High Integrity Systems C.O.D.E.

Developing Certified Components for High Integrity Embedded Systems

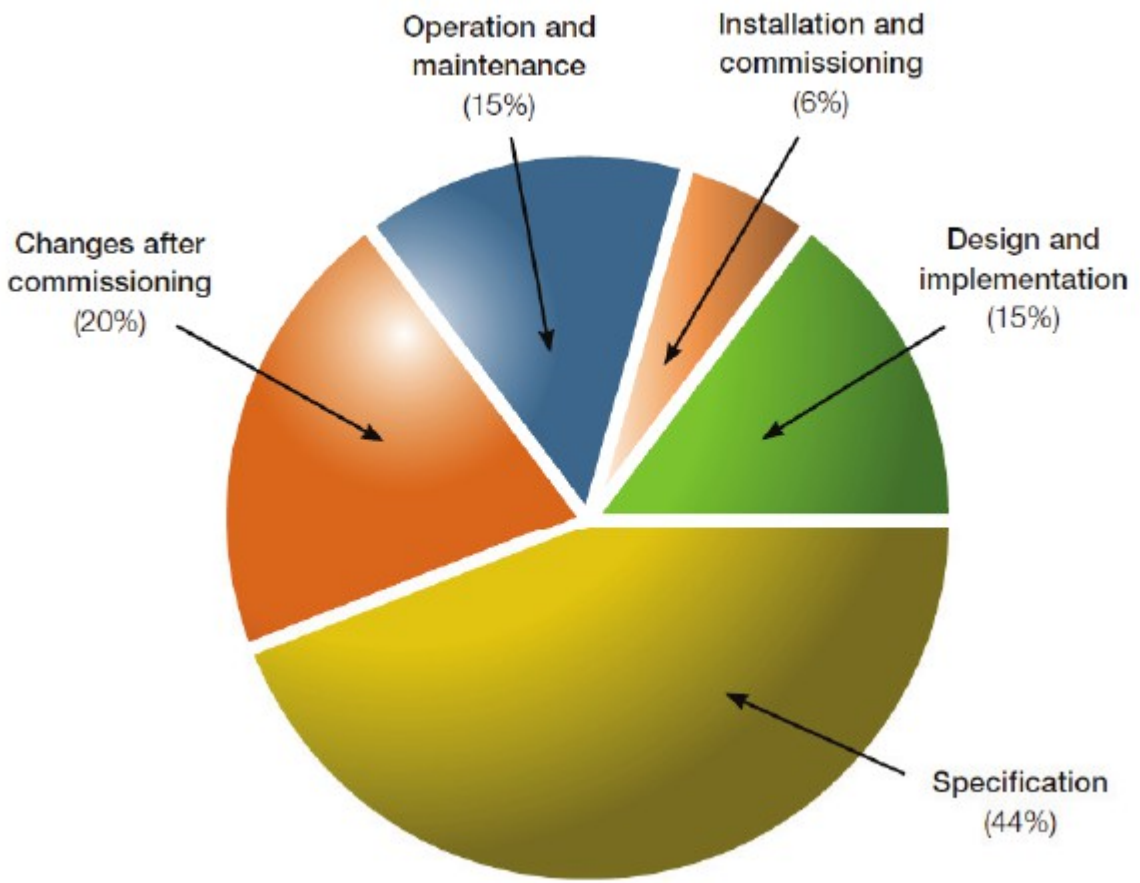
http://www.hidecs.co.uk/

Email: Paul_E.Bennett@topmail.co.uk

In a paper by Phil Koopman, titled “The Grand Challenge of Embedded System Dependability” he sets out that “Four significant challenges in embedded system dependability are:

