Technical Report: Driver Generation for IoT Nodes with Optimization of the Hardware/Software Interface

Stahl, Rafael Mueller-Gritschneder, Daniel
Schlichtmann, Ulf

Chair of Electronic Design Automation
Technical University of Munich
October 2019

1 Introduction

The number of tiny connected devices, so called IoT nodes, is growing steadily within the Internet of things (IoT). Increasingly, microcontroller units (MCU) are used as basis for IoT nodes, because they can be programmed more flexibly and allow easy distribution of software updates compared to fixed hardware. For industry, it is challenging to provide such devices at low cost. A significant part of that cost is spent on software development that needs to consider limited memory as well as computational resources. These MCUs usually come with a wide range of peripherals that allow the IoT node to interact with connected sensors and actuators. Such peripherals include General Purpose Input/Output Pins (GPIO), serial interfaces such as SPI or I^2C as well as ADCs and DACs. To enable software access, these peripherals provide a low-level hardware interface that is usually implemented through memory-mapped registers. For easier development, an MCU is typically provided to customers along with device driver code that abstracts this low-level interface of the peripheral to a more intuitive user-oriented one. Application developers target to implement smart functionality in software with highly limited resources in terms of design effort, available on-chip memory and computation power. This functionality should be able to make use of most of these resources while keeping the resources used by the drivers as low as possible. Yet, this driver code can make up a significant portion of an IoT node's limited memory. For example the RISC-V PULPino MCU has 32 KiB instruction memory of which all driver code already occupies 6% [12]. In a minimal application it occupies the majority of the entire code size as shown in Fig. 1.

To tackle software development cost, the development flow has to be simplified and parts have to be automated where possible. Instead of being able to

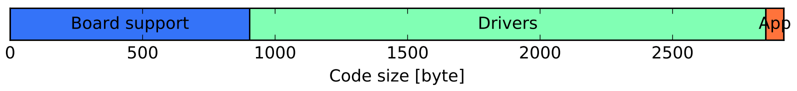

Figure 1: PULPino code size distribution

focus on driver behavior, developers right now have to simultaneously consider driver behavior, the register layout of the peripheral, performance and memory footprint. Additionally, certain choices of register layouts in the low-level interfaces may pollute the source code with macros and bit manipulation operations, decreasing readability and maintainability. This can be alleviated by using a Hardware Abstraction Layer (HAL) to hide register-layout-specific code like shifting and masking operations. Instead of accessing an address and performing some unrelated arithmetic, a device parameter or *bit field* is accessed by their name through a HAL function. Yet, this HAL approach can lead to inferior performance and memory footprint if the chosen register layout and HAL functions are not aligned with the driver behavior.

This report describes (1) a new driver development flow that enables the definition of a flexible HW/SW interface without fixed register layout. Driver behavior is described by an easy-to-adopt C-like domain-specific language (DSL). With this DSL, a developer can focus on describing behavior using special features such as bit field arrays and hierarchy, while not having to care how these are mapped into registers. In order to additionally exploit the possibilities of modifying the register layout to reduce performance and memory footprint, the report describes (2) a heuristic to find an optimized register layout and (3) a code analysis and generation method that exploits the optimized layout. Here, especially, software accesses to different bit fields are combined to reduce the number of accesses. Base-pointers are used systematically to reduce memory overheads from structural reuse.

In simple examples of the PULPino GPIO and SPI drivers the number of memory accesses are reduced by 36%, the estimated run time is reduced by 52% and the driver code size is reduced by 22%. This could be achieved at the cost of a 8.7x larger register map. The source code complexity is reduced by 39% when measured by Halstead effort [4].

2 Generation of Optimized Drivers

2.1 Driver Generation Workflow

In existing workflows, HW design dictates a fixed register layout that is used as input for driver development. The device drivers are then implemented by adhering to the given layout.

