S390 Debug Feature

files: arch/s390/kernel/debug.c
arch/s390/include/asm/debug.h

Description:

The goal of this feature is to provide a kernel debug logging API
where log records can be stored efficiently in memory, where each component
(e.g. device drivers) can have one separate debug log.
One purpose of this is to inspect the debug logs after a production system crash
in order to analyze the reason for the crash.
If the system still runs but only a subcomponent which uses dbf fails,
it is possible to look at the debug logs on a live system via the Linux
debugfs filesystem.
The debug feature may also very useful for kernel and driver development.

Design:

Kernel components (e.g. device drivers) can register themselves at the debug feature with the function call debug_register(). This function initializes a debug log for the caller. For each debug log exists a number of debug areas where exactly one is active at one time. Each debug area consists of contiguous pages in memory. In the debug areas there are stored debug entries (log records) which are written by event- and exception-calls.

An event-call writes the specified debug entry to the active debug area and updates the log pointer for the active area. If the end of the active debug area is reached, a wrap around is done (ring buffer) and the next debug entry will be written at the beginning of the active debug area.

An exception-call writes the specified debug entry to the log and switches to the next debug area. This is done in order to be sure that the records which describe the origin of the exception are not overwritten when a wrap around for the current area occurs.

The debug areas themselves are also ordered in form of a ring buffer. When an exception is thrown in the last debug area, the following debug entries are then written again in the very first area.

There are three versions for the event- and exception-calls: One for logging raw data, one for text and one for numbers.

Each debug entry contains the following data:

- Timestamp
- Cpu-Number of calling task
- Level of debug entry (0...6)
- Return Address to caller
- Flag, if entry is an exception or not

The debug logs can be inspected in a live system through entries in the debugfs-filesystem. Under the toplevel directory "s390dbf" there is a directory for each registered component, which is named like the

corresponding component. The debugfs normally should be mounted to
/sys/kernel/debug therefore the debug feature can be accessed under
/sys/kernel/debug/s390dbf.

The content of the directories are files which represent different views
to the debug log. Each component can decide which views should be
used through registering them with the function debug_register_view().
Predefined views for hex/ascii, sprintf and raw binary data are provided.
It is also possible to define other views. The content of
a view can be inspected simply by reading the corresponding debugfs file.

All debug logs have an actual debug level (range from 0 to 6).
The default level is 3. Event and Exception functions have a 'level'
parameter. Only debug entries with a level that is lower or equal
than the actual level are written to the log. This means, when
writing events, high priority log entries should have a low level
value whereas low priority entries should have a high one.
The actual debug level can be changed with the help of the debugfs-filesystem
through writing a number string "x" to the 'level' debugfs file which is
provided for every debug log. Debugging can be switched off completely
by using "-" on the 'level' debugfs file.

Example:

> echo "-" > /sys/kernel/debug/s390dbf/dasd/level

It is also possible to deactivate the debug feature globally for every
debug log. You can change the behavior using 2 sysctl parameters in
/proc/sys/s390dbf:
There are currently 2 possible triggers, which stop the debug feature
globally. The first possibility is to use the "debug_active" sysctl. If
set to 1 the debug feature is running. If "debug_active" is set to 0 the
debug feature is turned off.
The second trigger which stops the debug feature is a kernel oops.
That prevents the debug feature from overwriting debug information that
happened before the oops. After an oops you can reactivate the debug feature
by piping 1 to /proc/sys/s390dbf/debug_active. Nevertheless, its not
suggested to use an oopsed kernel in a production environment.
If you want to disallow the deactivation of the debug feature, you can use
the "debug_stoppable" sysctl. If you set "debug_stoppable" to 0 the debug
feature cannot be stopped. If the debug feature is already stopped, it
will stay deactivated.

