# Exploring and Comparing IEEE P1687.1 and IEEE 1687 Modeling of Non-TAP Interfaces

Hans-Martin von Staudt
Dialog Semiconductor
Kirchheim unter Teck, Germany
hansmartin.vonstaudt@diasemi.com

Bradford Van Treuren
VT Enterprises Consulting Services
Lambertville, NJ, USA
bradvt59@gmail.com

Jeff Rearick
Advanced Micro Devices
Fort Collins, CO 80528 USA
Jeff.Rearick@amd.com

Michele Portolan
Univ Grenoble Alpes
CNRS, Grenoble INP[1], TIMA
38000 Grenoble, France.
Michele.portolan@grenoble-inp.fr

Martin Keim
Siemens Digital
Industries Software
Wilsonville, OR 97070 USA
Martin_Keim@Mentor.com

***Abstract— The industry is moving forward with non-TAP, chip-level interfaces driving IEEE 1687-2014 networks. Recent literature not only describes such interfaces, like I²C and IEEE 1149.7 variants, but also demonstrates that such interfaces to IEEE 1687 are already proven in silicon. Common to those implementations is the need for "private" extensions of IEEE 1687 to make it support non-TAP interfaces. The goal of IEEE P1687.1 is to directly support non-TAP interfaces. In this work, we summarize the thought progression from IEEE 1687's data register callbacks to IEEE P1687.1's transfer function, which allows alignment with IEEE P2654, and possibly IEEE P1687.2.***

***Keywords—IEEE 1687, IEEE P1687.1, IEEE P1687.2, IEEE P2654, IJTAG, SJTAG, STAM, I2C, SPI, Transfer Function***

## I. INTRODUCTION

The IEEE 1687-2014 standard [1], also known as IJTAG, enjoys a very fast adoption rate in the semiconductor industry. This speed may be due to IJTAG builds on other established standards like IEEE 1500-2005 [2] and IEEE 1149.1-2001 [3], allowing a risk-free adoption of the software side of IJTAG, while keeping the earlier hardware implementation as a backup. Hardware modeled according to these standards are, by construction, compliant with IJTAG. This is in particular true for the 1149.1 compliant TAP controller, which is one of the most common access points to the inside of the design. The IJTAG standard includes specific solutions for the description and easy integration of the TAP controller as the interface between the device IOs and the IJTAG network it hosts.

On the other hand, the IJTAG standard does not provide the same level of descriptive power or integration ease-of-use for other established device IO (industry) standards, like Inter-Integrated Circuit (I²C) [4] or Serial Peripheral Interface (SPI), see e.g., [5]. This lack of support by the standard has not stopped users of IJTAG expanding the reach of their IJTAG-based implementations. Recent papers [6]-[9] describe numerous IJTAG-based solutions and implementations in silicon of non-TAP interfaces, including I²C and IEEE 1149.7-like two-pin serial interfaces driving IJTAG networks.

Typically, these solutions bend or expand the IJTAG standard to support the user's need. IEEE P1687.1 [10] aims at filling this gap in IJTAG by standardizing a method to generically describe device IO interfaces, TAP and non-TAP alike.

The remainder of this paper is structured as follows. The next section introduces the main concepts of IEEE P1687.1 and draws some comparisons to 1687. We will be looking at how device IOs and device interfaces are describable, following along some of the evolution of thinking of the IEEE P1687.1 working group. Section III illustrates in more detail options and difficulties of an I²C modeling in IJTAG. Section IV develops the rudimentary non-TAP concepts of IJTAG into the generic solution needed to describe a multitude of device interfaces. This generalization is captured by Interface-to-Interface Transfer Functions, abstracting away the entire body of the non-TAP interface module into software. Section V illustrates two implementations examples, one of which is the well-known Verilog Direct Programming Interface (DPI) [11], which could serve as a template for IEEE P1687.1. Thereafter, we summarize the paper and provide conclusions.

## II. INTRODUCING IEEE P1687.1

We must state that the IEEE P1687.1 standard is a work-in-progress. No syntax or semantics has been defined. Also, the concepts we are describing here and in the following are currently being discussed and are not final. Nonetheless, the understanding of the problem and solution space at the core of P1687.1 focuses more and more on what is described in this paper.

As mentioned in the introduction, P1687.1 shall enable a user of 1687 to choose a device IO interface that is not an 1149.1 compliant TAP controller. The beforementioned I²C is one example. P1687.1, just as 1687, will be a descriptive standard, not a prescriptive one. This means it will not contain prescriptions how one or the other existing device IO interface needs to be modeled. Instead, a generic description must be found that enables users to describe a wide variety of device IO interfaces, including the currently most common interfaces.

The latter could be provided in the P1687.1 document as examples in the non-normative section of this future standard.

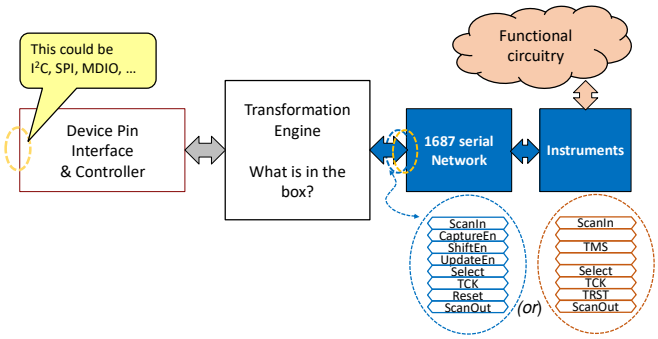



Figure 1: General depiction of the IEEE P1687.1 task

The general problem P1687.1 solves is shown in Figure 1. It is assumed that the device contains a 1687 compliant set of instruments, connected to a 1687 compliant network, which is hosted by an internal, 1149.1 compliant, TAP controller. On the other end of the device, there is the non-TAP Device Pin Interface & Controller (DPIC). Both sides are bridged by a "transformation engine" module. This module translates in both directions between the device's IO data and control protocol and the data and control protocol operating the TAP controller. Valid variations of this picture may have the TAP controller being part of the transformation engine, i.e. the transformation engine module directly hosts the IJTAG network. The task of P1687.1 is finding a way to describe this transformation engine module.

### *A. IEEE 1687 Callbacks and Access Links*

Early on the IEEE P1687.1 working group saw that many of the current interfaces use or can use a (portal) register that serves as the physical data transfer method, see Figure 2. If a DPIC could write data to and read data from this portal register, then the transformation engine only needs to interface between this register and the TAP controller. This was described in an embedded tutorial at ETS 2019 [6]. An evolution step forward was outlined in the introduction sections of [7], presented at ITC 2020.