Fig. 2 shows our proposed new workflow. The HW designer produces a list of bit fields with their properties, but without a fixed register layout containing offsets. On the SW side, the driver is then developed using a DSL where hard-

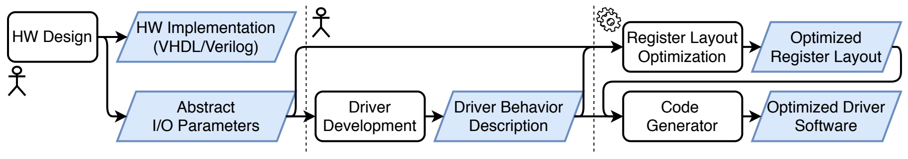
HW Design
HW Implementation (VHDL/Verilog)
Abstract I/O Parameters
Driver Development
Driver Behavior Description
Register Layout Optimization
Optimized Register Layout
Code Generator
Optimized Driver Software

Figure 2: Driver Generation Workflow

ware accesses are expressed using these bit fields explicitly. The register layout optimization then analyzes the access patterns to the bit fields and maps them to the registers such that they can be optimally accessed and driver code size is minimized. The code generator then generates valid C driver source code for the optimized register layout. In a final step, the bus interface of the peripheral needs to be adapted to implement the chosen register layout.

2.2 DSL-based Driver Development

Our proposed DSL-based flow extends the C language with a couple of driver-specific extensions.

2.2.1 Bit field groups

Firstly, the concept of struct definitions is adopted, because they enable a concise way to define members of a type. Struct definitions are familiar to C programmers and are flexible enough to allow additional annotations. To distinguish the new custom definitions from actual structs, instead of the keyword `struct`, the keyword `bfGroup` for *bit field group* is used. These groups have the important distinction to regular C structs, in that the order of their fields is not necessarily laid out in the order of their declarations as mandated by the C Standard [5]. This enables a later optimization of the register layout based on the driver behavior. C does have the language feature *bit fields* to define fields of a struct with bit-accurate sizes, but it does not support hierarchy nor multiplicity. The proposed DSL adds keywords for new type names as `uint1` up to `uint64` to define the bit-accurate size of fields in groups. Besides bit-accurate fields, groups are also allowed to contain fields with type of other groups. The composition of groups like this allows a hierarchical representation that enables code reuse. This is very important as register layout of peripherals often have a hierarchical structure. As final feature groups and bit fields support multiplicity with array types and indexed accesses. Once all the group definitions for a device have been made, the device has to be instantiated. This is done with `make_device( groupname, globalname, base_addr)` which is equivalent to creating a global pointer of type `groupname` with name `globalname` to the base address `base_addr`. To be able to group translation units as device drivers, the function `unit_of_device( globalname)` is introduced. It should be used at global scope in an implementation file. The DSL allows for a very compact description shown

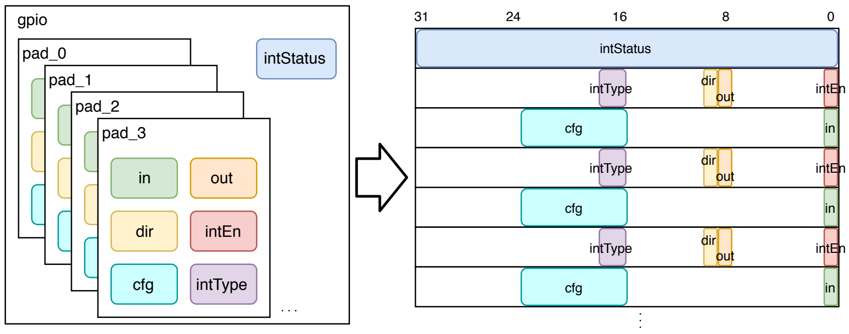

Figure 3: Mapping from abstract bit fields to a concrete layout.

in the following for the GPIO peripheral of the PULPino SoC.

```
bfGroup GPIOPad {
    bit dir;
    bit in;
    bit out;
    bit intEn;
    uint2 intType;
    uint8 cfg;
};
bfGroup GPIOType {
    bfGroup GPIOPad pads[32];
    uint32 rse(intStatus) intStatus;
};
make_device(GPIOType, gpio, 0x1a101000);
```

The bit field group `GPIOPad` combines all bit fields related to one IO pad. The group `GPIOType` uses the array feature to declare 32 pad groups together with one interrupt status register `intStatus`. Finally `make_device` creates the instance `gpio` of type `GPIOType`. This abstract definition of bit fields is free from actual layout information and only later gets transformed into a mapping to registers as shown in Fig. 3.