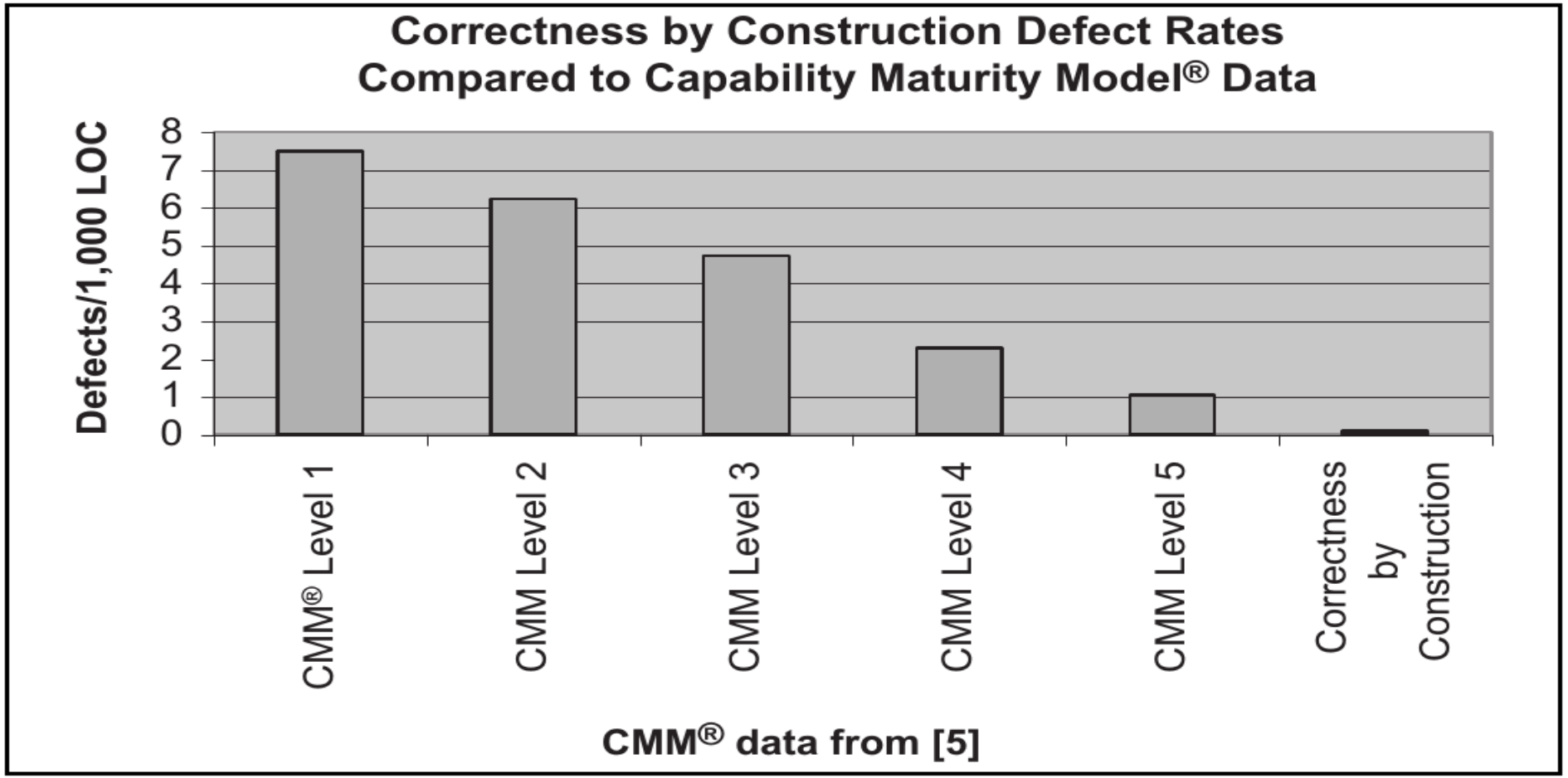
- embedded-specific security approaches,
- unifying security with safety,
- dealing with composable emergent properties,
- and enabling domain experts to use advanced dependability techniques.”