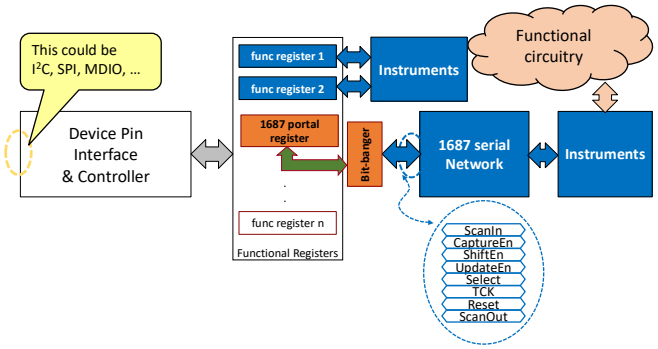
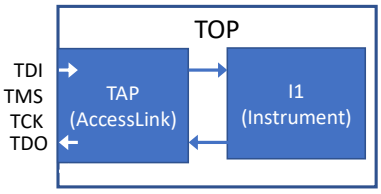



Figure 2: Solution through Portal Registers

Initially, the thinking was guided by the idea to expand the IJTAG concept of callback and access link. An IJTAG callback is a reference to an iProc, written in PDL (level 0 or 1) and attached to a data register. This iProc is called whenever the IJTAG retargeter encounters this data register, hence there can be an iProc that is called when there is a read operation from this data register, and another iProc when writing to the data register is needed. The 1687 standard sets no limits on these iProcs. For example, they can read and write to any other ICL register or port, can be cascaded, iMerged etc. In the most general case, these data register callbacks are nearly impossible to implement in an EDA tool. P1687.1 plans not use such data register callbacks.

1687 provides two types of access links. One, fully developed, for integrating the TAP controller, and a 'generic' one. The latter is hardly developed in 1687, and essentially unusable. Still, the idea of an access link is sound and was considered expandable for the needs of P1687.1. Through this mechanism, one can attach a protocol to an ICL instance, without actually describing much of the body of the ICL module. Figure 3 (top) depicts a simple access link example, as provided in the 1687 standard, with Figure 3 (middle) being the matching ICL code.




```
Module TOP {
 Instance I1 of MyInstrument { }
 AccessLink TAP of STD_1149_1_2001 {
  BSDLEntity TOP ;
  my_ijtag_en { // instruction name
   ScanInterface { I1.scan_client; }
  }
 }
}
```

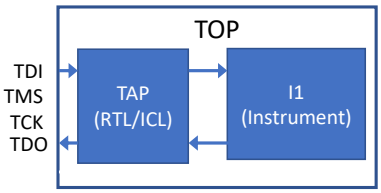



Figure 3: IEEE 1687 AccessLink Example

One observation how 1687 defines the access link is that is no longer matches the (RTL) design implementation. Note that in Figure 3 (top) the TDI, TMS, TCK, and TDO ports are all parts of the TAP (AccessLink) ICL 'instance'. In reality, there is no such instance, as the TAP module instance in the design has its own pins with its own pin names, connected to the device IO with its own port names, as shown in Figure 3 (bottom). This description discrepancy between ICL and the design is usually not a problem but might become an obstacle when one needs to concatenate multiple such transformation engines.

### *B. IEEE P2654 and IEEE P1687.2 Crossovers*

This concatenation may be an essential aspect of IEEE P2654 [12], and possibly P1687.2 [13][14], hence the access link description should be restricted to the design module boundaries.

The task in P2654 is to bring, for example, multiple devices into one description model at the system level, so that (IJTAG) patterns can be generated and applied across devices and across a variety of device IO interfaces.

P1687.2 is an IJTAG derivative that addresses analog ports and operate analog instruments. For example, P1687.2 might need to translate a voltage level at the input to another voltage level at the output of the transformation engine. In P1687.2 it is possible that analog properties undergo multiple transformations between the analog instrument and the device IO for example, see Figure 4

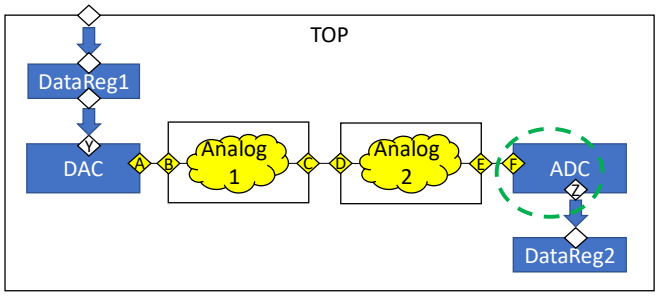



Figure 4: Concatenation of transformation modules

It becomes clear that all three standards look at different aspects of the very same problem: There's a design entity that transforms input data to output data and vice versa but is too complex to be described in any reasonable extension of ICL. On the other side, it is very clear what this transformation needs to do. One can easily sketch out how one set of data needs be transformed into the other set of data (in either direction), i.e. one can easily write an algorithm for this.

### C. *Transfer Function*

We finally arrived at the understanding that "all" we need to define is a description method for this algorithm in a way that is useful for all three standards.

It seems that only two algorithms are needed, one for taking the data at all the inputs of the module and compute the output data, and one for taking the data at the all the outputs and compute the input data. In engineering such a method is referred to as a Transfer Function, a mathematical concept. A transfer function models the output response of an electronic component for a range of possible input stimuli. Filters are typical implementations to transfer functions.

In the simplest case the transfer function for P1687.1 is strictly a combinational function from n binary-valued module inputs to m binary-valued module outputs. It is attached to an ICL instance by means of an AccessLink-like construct and executed through the software engineering method of callbacks by the EDA software. More on this in Section IV and Section V.

## III. I²C Modeling with IEEE 1687

The I²C interface is the epitome of the low-level digital host interface for small and medium chips, attached to embedded microcontrollers. It was presented by Philips in 1982 and soon became popular in the industry.

### A. *I²C and SPI Basics: Addressed Access Schemes*

Like other such interfaces, e.g. the Serial Peripheral Interface (SPI) communication happens bit-serially on very few wires. I²C uses two wires: one for clock and one for bidirectional data. The original specification only defines transactions composed from two elements as shown in Figure 5a and Figure 5b:

1) An address byte selecting one of many slave devices and determining the data direction of the second element.

