Zephyr™ Project
Developer Summit
June 8-10, 2021 • @ZephyrIoT

Logging subsystem overview

KRZYSZTOF CHRUŚCIŃSKI

Agenda

- Variety of requirements
- Deferred mode concept
- Zephyr logging subsystem history
- Version 1
  - Features
  - Weaknesses
- Version 2
  - Challenges
  - Solutions
  - Outlook

Requirements - output

- Printf-like
  - LOG_ERR("Error: %d", err)
- Logging data
  - LOG_HEXDUMP(buf, len, "packet:")
- Readable
- Prefixed (level, source, function name)
- Timestamp
  - High precision
  - coarse, non-wrapping
  - Custom
- Formatted output vs compressed
  - Ease of use vs resource optimized

Requirements - sink

- UART console
  - Single direction vs interactive
- Filesystem
- Remote (net, bluetooth)
- Simultaneous multiple transports (*backends*)

Requirements - filtering

- Built-time
  - Global level (e.g. only warnings and errors)
  - Per source (e.g. debugs in i2c, warnings in spi)
  - Per instance (e.g. debugs on uart0, errors on uart1)
- Run-time
  - Per source and instance
  - Per backend

Requirements - RTOS

- Closest to non-intrusive
  - String formatting
  - Backends handling
- Log from any context
  - Even highest priority interrupts
  - Synchronous logging takes time, impossible for certain backends
- Critical last logs
- In some configurations we don’t care (emulation, unit testing)

Requirements - RTOS

- ROM/RAM requirements
  - Speed
  - Size
  - Both
  - None

Deferred logging

- Heavy processing deferred to the known context
- Buffering
  - full policy: saturate vs discard the oldest
- Challenges
  - Deceiving when single stepping
  - Flush before reset (fault, watchdog timeout)
  - Panic context

Panic mode

- Most important data!
- Exception context
- No more scheduler and interrupts
- Backends must switch to synchronous, interrupt-less mode (if possible)

Modes of operation

- Deferred
  - Any backend
  - Any context
  - Close to non-intrusive
  - Delayed output
  - Build-time, run-time filtering
  - Rich formatting, timestamping
- Immediate
  - Intrusive
  - Limited backends supported
  - Immediate output
  - Filtering and formatting
- Minimal
  - Redirection to printk
  - No run time filtering, formatting, timestamping
  - Low footprint

Deferred vs immediate

- „Test string with 5 parameters %d %d %d %d %d”
  - Formatting **80us** (Cortex-M4 @64MHz)
    - 300us if %lld instead of %d
  - Transfer over UART @115.2k **4ms**
- Deferring
  - Store log message with string pointer, arguments, timestamp, etc. in few microseconds
  - Heavy weight processing in low priority thread (default or user context)

Deferred mode

- Log call context
  - (runtime filtering) Check if any backend enables given (source,level)
  - Allocate and enqueue message
- In the known context (idle) for each backend:
  - (runtime filtering) check if backend enables given source,level
  - pass message to a log backend.

Deferred mode – corner cases

- „%s” with transient strings
- Storing arguments for formatting
  - Arguments dump vs va_list
- Offline processing
- Handling full buffer
- 2 processing contexts
  - Default low priority context
  - Panic interrupting default processing context

```
void foo(void)
{
    char name[] = "name"; /* on stack */
    LOG_INF("Name: %s", name);
}
```

Logging subsystem history

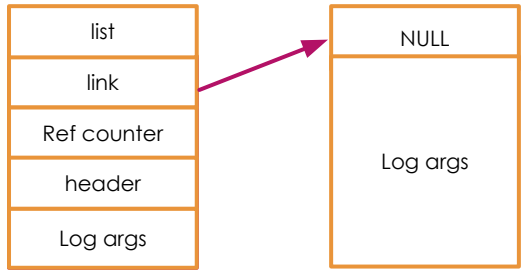
- Introduced in June 2018 (version 1)
- Based on logging from Nordic Semiconductor nRF5 SDK
  - Primary bare metal, 32 bit ARM Cortex M only
  - Limited need for logging floats and 64 bit
  - Transient strings logging may be avoided, or limits accepted
  - Asynchronous backend processing to not block the main loop
- While Zephyr scope continues to grow...
- Overhaul merged in April 2021 (version 2)
  - Keeping version 1 as the default one for now

Log message (v1)

- Cast all string arguments to the machine word
  - No support for floating point or long long
- Callculate size needed and allocate message from the pool
- Add timestamp, level and source ID
- Add message to the list

Log message buffer management (v1)

- Pool of fixed size slabs
- Variable length message consists of linked slabs
  - Slab size aligned to fit standard message (up to 2 arguments)
- Head slab contains reference counter
  - Backend gets and puts back the message
  - Last one frees the message
  - Asynchronous backends (no circular buffer)
  - No control when backend puts back the message
- List of buffered messages



Log message buffer management (v1)

- Pros
  - Fast allocation and enqueueing (single chunk)
  - No 2 contexts problem
  - Shared memory between backends
- Cons
  - Fragmented
  - Hard to self-contain (%s)
  - Costly allocation when saturated
  - No direct string formatting