2.2.2 Hardware side effects

Modern compilers find close-to-optimal representations of the behavior described in source code for both run time and memory size. Therefore we do not aim to improve the compiler internals more for marginal gains, but instead tackle a limitation in the source code description that is especially relevant for the HW/SW interface. In driver code, it is necessary to use the `volatile` type qualifier for accesses to memory-mapped registers [5]. Bit fields may be modified by the HW peripheral or accesses may have hardware side effects, e.g., clear-on-read. Such side effects are often described in data sheets or sometimes can even be retrieved from company internal machine-readable representations. Otherwise, the compiler optimizes the low-level accesses and thereby change behavior in an unwanted way. For example a write statement might be completely eliminated if the same address is never read again. However, using volatile is a very broad constraint that prevents most compiler optimization even if some optimization transformations would be compatible with the device. For example when multiple bit field values are read sequentially from the same register, it might be possible to read them at once, which is a prohibited optimization for volatile accesses. By starting our flow from HW design, we have sufficient information and flexibility to describe these register accesses in a more precise way than with volatile qualifiers. This added information should be expressed as bit field properties. The register layout optimization then takes these additional constraints into account and the code generator can produce more efficient and compact code.

There are multiple ways to provide the required meta information to a generation framework. It could be explicitly provided in a dedicated file, but this has the disadvantage, that closely related information is split into different places. In our approach, the dependencies between bit fields are described in a way that allows the code generator to apply such optimization steps. Dependencies between fields can be expressed with annotations next to group fields. The keyword `rse` is used for *read side effects* and `wse` for *write side effects.* They mark side effects for reading from or writing to the field and specify which other fields may be influenced by the side effect. The general concept of side effect definitions is that a read or write operation to the annotated bit field will cause a side effect on the specified influenced bit field(s) that invalidate any potentially cached or assumed values. In detail, there are several scenarios for side effects:

1. `type rse(a) a`: Whenever a is read, a is invalidated. A read operation on the field must always be followed by a repeated read operation to retrieve a possibly updated value. An example for this is a counter register. After reading it, it might have changed already to a new value. There is no restriction in sharing registers with fields of this side effect type.

2. `type rse(b) a`: Whenever a is read, b is invalidated. A read operation on the field must cause a read operation on the affected field if it is requested afterwards. This behavior is for example displayed by hardware lock registers. A read from them prepares data in another memory location for reading. It is undesired to share registers with fields of this side effect type, because a read operation from an unrelated field will then unnecessarily invalidate the affected field again, possibly with unwanted side effects as in the case of locking.

3. `type wse(a) a`: Whenever a is written to, it causes some side effect outside the abstract state machine and a is invalidated. For example a register that causes a buffer to be sent on a communication peripheral may not cause any visible effect on the device register interface, but has an effect in the real world. Sharing a register that contains this side effect type with other fields is not possible at all since it might cause wrong behavior.

4. `type wse(b) a`: Whenever a is written to, b is invalidated. A reset register for example shows this behavior, since it causes other fields to be set to

their reset value. Obviously, an unrelated write operation to another field may not cause such an effect, so sharing registers with fields of this side effect type is not possible.

Identifiers to declare the influenced fields, which are given to the `rse` or `wse` annotations, can be fields in the same group, other group names or even fields in other groups with the notation `groupname::fieldname`. In the above example of the GPIO peripheral of the PULPino SoC, there is one read side effect in line 11 declared on status register `intStatus` influencing itself.

2.2.3 Driver behavior

Embedded C driver functions use volatile pointer dereference to access bit fields as shown in the following example of a PULPino GPIO driver function.

```
void set_gpio_pin_value(int pinnumber, int value) {
  volatile int v;
  v = *(volatile int*) (GPIO_REG_PADOUT);
  if (value == 0)
    v &= ~(1 << pinnumber);
  else
    v |= 1 << pinnumber;
  *(volatile int*) (GPIO_REG_PADOUT) = v;
}
```

Since this code performs bit field masking directly in the driver implementation instead of using HAL functions, its appearance is overly complex and obfuscates its behavior.