Where mistakes are made



(Out of control, 2nd edition 2003, Health & Safety Executive HSE – UK)

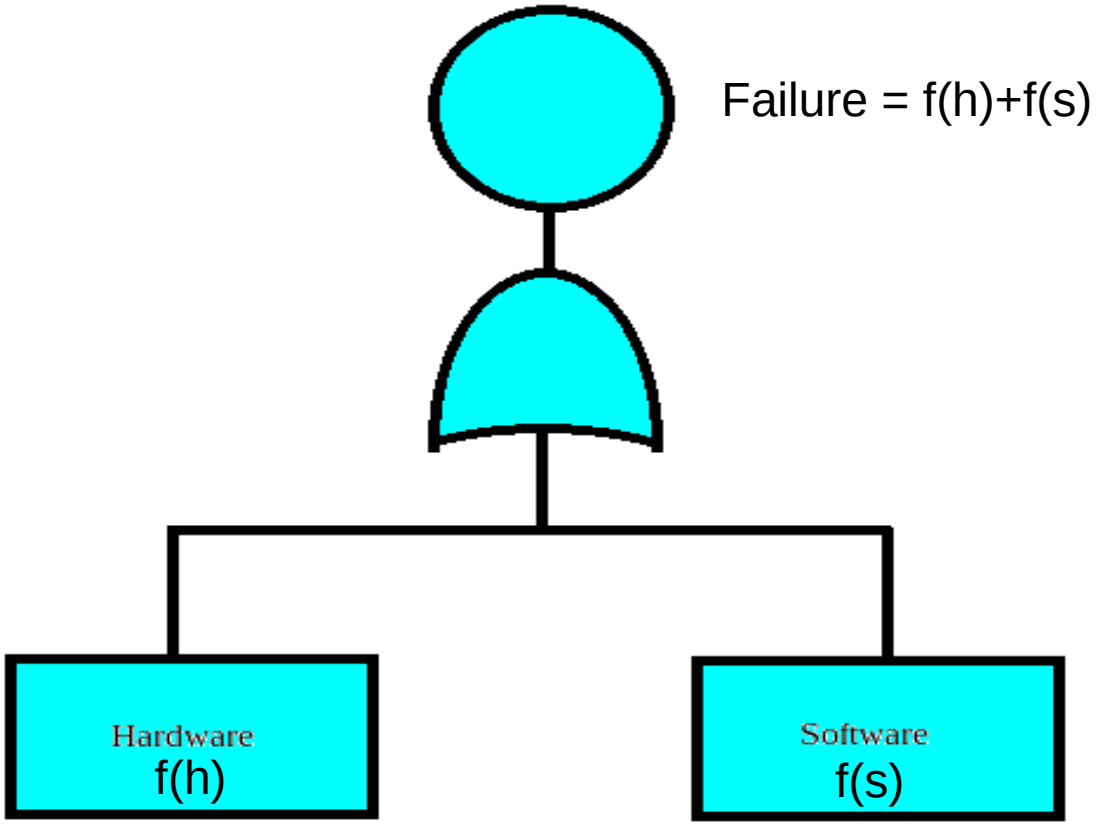
Capability & Correctness



Software Needs Hardware

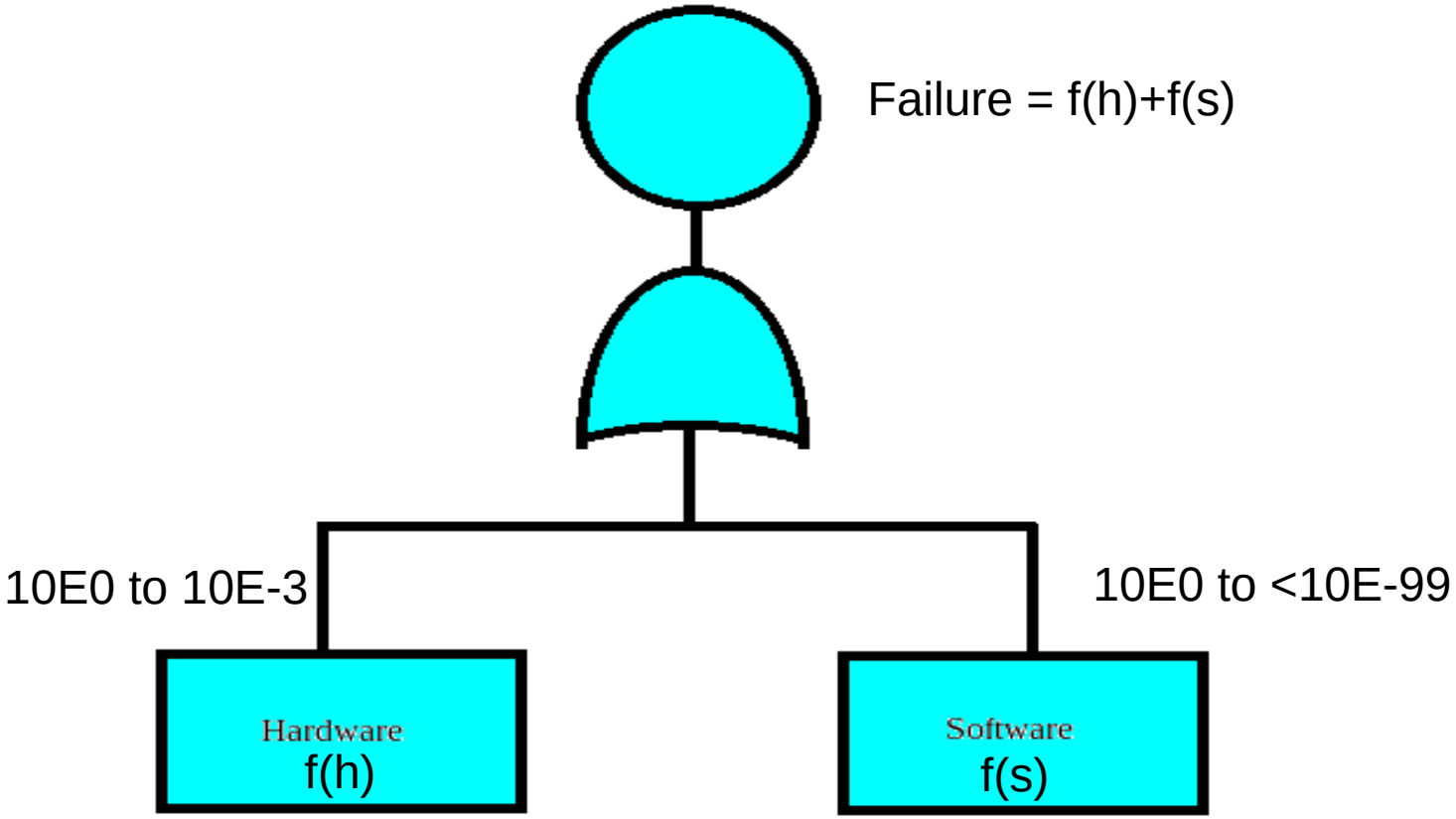
- Software does not operate without the support of the hardware on which it runs.
- Hardware can suffer random failures
  - Environmental Factors
  - Stress Induced Failures
  - Wear-Out Failures
- Software only suffers systematic failures