%s arguments

- String arguments must be wrapped with *log_strdup()*
- *log_strdup()* copies to a buffer from dedicated memory pool
  - Size of the pool must be tuned to the application
- When message is freed, additional scan is performed to find duplicates and free them

```
void foo(void)
{
    char name[] = "name"; /* on stack */
    LOG_INF("Name: %s", log_strdup(name));
}
```

String formatting (v1)

- Convert fragmented message to va_list
- Complex to handle other types than machine word
- Code size (0.7k) and stack usage bloats

```
switch (nargs) {
    case 0:
        print(str); break;
        break;
    case 1:
        print(str, args[0]); break;
    ...
    case 5:
        print(str, args[0], args[1],
              args[2], args[3], args[4]);
        break;
    ...
    case 15:
```

Deferred logging v1 limitations

- *log_strdup()* PITA
- Limited types support
- Fragmented, distributed messages
  - Complex implementation
  - Complicates string formatting, offline processing
  - multi-domain support

Zephyr™ Project
Developer Summit

Time for the ovehaul...

Overhaul goals

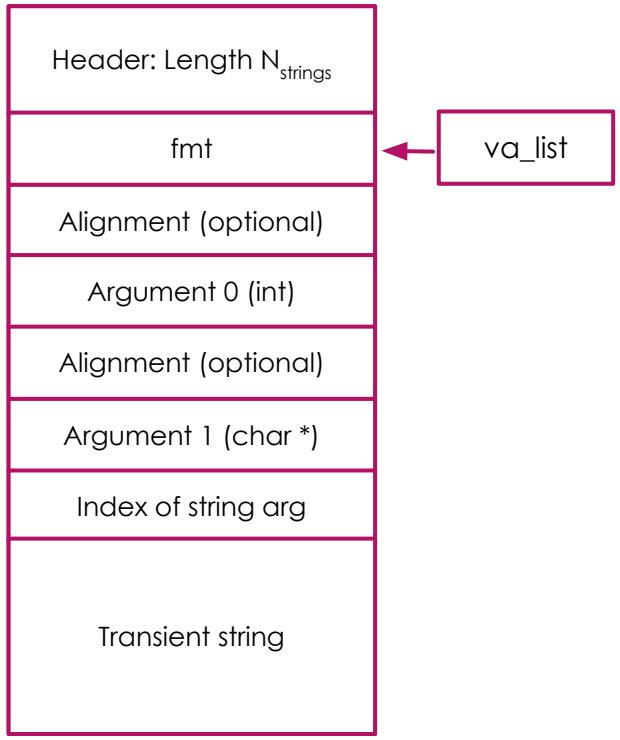
- Get rid of log_strdup()
- Support all format specifiers
- Self-contained message
- Keep v1 performance
- Others:
  - 64 bit timestamp
  - Simplify implementation
    - Unify standard and hexdump messages

Self-contained string package

- **cbprintf** – Zephyr string formatting library (since 11.2020)
- Way of deferring string formatting contrary to va_list or __VA_ARGS__
- Package
  - *int create_package(uint8_t *output, char *fmt, va_list ap)*
  - *output* is a buffer with opaque content. Can be moved and copied.
  - *snprintf_from_package(strbuf, sizeof(strbuf), output)*
- Self-contained
  - *fmt* pointer
  - Arguments
  - Any transient strings copied into the package body

String package

- „Test %d %s”
- Argument alignment
- Package body as va_list stack frame
- When copying alignment must be maintained
- Processing
  - Update pointers to in-package strings
  - Call standard string formatting



Building string package

- Walk over format string
- Copy arguments
- Maintain proper alignment
- If string is found, check if read only (rodata linker section)
- Multiple times faster than string formatting but still relatively slow (~20-30us)
- API
  - int cbprintf_package(package, len, fmt, …)
  - cbpprintf(cbprintf_cb out, void *ctx, void *packaged)

A missing piece

- Build string package but keep v1 performance
- Hypothesis: String package can be built by assigning arguments to an array but
  - How handle special cases (e.g., float to double promotion)?
  - How to handle %s?
  - How to add proper alignment?
  - How to detect when it cannot be done?

_Generic

- Introduced in C11
- C response to C++ overloading and templates ;)
- Supported by GCC, Clang

```
#define IS_INT(x) _Generic(x,\
                  int: printk("yes"), \
                  default: printk("no") \
                  )
```

_Generic new opportunities

- Static (compile time) string packaging
- Handle corner cases
- Compile-time detection when runtime packaging is necessary

Detect string pointers

- Detect char pointer

```
#define IS_CHAR_PTR(x) \
  _Generic(x, char *: 1, default: 0)
```

- Run it for each argument and sum the results

```
FOR_EACH(IS_CHAR_PTR, (+), __VA_ARGS__)
```

- Will return positive value if any char * in the argument list

```
MUST_RUNTIME_PACKAGE("%p", (char *)p)
MUST_RUNTIME_PACKAGE("%s", "read only")
```