2) One or more data bytes, either read or write direction

Around the bytes there are few more clever details (start/stop conditions, negative acknowledge, etc.) which all contributed to the popularity of the interface but in the context of this paper only the data transfer model is the focus.

Of course, accessing always all bits in a device is not overly flexible. Most devices therefore implement an internal address scheme. The first data byte as per the above definition becomes a device internal register address (not to be confused with the first element, which was a device address on the bus), followed by the payload data. A generic write access hence consists of three bytes: the device address, the register address, followed by the write payload data. See Figure 5c.

Unfortunately, the data direction cannot change within a transaction. Remember: the first byte contains the direction bit together with the device address. Hence, a read access following this scheme needs to consist of two transactions as shown in Figure 5d.

1) Write the register address

2) Read the payload data

The fundamental organization of addressed registers fits very well with the data model of microprocessors that access memory locations on a map. As no surprise there are variants of the protocol using 2 bytes for addressing. Also, there is no hard need to restrict the payload data to 8-bit chunks.

In contrast to I²C, SPI does not operate a bidirectional data line but features dedicated input and output data pins. Also, no bus addressing happens. Instead, a chip select signal chooses the slave to respond to the master. However, the underlying data model of the SPI may be just the same: Addressed register locations, each 8-bit wide.

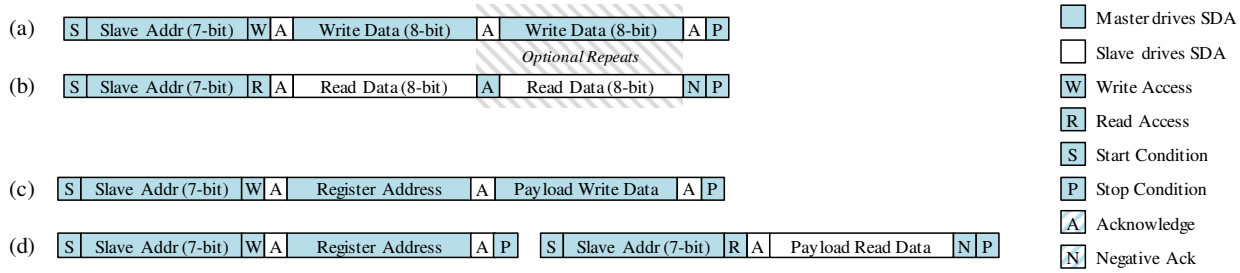



Figure 5: I²C transactions: Raw (a), (b). Addressed (c), (d)

Once a chip features such a host interface for the various registers controlling its functions (write) and presenting acquired data (read), it is quite natural to use it as Test Access Port (TAP) too. Hence there is no appetite for such small to medium size chips to implement another TAP like the one defined by 1149.1.

The concept of accessing a bit by the register address and the position within the register is sufficiently different to the 1149.1 idea of locating a bit by its position on a scan chain. Handling long and reconfigurable scan chains is fine in a test-only environment where patterns are pre-computed offline. An application style access to test registers is simpler and more attractive.

Also, it is more economical compared to the scan approach of 1149.1. A boundary scan register or the Test Data Register (TDR) as per the 1687 standard requires an extra flip-flop per control bit. The overhead of I²C and other addressed schemes is the de-serializer. It grows from a base line determined by the width of the data word only logarithmically with the number of bits. Let n be the number of bits and w the width of the data word (=8 for I²C and a set SPI configuration) then the number of required flip-flops is as follows.

- Scan based flop count: $n + n = 2n$
- Addressed flop count: $n + \log_2(n/w) + w$

One might say this is no longer relevant in the age of single digit nanometer technologies, but this is just one end of the spectrum. Mixed-signal Big-A/little-d devices, which only feature 5V logic (= 0.6µm) for cost reasons, are ubiquitous. A very interesting example was described in [15]. It showed a single pin implementation with clock and data on the same line, again together with an addressed access scheme.

It is fair to say that for small and medium chips there will always be a too high barrier for an 1149.1 TAP and scan chains to be accepted. Nevertheless, the design and test engineers of these chips equally want to benefit from the principles of IJTAG.

### *B. Approaches to Use I²C and IJTAG Together?*

Even though IEEE 1687 primarily describes scan-based access networks attached to an 1149.1 TAP it offers a few options and pointers to alternatives.

#### *AccessLink*

The will to support other external interfaces can be seen from the AccessLink instruction in ICL. It specifies the TAP which is connected to the scan-based access network. However, only one single type STD_1149_1_2001 is defined. IEEE 1687 simply points to the "user community" to define other interfaces. This is the purpose of P1687.1.

#### *Hardware Portal Solutions*

One of the challenges with any I²C implementation is that the access network cannot necessarily be described in a cycle accurate manner, as is the case with scan-based interfaces. There are three popular implementations:

a) Fully asynchronous operation driven by the I²C clock, which after deserialization also forwards the payload data to/from the register.

b) Asynchronous deserialization of the I²C signals. The complete transaction is then synchronized with an internal clock, which also forwards the payload data to/from the register.

c) Oversampling of the I²C signals with a chip-internal clock, which processes the complete transfer in a synchronous system.

Only option a) represents cycle accurate behavior, controlled by the I²C clock, for which a state machine could be described like the 1149.1 TAP. The other two implementations run on an internal clock.

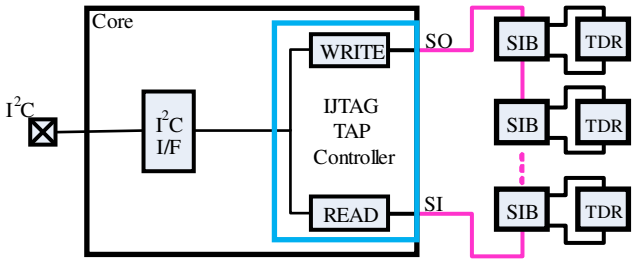



Figure 6: Portal Registers connecting I²C to IJTAG scan chain of "islands"

In [16] and [17] a direct mapping of an IJTAG scan access network to a portal register is described. As shown in Figure 6, the scan network is made up of so called "test and trim islands", each featuring a segment insertion bit (SIB) and a uniform TDR. The portal IJTAG TAP controller services a range of device addresses and maps them to the islands. An I²C write transaction is translated by the portal engine to a sequence of transactions of the IJTAG scan network, opening the respective SIB and writing/reading the payload data.

