libstack

COLLABORATORS			
	TITLE : libstack		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 4, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

libstack

1.1 libstack.guide

```
                  The current Amiga OS (V3.1) has a very limited stack handling
compared to most other OSs: Every process has it's own fixed sized
stack - and that's all about it. The usual default for this stack is
4k, but that's not enough for more complicated purposes (like for
example compilers). Setting a higher default is no real solution
because it costs a lot of (widely unused) memory and may be overrun,
too :-(.

   But fortunately you can get stack extension with a little help of
the compiler ;-).
```


1.2 libstack.guide/Overview

```
Overview
********
```

In this file you find everything necessary to use and understand stack extension. All parts of this document (as well as the stack extension code itself) are PD and therefore freely distributable. You use it at your own risk.

1.3 libstack.guide/Implementation

How stack extension works
****

For a reliable compiler supported stack extension mechanism you need to check the stack against overflow any time it grows by generating special checking code. This can happen at three different points in normal C code:

At function entry
Functions may need lots of variables, fixed sized arrays, preserve registers etc. The space needed for this is collected together by the compiler and allocated at function entry. Therefore you may have to swap to a new stackframe. Note that the function's arguments and local variables are accessed over the same mechanism - therefore it's necessary to copy the arguments to the new stackframe. Since you cannot know how many arguments you have to copy (C allows for a variable number of arguments) a fixed (user configurable, See
Advanced
.) amount of memory is copied.

When using alloca() or variable sized arrays
Both allocate memory off the stack.

When calling library functions
When calling library functions (glue code or link libraries) the arguments are put on the stack first, then the function gets called. Additionally a library function may use more or less stack on it's own. The easiest way to handle this is to guarantee a minimum of free stack (user configurable, See
Advanced
.).

The cleanup for the additional stack works equivalent.

Basic
Basic functions for stack handling

Additional
Additional wrappers needed for performance

1.4 libstack.guide/Basic

```
                Basic functions for stack extension
**********************************

   These functions are only callable from assembly code. They are the
core of the additional set.

   ___stkext

   Prepares a new stackframe with a minimum of d0 free bytes left.
Returns on the new stackframe. Preserves all registers

   ___stkext_f

   Similar to the previous one, but this function copies the arguments
of the caller and everything on top of them. d1 must be the offset (in
bytes) of the caller's returnaddress relative to sp.  This
returnaddress is replaced by a cleanup function's address (no explicit
cleanup necessary). Preserves all registers

   ___stkrst

   Similar to 'movel d0,sp'. Falls back to an old stackframe if
necessary.  Preserves all registers.

   ___stkovf

   Is the stack overflow handler (called if there is not enough memory
to extend the stack further and the program has to be aborted).  Does
not return.

   Additionally you find the current limit of the stack in

   ___stk_limit

   Note that this doesn't point to the lower stack boundary but leaves
some safety zone (See
                Advanced
                .).
```

1.5 libstack.guide/Additional

```
                Additional wrappers
******************

   These functions are all optimized for speed by checking if the
current stack is already large enough using it if possible and by not
preserving d0,d1,a0,a1. Future versions of the compiler (implementing
registerized parameters) might need a register preserving set. This is
the reason why the basic interface (See
                Basic
```

.) is documented.

Again all functions are only callable from assembly.

___link_a5_d0_f

May be used at the top of a function instead of link a5,d0. Checks if the current stackframe is large enough and allocates a new one (over ___stkext_f) if not.

___link_a5_0_f

Same as above but treats d0 as 0.

___sub_d0_sp_f

For usage at the top of a function, similar to sub d0,sp. Works over ___stkext_f.

___sub_0_sp_f

Same as above but treats d0 as 0.

___sub_d0_sp

Works like sub d0,sp but this time over ___stkext.

___move_d0_sp

___stkrst under a second name.

___unlk_a5_rts

Use this function instead of a simple unlk a5;rts if the current function calls ___sub_d0_sp and doesn't clean up later (a5 points to a different stackframe than sp). Works over ___stkrst. Preserves d0,d1.

___stkchk_d0

Just tests if there are d0 bytes space left on the stack - raising a stackoverflow if not.

___stkchk_0

Tests if there is any space left.

1.6 libstack.guide/Usage

How to use stack extension
****

First of all you need a gcc capable of generating the right assembly output. You should find some source patches for gcc 2.6.3 in the same

archive you got this file (you are reading) from. Use them to recompile your compiler.

Once you've got the new compiler you got two new target dependant switches:

* -mstackcheck Emits code that checks if there is enough stack left. The program exits if not.

* -mstackextend Tries to extend the stack before exiting (this may happen due to low or fragmented memory).

You can mix functions compiled with or without these switches without problems. Note that this library builds on libnix and needs the -noixemul switch.

Caution:

Do not use stack checking and/or extension switches when compiling hook or interrupt code. Both run in alien contexts with a different stack and all stack magic must fail. Also don't try to do some other stack magic if you want to use stack extension.

Also note that a program compiled with stack extension/checking may exit() at any function entry or when using alloca or variable sized arrays. Prepare your cleanup function accordingly (use atexit()).

If you like to write or call functions with more than 256 bytes of arguments (64 ints, longs or pointers) you should adjust the behaviour of the stack extension code (See
Advanced
.).

1.7 libstack.guide/Advanced

Stack extension fine tuning
****

To adjust the behaviour of the stack extension code to your personal needs you may set some of the following variables (or functions)

unsigned long __stk_minframe (default: 32768)

Minimum amount of memory to allocate for a new stackframe. Setting a higher value speeds the code up but costs more memory if it is unused.

unsigned long __stk_safezone (default: 2048)

Size of the safety zone (for __stk_limit). Set this to a higher value if you want to call functions with lots of arguments.

unsigned long __stk_argbytes (default: 256)

Number of bytes __stkext_f() copies as arguments. Set this to a higher value if your functions may have lots of arguments.

void _CXOVF(void)

Is a user replaceable stack overflow handler. The default one just pops up a requester, then exits. This function is not allowed to return.

1.8 libstack.guide/Costs

Overhead of stack extension
****

The additional code needed for stack extension (or checking) costs memory and CPU power. Here are some numbers to give you a feeling for it (times in 1/60s, sizes in bytes (remember that there are lies, damned lies and benchmarks ;-))):

Test	normal (big stack)	checking (big stack)	extending (big stack)	extending (small stack)
Simple recursive function runtime (function calling overhead)	152	221	225	226
Variable sized array runtime	52	136	398	468
alloca runtime	31	118	118	118
Own code size	1040	1160	1140	
Library code size	0	184	788	