Our DSL code that describes the behavior of driver functions uses the group definitions as if they were standard C struct definitions. The regular struct field accesses will later be transformed into appropriate volatile memory accesses by the code generator. The driver developer can exploit the hierarchical bit field declarations and arrays to write high-level code that facilitates reuse and is free from bit manipulation as shown in the following code of the same GPIO driver function:

```
void set_gpio_pin_value(int pinnumber, int value) {
  gpio->pads[pinnumber].out = value;
}
```

The DSL can be parsed with a unmodified Clang compiler using a support header, which defines our keywords as macros. This support header uses preprocessor directives to define all added keywords (bfGroup, rse, wse, make_device, unit_of_device, bit, uint1-64) as macros. The preprocessor of Clang is then used to extract these language extensions of the DSL and attach their meaning to the corresponding source code.

2.3 Register Layout Optimization

Knowing the driver behavior, a register layout can be generated that is optimized for this specific driver. A major optimization step of the proposed method is the combination of multiple read or write operations into a single one. This is only possible as long as there is no data-flow, control-flow or side effect induced dependency between the accesses.

2.3.1 Control-data-flow analysis

The control and data flow add dependencies between read and write accesses. Hence, first the Control-Data-Flow Graph (CDFG) [1] is extracted from the driver functions defined in the DSL. Since the DSL is compatible with the C language, we use the Clang libraries directly to get a source-level control flow. On top we added our own data-flow analysis. CDFG analysis is required for the proposed method, because statement reordering and merging should generally be allowed as optimization, except when violating data-flow dependencies or explicit side effect dependencies. It is required at source level to be compatible with other source level tools, for example verification tools. Data-flow analysis on source code level is challenging, because of the complex statement representation compared to IR or binary level. However, from those lower levels it is difficult to map the analysis back to source level statements.

2.3.2 Bit Field Access Conflict Graph (BFACG)

For describing all dependencies, we define the Bit Field Access Conflict Graph (BFACG) as a graph $BFACG(A, E)$ with nodes $A = \{a_1, a_2, ..., a_n\}$ representing specific accesses and edges $E = \{e_1, e_2, ..., e_m\}$ representing conflicts among the accesses. An access a_i represents an individual read or write memory access to a bit field occurring in a function of the driver. Multiple accesses can refer to the same bit field. A conflict e_i arises when there is a control or data dependency in the CDFG, always between read and write accesses and through side effects in the bit field definitions.

One BFACG is created per driver function. By using standard graph coloring on each BFACG (nodes without any conflicts end up with the same color), we find *access regions* which represent a group of accesses which may all be combined into a single one.

Fig. 4 shows a BFACG for the following driver function:

```
void combine_example(bfGroup GPIOPad *pad) {
  int dir = pad->dir;
  if (pad->intEn && dir)
    pad->dir = gpio->intStatus != 0;
}
```

The access `pad->dir` and `pad->intEn` could be combined. Hence, the register layout optimization should be guided to map both bit fields to the same register.

2.3.3 Bit field group simplification

The DSL bit field groups represent a high-level hierarchy of different types with possible array multiplicity. When this abstraction is not exercised in driver code by dynamic pointers or array indices, they cause an additional overhead

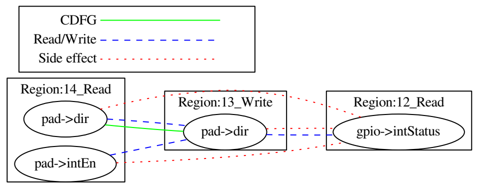

Figure 4: BFACG of function combine_example

to the interface. This is resolved by flattening out the bit field group hierarchy depending on the driver code. If array fields are only accessed by statically determinable array indices, they are expanded into their individual array elements. The members of sub-group elements are integrated into the parent group if the sub-group is never dynamically accessed by pointer. These two methods are applied repeatedly on the tree until no longer possible.

2.3.4 Optimization Heuristic

Using the BFACG of each driver function, the generation creates the register layout by mapping the bit field into registers. The target is to create a layout that enables driver code with minimized code size, run time, number of bus accesses and size of the memory map for the device. First a sorted list `C` of list of bit fields is created. Each entry, called *combination*, holds a subset of bit fields, that could be accessed jointly in a driver function, because they are part of the same access region. The sorting of `C` is done first by the number of access regions that the combination is part of, that is, at how many positions in the driver the accesses may be combined. If that is equal, it is sorted by the number of bit fields inside the combination to favor larger ones.