Kernel Interfaces:

```
debug_info_t *debug_register(char *name, int pages, int nr_areas,
                             int buf_size);
```

Parameter: name: Name of debug log (e.g. used for debugfs entry)
            pages: number of pages, which will be allocated per area
            nr_areas: number of debug areas
            buf_size: size of data area in each debug entry

Return Value: Handle for generated debug area

```
              NULL if register failed

Description:  Allocates memory for a debug log
              Must not be called within an interrupt handler

------------------------------------------------------------------------
debug_info_t *debug_register_mode(char *name, int pages, int nr_areas,
                                  int buf_size, mode_t mode, uid_t uid,
                                  gid_t gid);

Parameter:    name:        Name of debug log (e.g. used for debugfs entry)
              pages:       Number of pages, which will be allocated per area
              nr_areas:    Number of debug areas
              buf_size:    Size of data area in each debug entry
              mode:        File mode for debugfs files. E.g. S_IRWXUGO
              uid:         User ID for debugfs files. Currently only 0 is
                           supported.
              gid:         Group ID for debugfs files. Currently only 0 is
                           supported.

Return Value: Handle for generated debug area
              NULL if register failed

Description:  Allocates memory for a debug log
              Must not be called within an interrupt handler

------------------------------------------------------------------------
void debug_unregister (debug_info_t * id);

Parameter:     id:   handle for debug log

Return Value:  none

Description:   frees memory for a debug log
               Must not be called within an interrupt handler

------------------------------------------------------------------------
void debug_set_level (debug_info_t * id, int new_level);

Parameter:     id:        handle for debug log
               new_level: new debug level

Return Value:  none

Description:   Sets new actual debug level if new_level is valid.

------------------------------------------------------------------------
void debug_stop_all(void);

Parameter:     none

Return Value:  none

Description:   stops the debug feature if stopping is allowed. Currently
               used in case of a kernel oops.
```

```
--------------------------------------------------------------------------
debug_entry_t* debug_event (debug_info_t* id, int level, void* data,
                            int length);

Parameter:     id:     handle for debug log
               level:  debug level
               data:   pointer to data for debug entry
               length: length of data in bytes

Return Value:  Address of written debug entry

Description:   writes debug entry to active debug area (if level <= actual
               debug level)

--------------------------------------------------------------------------
debug_entry_t* debug_int_event (debug_info_t * id, int level,
                                unsigned int data);
debug_entry_t* debug_long_event(debug_info_t * id, int level,
                                unsigned long data);

Parameter:     id:     handle for debug log
               level:  debug level
               data:   integer value for debug entry

Return Value:  Address of written debug entry

Description:   writes debug entry to active debug area (if level <= actual
               debug level)

--------------------------------------------------------------------------
debug_entry_t* debug_text_event (debug_info_t * id, int level,
                                 const char* data);

Parameter:     id:     handle for debug log
               level:  debug level
               data:   string for debug entry

Return Value:  Address of written debug entry

Description:   writes debug entry in ascii format to active debug area
               (if level <= actual debug level)

--------------------------------------------------------------------------
debug_entry_t* debug_sprintf_event (debug_info_t * id, int level,
                                    char* string,...);

Parameter:     id:    handle for debug log
               level: debug level
               string: format string for debug entry
               ...: varargs used as in sprintf()

Return Value:  Address of written debug entry

Description:   writes debug entry with format string and varargs (longs) to
               active debug area (if level $<=$ actual debug level).
               floats and long long datatypes cannot be used as varargs.
```

```
debug_entry_t* debug_exception (debug_info_t* id, int level, void* data,
                                int length);

Parameter:     id:     handle for debug log
               level:  debug level
               data:   pointer to data for debug entry
               length: length of data in bytes

Return Value:  Address of written debug entry

Description:   writes debug entry to active debug area (if level <= actual
               debug level) and switches to next debug area
```

```
debug_entry_t* debug_int_exception (debug_info_t * id, int level,
                                    unsigned int data);
debug_entry_t* debug_long_exception(debug_info_t * id, int level,
                                    unsigned long data);

Parameter:     id:     handle for debug log
               level:  debug level
               data:   integer value for debug entry

Return Value:  Address of written debug entry

Description:   writes debug entry to active debug area (if level <= actual
               debug level) and switches to next debug area
```

```
debug_entry_t* debug_text_exception (debug_info_t * id, int level,
                                     const char* data);

Parameter:     id:     handle for debug log
               level:  debug level
               data:   string for debug entry

Return Value:  Address of written debug entry

Description:   writes debug entry in ascii format to active debug area
               (if level <= actual debug level) and switches to next debug
               area
```

```
debug_entry_t* debug_sprintf_exception (debug_info_t * id, int level,
                                        char* string,...);

Parameter:     id:    handle for debug log
               level: debug level
               string: format string for debug entry
               ...: varargs used as in sprintf()

Return Value:  Address of written debug entry
```