Software Needs Hardware



Software Needs Hardware

For a Single Channel of Control



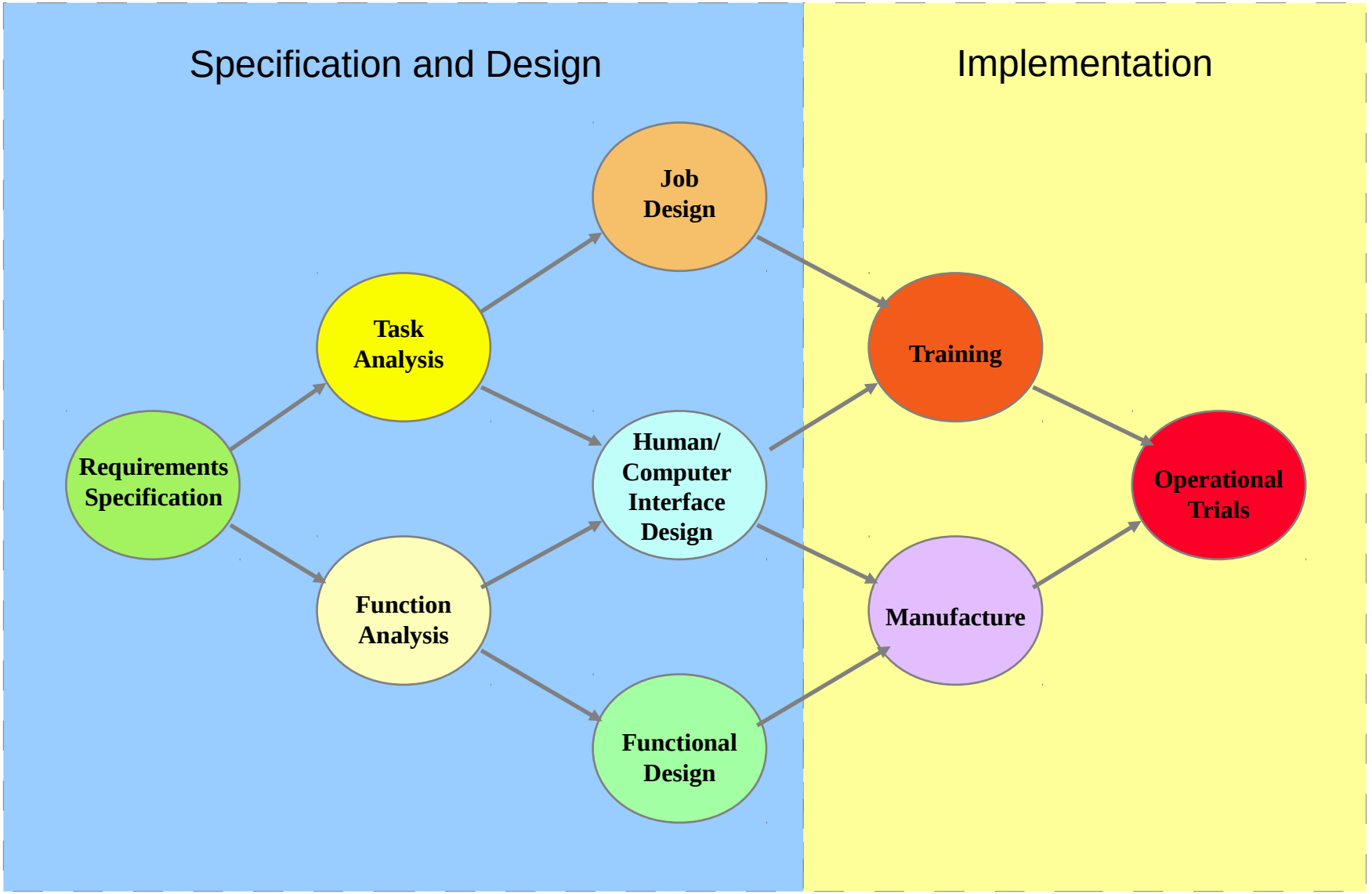
Design Integrity is vital

- Know the working environmnt
  - Design to operate within limitations of that enviornment
- Be clear about the Tasks to be performed
  - Task Description
  - Task Analysis
  - Hazop Study & Risk Assessment
- Revisit the early concept
  - It will not usually be right on the first pass so go back and look at what you can improve as early as possible.