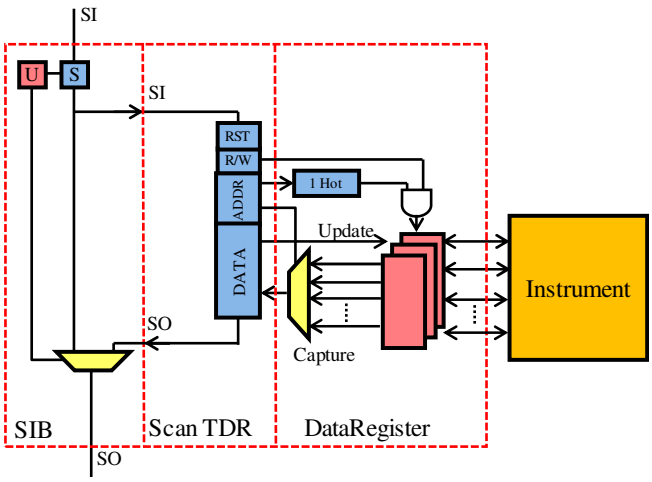



Figure 7: Island architecture with addressed register array

The interesting element of this solution is that it exploits an option which was already foreseen by the 1687 standard: Addressable DataRegisters. As shown in Figure 7 the address for selecting the DataRegister comes from a ScanRegister. Also, the data source and the destination of the DataRegister is a ScanRegister. Hence it replicates the mechanism of the I²C interface itself: A serializer/deserializer followed by parallel read/write access. Compatibility is achieved by selecting the DataRegister 8 bits wide and allocating 16 addresses (4-bit

addressing) for each island. The IJTAG network is not advertised to the outside world (no ICL required).

*Callback Registers*

Another alternative access mechanism foreseen in the 1687 standard is DataRegisters with an associated "callback". For these registers no description of an access network is required. The standard states: "The callback register exists outside the logical connectivity described in ICL and instead relies on prewritten PDL procedures". This mechanism avoids all complications introduced by chip-internal clocking (see discussion on I²C implementations above).

The authors of [9] have exploited this concept. An iWrite or an iRead command to the DataRegister in the device is triggering the callback, which creates the bitstream of the transaction at the chip pins and hence can be mapped directly to the ATE.

The drawback, which is usually associated with the use of PDL is that it describes the access mechanism only in one direction, the direction of retargeting. Implementers and tool writers may find it necessary to know the exact mapping of bits between original and retargeted PDL, e.g. for updating the original PDL if changes are done to the PDL and test pattern on the ATE.

Another problem is the handling of registers and bitfields longer than 8 bits, which is the limit for an I²C atomic access. Imagine a 12-bit DAC whose controlling bitfield unavoidably stretches 2 I²C registers. If only the LSB needs changing, will all 12 bits be written (2 I²C transactions) or just what is changing (1 transaction)? Or, what happens if a carry-over from one register to the other happens such that both need changing? Which one is written first?

These examples show some of the limits of the existing callback definition in the 1687 standard.

## IV. Modeling Non-TAP Interface with IEEE P1687.1

As mentioned in Section II above, the essential contribution planned for P1687.1 is to formalize the mechanism to mate a 1687 network to a device interface other than an 1149.1 TAP. Also as noted above, the primary device interface candidates like I²C and SPI are often utilized as the conduit to exchange data with a bank of addressable data registers. The alternate mechanism of using callback routines to communicate with such registers became the inspiration for the technical solution currently favored by the P1687.1 Working Group.

### A. Expanding on the Callback DataRegister Notion

A Callback DataRegister is said to be disembodied since its hardware connectivity is not described in ICL. Rather, its definition includes one or two callback procedures, WriteCallback and ReadCallback, which the retargeter summarily invokes when it sees this register referenced in the PDL test that is being retargeted. This approach is useful for handling cases with complex connectivity which might prove too challenging for a retargeter to solve, or for cases where the IP provider wishes to keep the details of the connectivity and network hidden from the user. Both of those reasons apply to the situation at hand: retargeting through the circuitry between the edge of the 1687 network and a non-TAP device interface.

### B. Viewing DPIC as a Callback DataRegister

Figure 8 shows an abstraction of the region of interest for P1687.1: the circuitry between the (non-TAP) device interface pins and the edge of the 1687 network. That edge consists of two types of interfaces: a ScanInterface (defined in 1687) and a DataInterface (new in P1687.1 but built entirely from original 1687 elements).

Though it is conceivable that the cloud of circuitry between the DPIC pins and those interfaces could be described in ICL, writing that ICL might be a challenging or undesirable task. The DPIC may contain clock generators, data FIFOs, retimers, and other structures that might confound a retargeter. The DPIC could also contain proprietary features that the vendor may not wish to share. Most importantly, though, none of that high-effort ICL is even necessary: all the details can be abstracted away by simply viewing the entire DPIC in the same way as a Callback DataRegister. Specifically, rather than describing the hardware, the ICL for the DPIC module could simply point to callback procedures which would be invoked when the retargeter encountered any of the interfaces.

This trick allows the DPIC creator to capture the behavior of the circuitry rather than try to convey its details to a retargeter, just as is done for the 1687 Callback DataRegister. There are, of course, many details of the syntax and semantics involved in documenting a callback for a DPIC; those constitute much of the work by the P1687.1 team and are beyond the scope of this paper. Suffice it to say that there will be a mechanism (ideally AccessLink) for the retargeter to recognize when and how to invoke a DPIC callback.

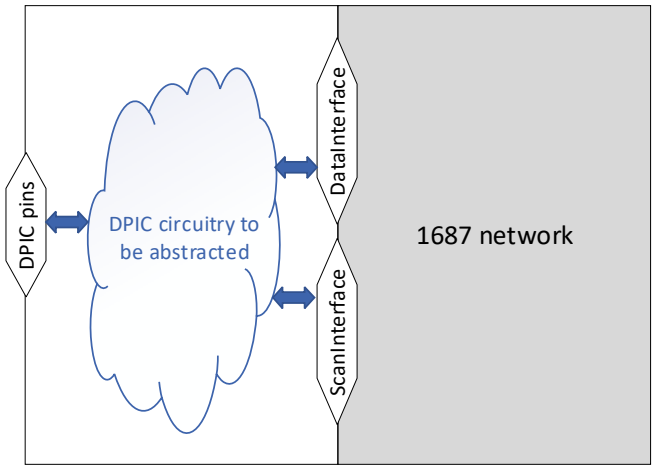



Figure 8: DPIC connection to 1687 network

### C. Viewing a callback procedure as a macro expansion

Taking a similarly abstract view of the operation of a callback procedure during retargeting leads to the simple conclusion: the contents of a callback can be thought of as a macro (i.e. a pre-defined snippet of PDL code) which is expanded in place where the original PDL test made a reference to the object whose ICL points to a callback procedure (e.g. a DataRegister in IEEE 1687, or a DPIC in P1687.1).