Description: writes debug entry with format string and varargs (longs) to
              active debug area (if level $<=$ actual debug level) and
              switches to next debug area.
              floats and long long datatypes cannot be used as varargs.

---

int debug_register_view (debug_info_t * id, struct debug_view *view);

Parameter: id: handle for debug log
              view: pointer to debug view struct

Return Value: 0 : ok
              < 0: Error

Description: registers new debug view and creates debugfs dir entry

---

int debug_unregister_view (debug_info_t * id, struct debug_view *view);

Parameter: id: handle for debug log
              view: pointer to debug view struct

Return Value: 0 : ok
              < 0: Error

Description: unregisters debug view and removes debugfs dir entry

Predefined views:
---

extern struct debug_view debug_hex_ascii_view;
extern struct debug_view debug_raw_view;
extern struct debug_view debug_sprintf_view;

Examples

```
/*
 * hex_ascii- + raw-view Example
 */

#include <linux/init.h>
#include <asm/debug.h>

static debug_info_t* debug_info;

static int init(void)
{
    /* register 4 debug areas with one page each and 4 byte data field */

    debug_info = debug_register ("test", 1, 4, 4 );
    debug_register_view(debug_info,&debug_hex_ascii_view);
    debug_register_view(debug_info,&debug_raw_view);
```

```
    debug_text_event(debug_info, 4 , "one ");
    debug_int_exception(debug_info, 4, 4711);
    debug_event(debug_info, 3, &debug_info, 4);

    return 0;
}

static void cleanup(void)
{
    debug_unregister (debug_info);
}

module_init(init);
module_exit(cleanup);
```

```
/*
 * sprintf-view Example
 */

#include <linux/init.h>
#include <asm/debug.h>

static debug_info_t* debug_info;

static int init(void)
{
    /* register 4 debug areas with one page each and data field for */
    /* format string pointer + 2 varargs (= 3 * sizeof(long))       */

    debug_info = debug_register ("test", 1, 4, sizeof(long) * 3);
    debug_register_view(debug_info,&debug_sprintf_view);

    debug_sprintf_event(debug_info, 2 , "first event in
%s:%i\n",__FILE__,__LINE__);
    debug_sprintf_exception(debug_info, 1, "pointer to debug info:
%p\n",&debug_info);

    return 0;
}

static void cleanup(void)
{
    debug_unregister (debug_info);
}

module_init(init);
module_exit(cleanup);
```

Debugfs Interface

Views to the debug logs can be investigated through reading the corresponding

debugfs-files:

Example:

```
> ls /sys/kernel/debug/s390dbf/dasd
flush  hex_ascii  level pages raw
> cat /sys/kernel/debug/s390dbf/dasd/hex_ascii | sort +1
00 00974733272:680099 2 - 02 0006ad7e  07 ea 4a 90 | ....
00 00974733272:682210 2 - 02 0006ade6  46 52 45 45 | FREE
00 00974733272:682213 2 - 02 0006adf6  07 ea 4a 90 | ....
00 00974733272:682281 1 * 02 0006ab08  41 4c 4c 43 | EXCP
01 00974733272:682284 2 - 02 0006ab16  45 43 4b 44 | ECKD
01 00974733272:682287 2 - 02 0006ab28  00 00 00 04 | ....
01 00974733272:682289 2 - 02 0006ab3e  00 00 00 20 | ...
01 00974733272:682297 2 - 02 0006ad7e  07 ea 4a 90 | ....
01 00974733272:684384 2 - 00 0006ade6  46 52 45 45 | FREE
01 00974733272:684388 2 - 00 0006adf6  07 ea 4a 90 | ....
```

See section about predefined views for explanation of the above output!

