full-screen windows (whose width are >256). But don't panic;) It will be fixed!!!

3.25.2 Planned Extensions

Sending data to other tasks

To let the user drag data, it will be possible in future to call a WiOS function which drags a user defined graphic icon over the desktop. This routine is fed with the same arguments as an 'E_EDRAG' event. The only difference is that the user may decide which task will receive the data. When the mouse button is released, the 'E_EDRAG' event is sent to the task belonging to the window where it was released and the sending task will return the handles of the task and the window. During this user drag, multitasking is paused.

This is especially for dragging files from the file manager to tasks, since it is desired that no task has its own disk menu. With WiOS, it is planned to make this as the standard way to open files.

Font Handling

WiOS' font manager has a lot of restrictions. The fonts can not be outlines, written in italics, their height is limited to 16 pixels, and if the VRAM is full, no more fonts can be added. Also the switching between fonts is quite slow (relatively...), because for every change, the statusdata is read from VRAM. This slows down some functions significantly, especially if several menus with different fonts are open.

If there is nobody who can support me with concrete ideas of how to make a senseful font handling, it is at least planned to speed up the procedure of changing fonts by placing the significant data which has to be read from VRAM normally into RAM, so only a pointer to the data has to be changed.

Right Mouse Button

When clicking at windows with the right mouse button, they will not be placed in front, but stay at their present level (in future). Also when clicking on a standard window slider (to adjust the scroll offset) with the right button, it will be possible to adjust it in both directions, horizontally <u>and</u> vertically.

4 FAQ - Frequently Asked Questions

Is WiOS fixed to 512x212 resolution with 32768 colors?

Yes, although no mode-checking is done. It is possible to change to another screen-mode. But before doing so, you should tell WiOS to save the environment (i.e. icons and fonts stored in the VRAM) and after you're finished, take action to redraw the complete desktop. WiOS does not prevent programs from doing 'dirty' graphic-tricks, but the results are unpredictable.

Is it possible to use more ram than available (i.e. virtual memory storage)? No, WiOS does NOT support scratch files on harddisk. If you want to swap unused memory to disk, feel free to do it, but the system is not supporting this automatically.

Will my slow mouse ever work correct in turbo mode?

Yes, but until now, this bug had a very low priority. It will be fixed (latest!) with the Zandvoort '97 version which comes with the file-manager.

Does WiOS support any printers?

That depends on the ideas of the programmers who think of being capable to write a printer driver. If somebody feels like doing it, I can support him with some dithering algorithms (sources!). Still there is no way to use a mix of bitmap and vector graphics in WiOS, but at the moment, printing images is not the most important task.

Well, it's up to you to fill up this page by asking me frequent questions (that does not mean that you ask me the same question frequently, but if you want an important question to be answered here, you could ask your friends to ask me the same question so I will be thinking that it is a frequent question:) just kidding...

5 Keyword Description

address block 16 byte area which stays untouched from segment switching. It can be used to store immediate data which has to be sent to other program parts. The GDA variable 'adrblk' points to its address.

Application = Task

GDA Global Data Area - the area where global variables and function addresses which should be able to be accessed by other programs are pointed to.

Task = Application

WIB Window Information Block - a data structure defined as WIBSTR window block The complete structure of a window which is defined as WINSTR. This block must be set up to use the window functions.

This is your page, write me anything you want to know and fill it in here :-)

6 Last But Not Least

If you encounter any problems, either with WiOS, the Application Creation Toolkit or with this manual, please send a message to:

Michael Stellmann

Geleenerstr. 14 71034 Böblingen Germany

E-Mail: mstellmann@msx.ch

Sunrise for MSX

Rob Hiep Postbus 61054 NL-2506 AB Den Haag Netherlands

E-Mail: rhiep@msx.ch

or

Sunrise for MSX

Peter Burkhard Moosmatten 1 CH-9244 Niederuzwil Switzerland

E-Mail: pburkhard@msx.ch

Our Homepage http://www.msx.ch

E-mails are the preferable way to communicate since it is faster and cheaper than snail-mail.

If something works wrong with WiOS, please write some words about the error and, if possible, send the source-code including the compiled version of your task or driver (i.e. your own '.C', '.H' and '.TSK' or '.DRV' files), so I can try to fix the problem as soon as possible.

If you find errors in the documentation, write where the error is or for which topic you want to receive more detailed information, so I can add or change this section.