The author of the callback procedure is effectively stepping in to tell the retargeter exactly what to do, whether that means selecting a particular data path from several choices, setting up a specific set of values to enable communication, or even manipulating the data through

complicated circuitry (e.g. going through an adder by setting the other operand to zero, calculating parity bits to augment the payload, dividing the data into multiple words to be sent sequentially, etc.). The callback relieves the retargeter from the burden of figuring out how to move data. Of course, this means that the author accepts that burden, but in the case of modeling the DPIC behavior, the author will likely be the designer of the DPIC and thus well qualified.

### *D. Beyond PDL: Specifying Transactions at the DPIC*

The original 1687 standard dictates that callbacks be written in PDL, which makes perfect sense given this view of a callback as a macro to be expanded in-line with the source PDL being retargeted. However, in the broader context being embraced by P1687.1, and for extension into P2654, this restriction to PDL becomes not just awkward, but limiting: PDL enables just primitive bit-banging of pins, whereas most DPICs come with meaningful protocols that are often codified in high-level application programming interface (API) calls.

The logical step to resolve this limitation is to remove the restriction of writing DPIC callback procedures in only PDL. Introducing language flexibility at the callback level will allow non-TAP devices to be exercised by the software which is native to the embedded system in which the devices are used. For example, the result from a DPIC callback for an $I^2C$ interface might look like a sequence of "write_i2c(w_data)" and "read_i2c(r_data)" statements, which is far more useful than a (much longer) sequence of "iWrite SCK 0b0; iApply; iWrite SDA 0b0; iApply; iWrite SCK 0b1; iApply; …" commands.

As simple as that sounds, however, there are many details to manage: language compatibility across devices from different vendors in the same system, standard data types, neutral data exchange formats, consistent language and compiler versions, and even low-level details like endianness and byte ordering. The final section of this paper will show a working example which resolves those issues.

### *E. Mapping Payload Information*

Independent of the language details, a callback procedure has two types of content: static operations which sensitize data paths and/or walk-through state machines, and symbolic operations which manipulate the data payload itself. Tracking that payload information through the various retargeting and transformation steps is essential for debugging and diagnosis of failures, so special steps may need to be taken to facilitate these tasks.

The most direct method to explicitly track the data payload is to insert comments into the retargeted test whenever a piece of data has been manipulated. In the context of a DataRegister callback done entirely in PDL, this can take the form of a PDL iNote command that will be persevered in the retargeted output. For a DPIC callback, particularly if it is written in a different language than PDL as described above, some equivalent mechanism will need to be provided (e.g. special comment annotations, extra procedure calls which can serve as hooks for debugging, etc.).

### *F. Generalizing: Interface-to-Interface Transfer Functions*

In light of the previous two abstractions (using pre-coded macros to represent circuit behavior instead of using a hardware connectivity description, and flexibly using programming languages to implement callback procedures to replace retargeting), it became clear to the P1687.1 and P2654 Working Groups that there could be a general alignment across the different scopes of those two standardization efforts. This alignment (which we intend to guide the development of both standards) is built around two key pillars: interfaces and transfer functions.

#### *Interfaces*

Though it wasn't originally conceived of in 1687 in this way, the process of retargeting a test can be thought of as making successive hops through modules, each of which has an interface to its neighbors. The objective of the retargeter is to convert an action (writing or reading some data payload) at one interface to a corresponding action at the next interface until it arrives at the desired endpoint.

In the proposed formulation, we think of each device as having a client interface (connected to an upstream host) and a host interface (connected to a downstream client), with device logic the middle of the black box between its interfaces. The retargeter needs to be aware of the interconnection of the interfaces, but not of the logic between them (since that will be abstracted away into the callback procedures by the designer who created the logic). The job of the retargeter is thus reduced to navigating a path between interfaces connected by callback procedures and passing the manipulated data payload between them.

The objectives of both P1687.1 and P2427 fit into this very generalized vision, with P1687.1 representing the most downstream endpoint device of a chain of other devices all handled by P2654 at a board or system level.

#### *Transfer Functions*

Given a device with a client interface on one side and a host interface on the other side, the job of the callback procedure is to move data from one side to the other. In fact, there is generally a callback procedure for each direction. This is exactly analogous to the role of a transfer function (and a reverse transfer function) in system engineering. However, instead of a single equation, these transfer functions may be implemented by blocks of code which make requests and provide responses, and they do so with standardized data structures representing the payload and other static control bits.

It is clear that a transfer function is not just a simple combinational function describing an input to output transfer. For example, think of an 1149.1 compliant TAP controller as the module to model. It is a combination of logic functions and a state machine. A second example would be an IEEE 1149.7 [18] two-pin interface. For this, a single cycle for the IJTAG network is translated into 4 cycles at the interface, see e.g. [7] for details.

Transfer functions must also be reversible: The retargeter computes a pattern through the network up to the host scan interface. Using the transfer function, certain input sequences are then computed. These computed input sequences, when applied to the client interface (connected to the DPIC), must truthfully 'replay' the retargeted pattern at the host scan interface. The reverse must also hold true.

Being able to describe existing interfaces like $I^2C$, SPI, 1149.1 TAP, and 1149.7's two-pin solution are all good milestones for the P1687.1 working group to validate their proposed solution.

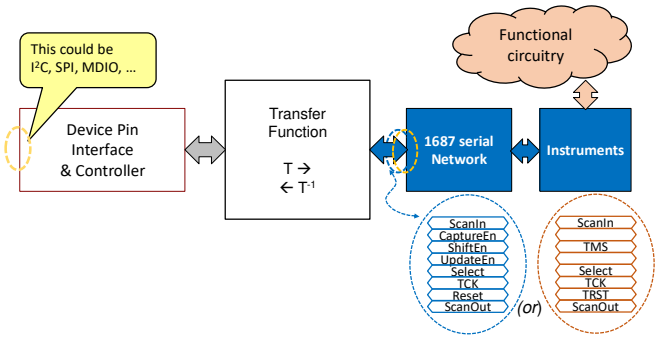



Figure 9: Transfer Function

Putting the properties of the transfer function together, what seems to appear is that the transformation engine from Figure 1 is realized by a finite-state machine, where the transfer function implements the machine's transition function, Figure 9. Following the discussion in this section, a Mealy machine however, seems not sufficient to express the DPIC transaction from Section IV.D. Maybe a Unified Modeling Language (UML) state machine is what is needed. The P1687.1 working group has not worked through the details yet - this is work-in-progress.