The algorithm starts at the deepest level of the bit field group hierarchy, with the groups that only contain bit fields. For each bit field we go through the sorted list `C`. If the bit field is part of a combination, all bit fields of that combination are mapped to the same register, as long as they fit. Else, the bit field is mapped into an exclusive register. The higher levels of hierarchy may be groups that contain both bit fields and other child-groups as members. The bit fields are mapped as before to registers, while the registers holding the bit fields of the child-groups are appended to the register list of the parent group. Is it important to notice that if there are different group instances of the same group type, then the mapping of the contained bit fields and groups is only done once and reused. This assures that functions using the base-pointer of this group type can be shared between all its instances. Additionally, if a combination contains arrays of bit fields of same array size, the heuristic maps them element-wise to registers, e.g., the first register holds `bf1[0],bf2[0]`, the second holds `bf1[1],bf2[1]`, etc. This allows to generate HAL functions for combined accesses that support indexing.

2.4 Code Generator

With the final register layout available, it is possible to generate the driver C source code from the DSL. Since the DSL is very near to C, source to source transformation with Clang is used [6].

The generated driver functions are parametrized by their group pointer, which allows for easy reuse and very low code size, e.g., if a device has several channels there is one set of driver functions shared between those channels. To differentiate, the driver functions receive a bit field group pointer as input. Hence, in the generated code the defined hierarchical structure of the bit field groups is kept, making them highly readable and debugable. Groups and bit fields using multiplicity will have an index as additional parameter in the driver function to generate compact code without replication for different indices.

A HAL is generated to allow the drivers to access the hardware. These functions implement the shift and mask operations to extract bit fields and possibly read-modify write operations, if a single bit field is set that shares its containing register with other bit fields. One major motivation for the register layout optimization is the combination of accesses. When the code generator identifies that certain read or write accesses have no conflict in the BFACG of the driver function and are located in the same register of the layout, a HAL function is generated that allows combined access to those bit fields. This improves code size, run time and number of bus accesses as only a single read or write is required.

Group simplifications break the direct relation between groups and access expressions in behavior code. However, driver code may use the original subgroup names that would no longer be recognized. One solution to this is the generation of *support structs* whose sole purpose is to establish relation between hierarchical types. These consist of just sub-group member definitions without bit fields since those are access through HAL functions by address and not by name. The support structs of the GPIO driver look like the following:

```
struct GPIOPad
{
    uint8_t pad0[20];
    // size: 0x00000014
} __attribute__((packed));
struct GPIOType
{
    uint8_t pad0[4];
    // 0x00000004
    struct GPIOPad pads[32];
    uint8_t pad24[620];
    // size: 0x00000284
} __attribute__((packed));
```

With this, we can reuse an existing expression like `gpio->pads[i + 1]` in the generated code to pass to a HAL function and avoid complex expression parsing.

The code generation is illustrated with the example in Sec. 2.3.2. The BFACG shows that accesses to `dir` and `intEn` can be combined. In the generated code this produces a HAL function call that retrieves both values with a single read as shown in the generated code below.

```
void combine_example(bfGroup GPIOPad *pad) {
  uint8_t tmp1, tmp2;
  HAL_READ_GPIOPad_dir_intEn(pad, &tmp1, &tmp2);
  int dir = tmp1;
  if (tmp2 && dir)
    HAL_WRITE_GPIOPad_dir(pad,
      HAL_READ_GPIOType_intStatus(gpio) != 0);
}
```

Since all of these generators are built on a common model of the drivers, we can add new generators easily. For example for the evaluation in Sec. 3, we generate SystemC headers that contain the chosen register layout. In the same way, it is possible to generate IP-Xact descriptions in order to export the final register layout to other hardware related tools.

3 Results

The proposed flow was implemented to generate the drivers for the GPIO and SPI peripherals of the PULPino SoC. The static driver code size was retrieved from the compiled binary optimized for size with `-Os`. The memory map size is a known output of the generator when allocating registers.