Static string packaging

- Detects if static packaging can be applied
- Calculates space needed for a package
- Write package
- Fallback to runtime package if _Generic not supported
- API:
  - CBPRINTF_MUST_RUNTIME_PACKAGE(skip, …)
  - CBPRINTF_STATIC_PACKAGE(packaged, inlen, outlen, align_offset, …)

Static string packaging use


```
#define STATIC_PACKAGE(...)                                                \
int len;                                                                   \
if (CBPRINTF_MUST_RUNTIME_PACKAGE(0, __VA_ARGS__)) {                       \
    len = cbprintf_package(NULL, 0, __VA_ARGS__);                          \
} else {                                                                   \
    CBPRINTF_STATIC_PACKAGE(NULL, 0, len, 0, __VA_ARGS__);                 \
}                                                                          \
uint8_t buf[len];                                                          \
if (CBPRINTF_MUST_RUNTIME_PACKAGE(0, __VA_ARGS__)) {                       \
    len = cbprintf_package(buf, len, __VA_ARGS__);                         \
} else {                                                                   \
    CBPRINTF_STATIC_PACKAGE(buf, len, len, 0, __VA_ARGS__);                \
}                                                                          \
```

Preprocessor + Compiler

- Each static packaging resolves to hundreds of line of C code
- Compiler is smart and resolves it to few simple memory assignments or call to runtime packaging

```
int y = 5;
int x = 5 - y;
static int z = 100;
if (x > 0) {
    printk("removed %d", z);
}
```

Compilation


```c
void func(int i)
{
    STATIC_PACKAGE("test string: %d", i);
    memcpy(out, buf, len);
}
```

```
0000040E    LDR             R3, =0x00004A10
00000412    STRD            R3, R0, [SP, #8]
00000418    MOVS            R3, #3
0000041C    STR             R3, [SP, #4]
```

```c
void func2(int i, uint8_t j, long long ll)
{
    STATIC_PACKAGE("test string: %d %hhu %lld", i, j, ll);
    memcpy(out, buf, len);
}
```

```
0000040E    LDR             R4, =0x00004A20
00000410    SUB             SP, SP, #24
00000412    STRD            R2, R3, [SP, #16]
00000416    STRD            R4, R0, [SP, #4]
0000041C    MOVS            R3, #6
00000424    STR             R3, [SP, #0]
0000041A    STR             R1, [SP, #12]
0000041E    MOVS            R2, #24
00000420    MOV             R1, SP
00000422    LDR             R0, =out
00000426    BL              memcpy
```

Log message v2

- Use *cbprintf* packaging
- At compile time determines message size (except hexdump case)
- Built log message as few memory assignments
- Stores the message
- Message has variable size and must not be fragmented

Log message v2

- Generic message (standard/hexdump)
- +12 bytes (32 bit arch)

Domain ID (3 bits)	**Descriptor**
Level (3 bits)	
String package length (10 bits)	
Data length (12 bits)	
Timestamp (32 or 64 bits)	
Source (pointer to static or dynamic struct)	
Package (0 – n bytes)	
Data (0 – n bytes)	

Log backend API change

- v1
  - Put(backend, msg)
  - Immediate
    - Put_sync_string()
    - Put_sync_hexdump()
- Separate handling of hexdump and standard messages

- v2
  - Process(backend, msg2)
- Single message type

Circular buffer vs fragmented chunks

- Variable length, continuous space preferable over fragmented chunks
  - Easier to build, move, process
- Thread can block
  - Asynchronous backends not that beneficial
- Circular buffer challenges
  - Efficiency – hard to bit memslab
    - No memcpy: Allocate, Commit, Claim, Free
  - Wrapping – padding needed
  - Multiple producer single consumer
  - ...unless panic
  - No space policy – saturate vs drop the oldest

Multi producer single consumer packet buffer

- No memcpy
  - Allocate, Commit
  - Claim, Free
- Memory optimized: 2 bit header
- User handlers
  - Get packet length
  - Notify about drop

valid	busy	
0	0	Free space
1	0	User packet
1	1	Claimed user packet
0	1	Internal padding

Summary

- Improvements
  - Any variable type supported
  - Soon to forget about log_strdup()
  - Subsystem code size 6.2k down to 3.5k
  - ~20% better buffer utilization (continuous vs fragmented)
  - Logging from user space 5x faster
- Logging macros
  - 2x bigger (average message size 15 to 32 bytes)
  - CONFIG_LOG_SPEED=y bigger (47 bytes) but close to v1 performance

Outlook

- Self-contained messages opportunities
- Dictionary based logging (#30539)
  - Message send to host for offline processing
  - Log strings might be stripped from the binary (TBD)
- Flash/filesystem backend can be optimized
  - Currently stores formatted strings
- Multidomain logging
  - Multicore system
  - One master core with "real" backend
  - Other cores with ipc backends
  - TODO: String packaging alignment

Zephyr™ Project
Developer Summit

Thank you

Zephyr™ Project

Developer Summit

June 8-10, 2021 ▪ @ZephyrIoT