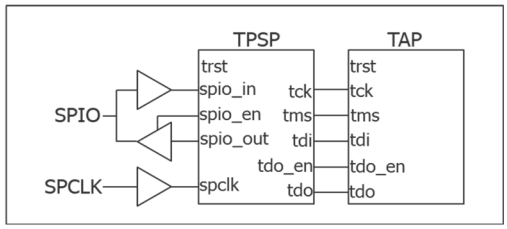
Nonetheless, for illustration purposes only, Figure 10 shows an example. By no means shall this be understood as a syntax or semantic finalized by any of the working groups. It can only be understood as an illustration of the principles. One can imagine port-specific transfer functions; this example, however, defines a transfer function for the entire module. In fact, there are two transfer functions needed. One for deriving the left side module interface from the right side, and one for deriving the right-side module interface from the left side. Since the TPSP module of this example requires three cycles on its left side interface for each one cycle on its right-side interface, see [19], this example uses a 'cycle' to model this. A significant take-away is that the transfer function as used here creates a template for an EDA tool, how the ports relate to each other. Using this template, an EDA tool would never need to 'copy' data to the user's code but could compute the transformation entirely on its internal data structures, thus improving throughput and eliminating user errors in coding.

Other transfer function example descriptions for the transfer function for this TPSP module could use the 3-state FSM of the TPSP instead of an unrolling into individual cycles. Yet, another example could be a closed-form of the function itself, or low-level user code, copying data from ports to ports. Much work needs to be done in the P1687.1 working group.

To make this abstract and generalized scheme of interfaces and callback procedures implementing transfer functions more concrete, Section V will review a working implementation.

### G. *Scalable Payload Mapping*

As stated before, tracking payload information through the various retargeting and transformation steps is essential for debugging and diagnosis of failures. The transfer function manages the payload mapping from the scope of the downstream host interface to that of the upstream client interface as part of the transfer function. Likewise, the reverse transfer function manages the payload mapping from the scope of the upstream client interface into the scope of the downstream host interface. Thus, each callback level in the model resolves the payload mapping in the scope it




```
Module TPSP {
 DataInPort spio_in ;
 ClockPort  spclk ;
 […]
 ToTCKPort  tck ;
 ToTMSPort  tms ;
 […]
}

iProcsForModule TPSP {
 JustifyTransfer {} {
  # derive left side from right side
  Cycle 1 {
   spio_in  = tdi ;
   spio_en  = 0b0 ;
   spio_out = 0bx ;
   pulse      spclk ;
  }
  Cycle 2 {
   spio_in  = tms ;
   spio_en  = 0b0 ;
   spio_out = 0bx ;
   pulse      spclk ;
  }
  Cycle 3 {
   spio_in  = 0bx ;
   spio_en  = 0b1 ;
   spio_out = tdo ;
   pulse      spclk ;
  }
 }

 PropagateTransfer {} {
  # derive right side from left side
  […]
 }
}
```

Figure 10: Example of a transfer function for a TPSP module

understands by definition of the input and output mappings performed by the transfer functions. Therefore, there is no longer a dependence on sharing payload mapping information from the leaf entities to the top level interface scope. The reverse transfer function provides the unwinding of response payload content to the correct positions in the interface scope where the data is received. This provides a scalable solution for payload mapping for any circuit complexity. Diagnosis may then be performed at the lowest retargeted level where the context is known.

## V. A WORKING EXAMPLE OF IEEE P1687.1 CALLBACKS

The callback is a classical computer science strategy that allows a "Main" program to execute a function that is defined in an external Library but which is not available at compilation time and that therefore cannot be linked through traditional means. This is done by placing a placeholder symbol in the Main executable file, which is resolved at execution during the "load-linking" phase. The code is therefore able to execute the external function and receives "back" the result, as depicted in Figure 11.

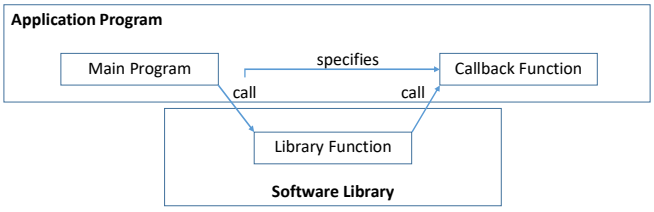



Figure 11: Callback Scheme, from Wikipedia

Callbacks are pretty straightforward when both the Main and the Library have been compiled from the same Programming Language but can become pretty tricky when this is not the case. In those cases, special caution must be put to correctly specify the formats of both input and output data.

### A. *SystemVerilog DPI*

In EDA, a perfect example is the Verilog Direct Programming Interface (DPI) [11], which allows linking a SystemVerilog (SV) Simulator with an external Library. This is done by defining two "layers":

- A SystemVerilog (SV) layer, which defines the data types and functions calls from the Simulator point of view. Functions can either be "imported" (external functions executed in the simulator") or "exported" (SV functions which can be called from the external code). This takes the form of `import` and `export` pragmas to be used in the SV testbench file.
- A DPI Foreign Language Layer, which defines the Application Programming Interface (API) for a given language to specify argument passing and data type conversion. This takes the form of a normative svdpi.h header that must be provided by all simulators.

The standard defines a DPI-C layer, allowing users to add their own code, or to develop a C wrapper for their code. Figure 12 shows the final setup: The User source code (left-hand side of the picture) is compiled and linked against SV DPI libraries (not depicted) to obtain an Object Code (in the middle), that is then loaded at run-time by the Simulator into the final SV application (right-hand side of the picture).

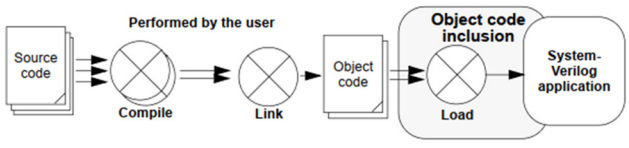



Figure 12: Inclusion of object code into a SystemVerilog application, from [11]

The two layers allow easy symbolic referencing, as depicted in Figure 13 (based on a code example from [11]): on the right-hand side, the SV layer defines an import and an export point. This is mirrored by the C DPI layer on the left-hand side. The import of the svdpi.h header assures the usage of compatible types and references.