In order to evaluate the driver performance and correctness of the driver generator, a SystemC virtual prototype (VP) based on the instruction set simulator ETISS [9] was used. The SystemC modules for the peripherals were implemented such that the register layout can be dynamically changed using a generated header file that specifies the bit field offsets. The VP simulations allow to measure estimated number of CPU cycles and the number of peripheral bus accesses. From the main function we exercise the driver with some initialization and a simple loop over four GPIO pads. The SPI is exercised by sending and receiving a frame.

Table 1 compares the investigated metrics for the original PULPino GPIO driver and the optimized solution based on the heuristic described above. As can be seen, code size, run time and number of required accesses could be reduced significantly by 38%/12%, 60%/43% and 38%/28% for GPIO and SPI respectively, but, at the cost of larger memory map. Another improvement lies in the simpler description of the driver behavior using the DSL, which we measure with the Halstead effort [4]. Here, the effort could be reduced by 85% and 1%. The complexity of SPI code is not reduced because the original driver forwards many registers directly to the user, so bit field extraction happens in the application.

4 Related Work

There are several DSLs for describing driver behavior. Devil [8] and HAIL [11] also provide more granular side effect description than the `volatile` qualifier in C. A major difference to our work is that Devil and HAIL introduce their own language different from C. Devil also fixes the final register layout in the device

Table 1: PULPino Driver Evalution for GPIO and SPI

Variant	Runtime (cycles)	Memory map size (bytes)	Code size (bytes)	Number of accesses	Halstead Effort
original GPIO	639	64	448	100	115k
optimized GPIO	257	840	276	62	17k
original SPI	557	40	728	25	142k
optimized SPI	318	68	640	18	141k
original GPIO+SPI	1196	104	1176	125	257k
optimized GPIO+SPI	575	908	916	80	158k

parameter descriptions, not allowing for register layout optimization. NDL [2] is a DSL that builds on top of Devil by extending it with driver state-level functions, which is outside the scope of our work.

The work in [7] also targets register layout optimization. The objective function includes total code size and a performance metric defined as instruction costs weighted by the number of occurrences during profiling. The authors present an ILP formulation which can optimize for either SW or HW cost, but admit that such an ILP cannot be solved in a reasonable amount of time. As a practical solution they provide two heuristic approaches. This paper approaches the same issue at a higher level. We abandon the concept of a flat collection of bit fields and instead use hierarchies and multiplicity to allow higher level driver code to be more generic and reusable.

The generated HAL functions of our approach use base-pointers to be more generic and facilitate code reuse for small memory footprints. This follows practical HAL coding guidelines as, e.g., described in [3]. In order to verify that the drivers work correctly, [10] proposed a formal verification method for HW-near software, that can be applied.

5 Conclusion

In this report, a driver generation flow was proposed that reduces development effort and memory use of MCUs in the IoT field. This is achieved through abstraction of fixed bit field offsets while following constraints that promote code reuse.

References

[1] S. Amellal and B. Kaminska. Scheduling of a control data flow graph. In *IEEE ISCAS*, 1993.

[2] C. L. Conway and S. A. Edwards. NDL: a domain-specific language for device drivers. In *ACM Sigplan Notices*, volume 39, 2004.

[3] W. Ecker, W. Müller, and R. Dömer. Hardware-dependent software. In *Hardware-dependent Software*. Springer, 2009.

[4] M. H. Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[5] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. ISO, 2018.

[6] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE CGO*, 2004.

[7] K. J. Lin et al. Optimal allocation of I/O device parameters in hardware and software codesign methodology. In *IEEE EUC*, 2007.

[8] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *USENIX OSDI*, 2000.

[9] D. Mueller-Gritschneder et al. The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping. In *IEEE RSP*, 2017.

[10] M. Schwarz, R. Stahl, D. Müller-Gritschneder, U. Schlichtmann, D. Stoffel, and W. Kunz. ACCESS: HW/SW co-equivalence checking for firmware optimization. In *ACM DAC*, 2019.

[11] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: a language for easy and correct device access. In *ACM EMSOFT*, 2005.

[12] A. Traber et al. PULPino: A small single-core RISC-V SoC. In *3rd RISCV Workshop*, 2016.