Changing the debug level

Example:

```
> cat /sys/kernel/debug/s390dbf/dasd/level
3
> echo "5" > /sys/kernel/debug/s390dbf/dasd/level
> cat /sys/kernel/debug/s390dbf/dasd/level
5
```

Flushing debug areas

Debug areas can be flushed with piping the number of the desired
area (0...n) to the debugfs file "flush". When using "-" all debug areas
are flushed.

Examples:

1. Flush debug area 0:
```
> echo "0" > /sys/kernel/debug/s390dbf/dasd/flush
```

2. Flush all debug areas:
```
> echo "-" > /sys/kernel/debug/s390dbf/dasd/flush
```

Changing the size of debug areas

It is possible the change the size of debug areas through piping
the number of pages to the debugfs file "pages". The resize request will
also flush the debug areas.

Example:

Define 4 pages for the debug areas of debug feature "dasd":
```
> echo "4" > /sys/kernel/debug/s390dbf/dasd/pages
```

Stooping the debug feature

Example:

1. Check if stopping is allowed
> cat /proc/sys/s390dbf/debug_stoppable
2. Stop debug feature
> echo 0 > /proc/sys/s390dbf/debug_active

lcrash Interface

It is planned that the dump analysis tool lcrash gets an additional command 's390dbf' to display all the debug logs. With this tool it will be possible to investigate the debug logs on a live system and with a memory dump after a system crash.

Investigating raw memory

One last possibility to investigate the debug logs at a live system and after a system crash is to look at the raw memory under VM or at the Service Element.
It is possible to find the anker of the debug-logs through the 'debug_area_first' symbol in the System map. Then one has to follow the correct pointers of the data-structures defined in debug.h and find the debug-areas in memory.
Normally modules which use the debug feature will also have a global variable with the pointer to the debug-logs. Following this pointer it will also be possible to find the debug logs in memory.

For this method it is recommended to use '16 * x + 4' byte (x = 0..n) for the length of the data field in debug_register() in order to see the debug entries well formatted.

Predefined Views

There are three predefined views: hex_ascii, raw and sprintf.
The hex_ascii view shows the data field in hex and ascii representation (e.g. '45 43 4b 44 | ECKD').
The raw view returns a bytestream as the debug areas are stored in memory.

The sprintf view formats the debug entries in the same way as the sprintf function would do. The sprintf event/exception functions write to the debug entry a pointer to the format string (size = sizeof(long)) and for each vararg a long value. So e.g. for a debug entry with a format string plus two varargs one would need to allocate a (3 * sizeof(long)) byte data area in the debug_register() function.

IMPORTANT: Using "%s" in sprintf event functions is dangerous. You can only use "%s" in the sprintf event functions, if the memory for the passed string is available as long as the debug feature exists. The reason behind this is that due to performance considerations only a pointer to the string is stored in the debug feature. If you log a string that is freed afterwards, you will get

an OOPS when inspecting the debug feature, because then the debug feature will access the already freed memory.

NOTE: If using the sprintf view do NOT use other event/exception functions than the sprintf-event and -exception functions.