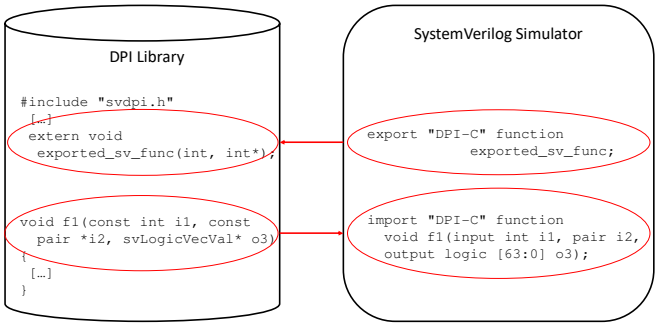



Figure 13: Symbolic referencing in SV DPI

While the DPI Object Library itself is not directly inter-operable, all EDA tools provide examples and compilation makefiles and the standard mandates specific command-line options for the SV compilers, making porting between simulators trivial.

### B. *IEEE P1687.1 and IEEE P2654: Callbacks for Test*

The P1687.1 and P2654 working groups have both been investigating the issue of access to the test infrastructure, even if from different points of view: as an extension of chip-level DFT for the former, and as a system-level Test Access Management for the latter. Both came to the same conclusions: the high variability of solutions makes it close to impossible to define a "one-solution-fits-all" new language.

Even though neither standard is yet complete, the general consensus is to move to a callback-based solution, Figure 11, on transfer functions on the stream of payload, user data and protocol/command information. The working groups are converging over the definition of the exchange format as a derivation of the Relocatable Vector Format first introduced in [20] but no decision has been made yet on the exact form the callbacks method will assume in the standards.

### C. *MAST: A Callback-Enhanced EDA Tool*

In this section, we will show what a P1687.1 EDA tool would look like using the MAST tool [21] from TIMA as a reference. First developed for pure IEEE 1687 retargeting, MAST has since been used as the basis for the development of the RVF format and the callback model. Its P1687.1 setup, depicted in Figure 14, is divided in three parts:

- A 1687 part, in the top half of the figure, where a C++ algorithm containing PDL 0 and 1 commands thanks to a "1687 CPP Player" is compiled into an object library
- A P1687.1 part, in the lower half of the figure, where the callbacks implementations for the Interfaces are compiled against a "P1687.1 Layer", modelled upon the SV DPI model
- The MAST kernel, in the middle, which loads the necessary libraries depending on the system description in ICL. As its P1687.1 extensions are not fixed yet, we use our "Simplified ICL Tree" (SIT) language to experiment with the new features. The kernel interfaces with the back-end (simulation, emulation or other) and also produces a log file for debug purposes.

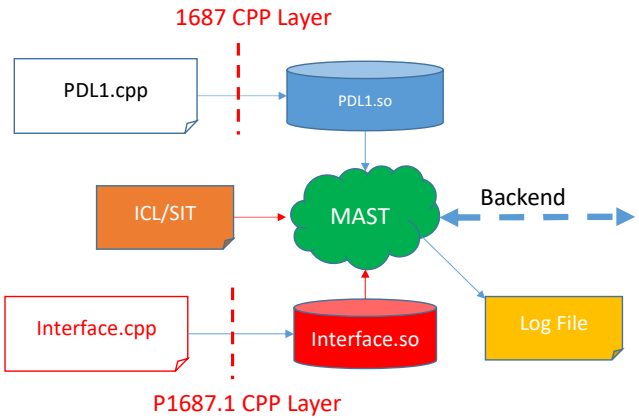



Figure 14: MAST P1687.1 Setup

The SIT description of a system is a straightforward representation of its hierarchy. For instance, the following code describes a JTAG DPIC connected to a simulation backend. For simplicity, the 1687 network is reduced to a single register.

```
TRANSLATOR top Simulation
(
 JTAG_TAP TAP 4 1
 PDL PDL1
 (
   REGISTER regHI 12 Bypass: "0xABC"
  )

 )
```

The P1687.1 layer proposes a set of headers that allow the description of a DPIC or Transformation engine. The most important parts are:

- "BinaryVector.hpp", which provides an unified representation of binary vector data, in the same way SystemVerilog proposes svLogicVal for DPI
- "RVF.hpp" provides the base classes for the Relocatable Vector Format [21], as can be seen in the following code snippet:

```
class RVFRequest
{
 […]
  private:
  std::string m_CallbackId;
  BinaryVector  m_ToSutVector;
  void* m_interfaceData=nullptr;
[…]
}
```

```
class RVFResponse
{
 […]
  private:
  BinaryVector  m_ToSutVector;
 […]

}
```

- The callback identifier is provided as a string for easier debug, while the interface-specific data is of type void for maximal portability. Depending on the interface, it could take any form: an address table for $I^2C$ or SPI, the baud speed for an UART, etc… Its interpretation is up to the actual callback, as recommended by the Delegation Design Pattern [22].
- "TranslatorProtocol.hpp", which provides the base virtual class for transformations. It defines the callback used to translate the packets composing an RVF stream, as shown in this snippet:

```
class AccessInterfaceTranslatorProtocol
{
   […]

 Virtual BinaryVector TransformationCallback
      (CallbackRequest current_request) = 0;
   […]

};
```

- "TranslatorProtocolFactory.hpp" implements a classical Factory Design Pattern [22] that allows MAST to instantiate and parametrize Protocols defined in an external library. It is notably used inside the SIT parser.

By combining these elements, MAST can provide the P1687.1 execution flow depicted in Figure 15 for the SIT file mentioned previously.

Upon startup, MAST scans its configuration directories for available protocol plugins, and loads them thanks to the factory wrapper. This same interface is then used during SIT parsing to construct the desired protocols, identified by their factory registration. In this example, the "JTAG" and "Simulation" protocols. The execution is then a loop of 1687 retargeting + P1687.1 transformations.

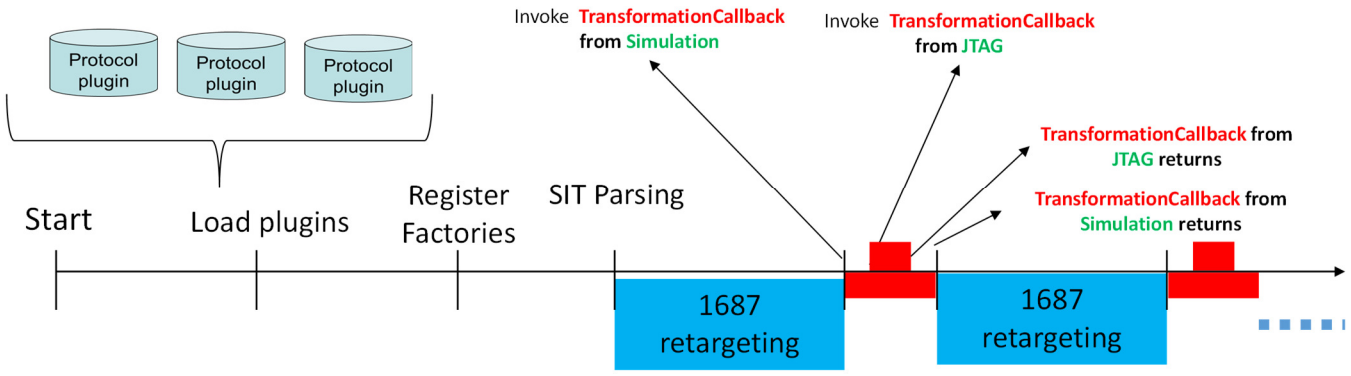



Figure 15: MAST P1687.1 execution flow

## VI. CONCLUSIONS AND PERSPECTIVES

In this paper we restated the need for IJTAG to support non-TAP device interfaces, especially for small and medium size designs, where $I^2C$ and SPI interfaces are common. Modeling such interfaces with standards IEEE 1687 requires bending the rules or inserting non-standard extensions. Still, the modeling remains difficult.

IEEE P1687.1 promises to expand the IJTAG standard covering generically a wide array of interfaces, current and future ones. The P1687.1 working group came to realize that building on 1687's foundation, simply elaborating the already defined use model of data register callbacks, is insufficient. Instead, the callback idea grew into the generalized concept of Interface-to-Interface transfer of information by means of transfer functions, defined by the owner of the (device interface) module. This concept is not limited to digital device pins, but may even include analog pins from IEEE P1687.2, or system interfaces from IEEE P2654.

The EDA industry already uses such concepts. We described briefly how SystemVerilog simulators are expanded using the Verilog Direct Programming Interface. We also elaborated an example implementation in MAST, further underlining the applicability of the concepts currently being developed by the IEEE P1687.1 working group.

## REFERENCES


[1] IEEE Std 1687-2014, "IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device", IEEE, USA, 2014

[2] IEEE std 1500-2005, IEEE Standard for Embedded Core Test, IEEE, USA, 2005.

[3] IEEE Std 1149.1-2001, "IEEE Standard Test Access Port and Boundary-Scan Architecture", IEEE, USA, 2001.

[4] UM10204 $I^2C$-bus specification and user manual, https://www.nxp.com/docs/en/user-guide/UM10204.pdf, Rev. 6 — 4 April 2014.

[5] Piyu Dhaker, "Introduction to SPI Interface", Analog Dialogue 52-09, https://www.analog.com/media/en/analog-dialogue/volume-52/number-3/introduction-to-spi-interface.pdf, September 2018

[6] M. Portolan, J. Rearick, M. Keim, "Linking Chip, Board, and System Test via Standards", 2020 IEEE European Test Symposium (ETS), May 2020.

[7] Modeling Novel Non-IJTAG IEEE 1687.1 like Architectures, M. Laisne et.al, ITC 2020

[8] IJTAG Through a Two-Pin Chip Interface, Jonathan Gaudet et. al., ITC 2020

[9] H. M. v. Staudt, M. A. Benhebibi, J. Rearick and M. Laisne, "Industrial Application of IJTAG Standards to the Test of Big-A/little-d devices," 2020 IEEE International Test Conference (ITC), 2020, pp. 1-10, doi: 10.1109/ITC44778.2020.9325267.

[10] IEEE P1687.1 – Standard for the Application of Interfaces and Controllers to Access 1687 IJTAG Networks Embedded Within Semiconductor Devices, https://standards.ieee.org/project/1687_1.html

[11] IEEE std 1800-2012, "SystemVerilog -Unified Hardware Design, Specification, and Verification Language", IEEE, USA, 2012.

[12] IEEE P2654 – Standard for System Test Access Management (STAM) to Enable Use of Sub-System Test Capabilities at Higher Architectural Levels, https://standards.ieee.org/project/2654.html

[13] IEEE P1687.2 – Standard for Describing Analog Test Access and Control, https://standards.ieee.org/project/1687_2.html

[14] P. Sarson and J. Rearick, "Use models for extending IEEE 1687 to analog test," 2017 IEEE International Test Conference (ITC), Fort Worth, TX, 2017, pp. 1-8.

[15] M. Laisne, H. M. von Staudt, S. Bhalerao and M. Eason, "Single-pin test control for Big A, little D devices," *2017 IEEE International Test Conference (ITC), Fort Worth, TX*, 2017, pp. 1-10.

[16] H. M. von Staudt and A. Spyronasios, "Using IJTAG digital islands in analogue circuits to perform trim and test functions," Poster at IEEE International Test Conference, Paris, 2015, pp. 1-5.

[17] H. M. von Staudt and A. Spyronasios, " Integration of IJTAG Test and Trim Islands in I²C Legacy Designs," 2015 IEEE 20th International Mixed-Signals Testing Workshop (IMSTW), Anaheim, 2015

[18] IEEE 1149.7 - IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture, https://standards.ieee.org/standard/1149_7-2009.html

[19] Manu Baby;Bernd Büttner;Piet Engelke;Ulrike Pfannkuchen;Reinhard Meier;Jonathan Gaudet;J-F Côté;Givargis Danialy;Martin Keim;Lori Schramm, "IJTAG Through a Two-Pin Chip Interface", 2020 IEEE International Test Conference (ITC), 2020, pp. 1-5, doi: 10.1109/ITC44778.2020.9325232.

[20] M. Portolan, "The Automated Test Flow, the Present and the Future", IEEE Transactions on Computer-Aided Design (TCAD), DOI: 10.1109/TCAD.2019.2961328, December 2019

[21] M. Portolan, "A Novel Test Generation and Application Flow for Functional Access to IEEE 1687 instruments", Proc European Test Symp. (ETS), pp. 1-6, 2016

[22] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995 ISBN 978-0-201-63361-0.