The format of the hex_ascii and sprintf view is as follows:
- Number of area
- Timestamp (formatted as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970)
- level of debug entry
- Exception flag (* = Exception)
- Cpu-Number of calling task
- Return Address to caller
- data field

The format of the raw view is:
- Header as described in debug.h
- datafield

A typical line of the hex_ascii view will look like the following (first line is only for explanation and will not be displayed when 'cating' the view):

```
area  time           level exception cpu caller    data (hex + ascii)
-----------------------------------------------------------------------
00    00964419409:440690 1 -         00  88023fe
```

Defining views

Views are specified with the 'debug_view' structure. There are defined callback functions which are used for reading and writing the debugfs files:

```
struct debug_view {
        char name[DEBUG_MAX_PROCF_LEN];
        debug_prolog_proc_t* prolog_proc;
        debug_header_proc_t* header_proc;
        debug_format_proc_t* format_proc;
        debug_input_proc_t*  input_proc;
        void*                private_data;
};
```

where

```
typedef int (debug_header_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
                                   int area,
                                   debug_entry_t* entry,
                                   char* out_buf);

typedef int (debug_format_proc_t) (debug_info_t* id,
                                   struct debug_view* view, char* out_buf,
                                   const char* in_buf);
typedef int (debug_prolog_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
```

```
                                    char* out_buf);
typedef int (debug_input_proc_t) (debug_info_t* id,
                                  struct debug_view* view,
                                  struct file* file, const char* user_buf,
                                  size_t in_buf_size, loff_t* offset);
```

The "private_data" member can be used as pointer to view specific data.
It is not used by the debug feature itself.

The output when reading a debugfs file is structured like this:

"prolog_proc output"

"header_proc output 1" "format_proc output 1"
"header_proc output 2" "format_proc output 2"
"header_proc output 3" "format_proc output 3"
...

When a view is read from the debugfs, the Debug Feature calls the
'prolog_proc' once for writing the prolog.
Then 'header_proc' and 'format_proc' are called for each
existing debug entry.

The input_proc can be used to implement functionality when it is written to
the view (e.g. like with 'echo "0" > /sys/kernel/debug/s390dbf/dasd/level).

For header_proc there can be used the default function
debug_dflt_header_fn() which is defined in debug.h.
and which produces the same header output as the predefined views.
E.g:
00 00964419409:440761 2 - 00 88023ec

In order to see how to use the callback functions check the implementation
of the default views!

Example

```
#include <asm/debug.h>

#define UNKNOWNSTR "data: %08x"

const char* messages[] =
{"This error.....\n",
 "That error.....\n",
 "Problem.....\n",
 "Something went wrong.\n",
 "Everything ok.....\n",
 NULL
};

static int debug_test_format_fn(
   debug_info_t * id, struct debug_view *view,
   char *out_buf, const char *in_buf
)
{
```

```
  int i, rc = 0;

  if(id->buf_size >= 4) {
     int msg_nr = *((int*)in_buf);
     if(msg_nr < sizeof(messages)/sizeof(char*) - 1)
        rc += sprintf(out_buf, "%s", messages[msg_nr]);
     else
        rc += sprintf(out_buf, UNKNOWNSTR, msg_nr);
  }
 out:
   return rc;
}

struct debug_view debug_test_view = {
  "myview",                   /* name of view */
  NULL,                       /* no prolog */
  &debug_dflt_header_fn,      /* default header for each entry */
  &debug_test_format_fn,      /* our own format function */
  NULL,                       /* no input function */
  NULL                        /* no private data */
};
```

=====
test:
=====

```
debug_info_t *debug_info;
...
debug_info = debug_register ("test", 0, 4, 4 ));
debug_register_view(debug_info, &debug_test_view);
for(i = 0; i < 10; i ++) debug_int_event(debug_info, 1, i);

> cat /sys/kernel/debug/s390dbf/test/myview
00 00964419734:611402 1 - 00 88042ca   This error.....
00 00964419734:611405 1 - 00 88042ca   That error.....
00 00964419734:611408 1 - 00 88042ca   Problem.....
00 00964419734:611411 1 - 00 88042ca   Something went wrong.
00 00964419734:611414 1 - 00 88042ca   Everything ok........
00 00964419734:611417 1 - 00 88042ca   data: 00000005
00 00964419734:611419 1 - 00 88042ca   data: 00000006
00 00964419734:611422 1 - 00 88042ca   data: 00000007
00 00964419734:611425 1 - 00 88042ca   data: 00000008
00 00964419734:611428 1 - 00 88042ca   data: 00000009
```