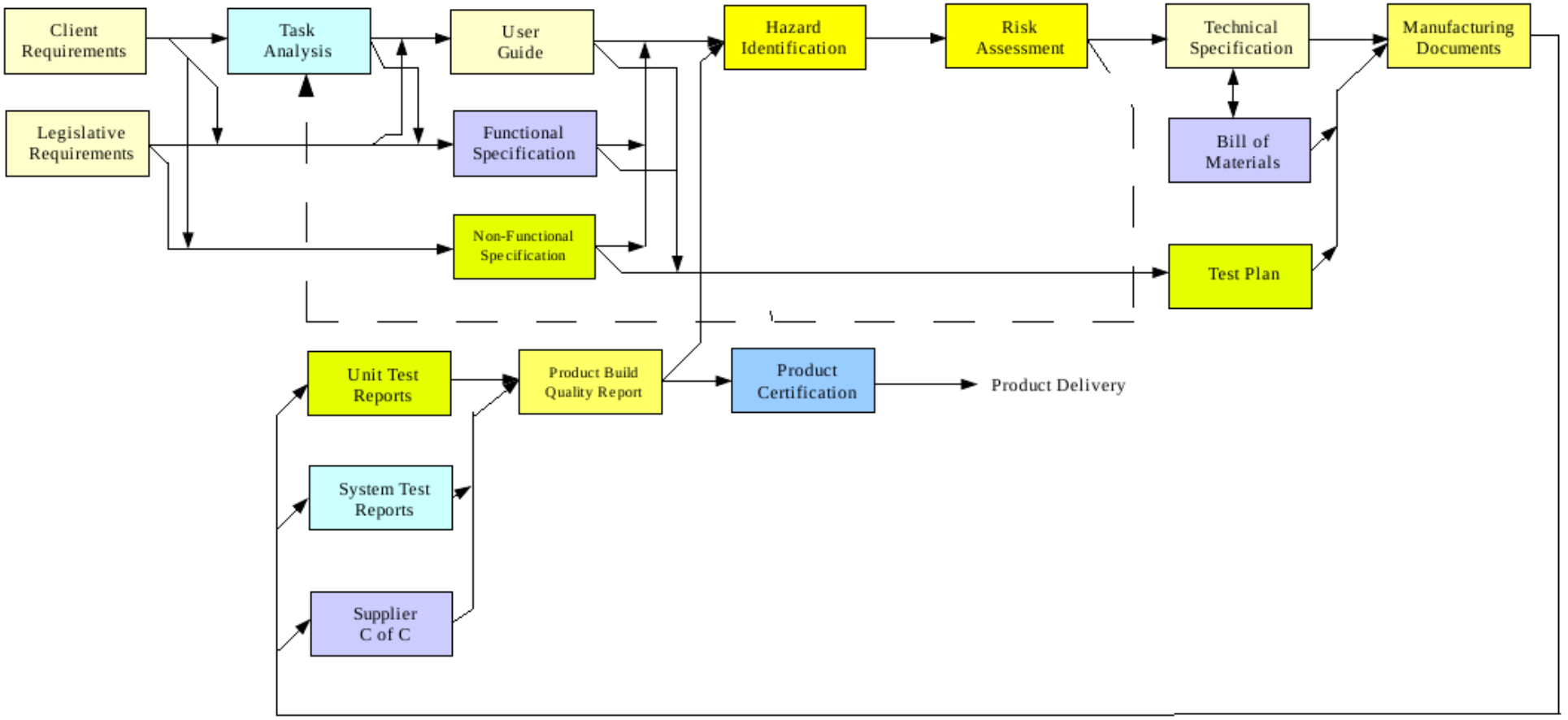
Have a Robust Process

- To develop a High Integrity System you need to be at CMM-3 or better from the start
- Your process needs to manage a multitude of versions and changes
- You need to keep the information and knowledge safe and secure
- You need to know that you and your clients are working to the same specification

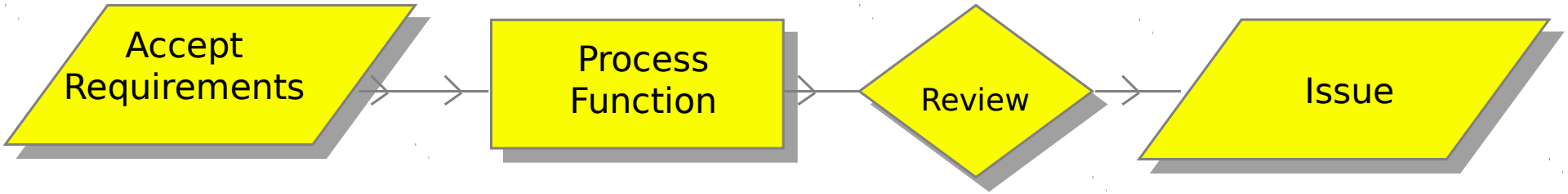
Specification Discovery



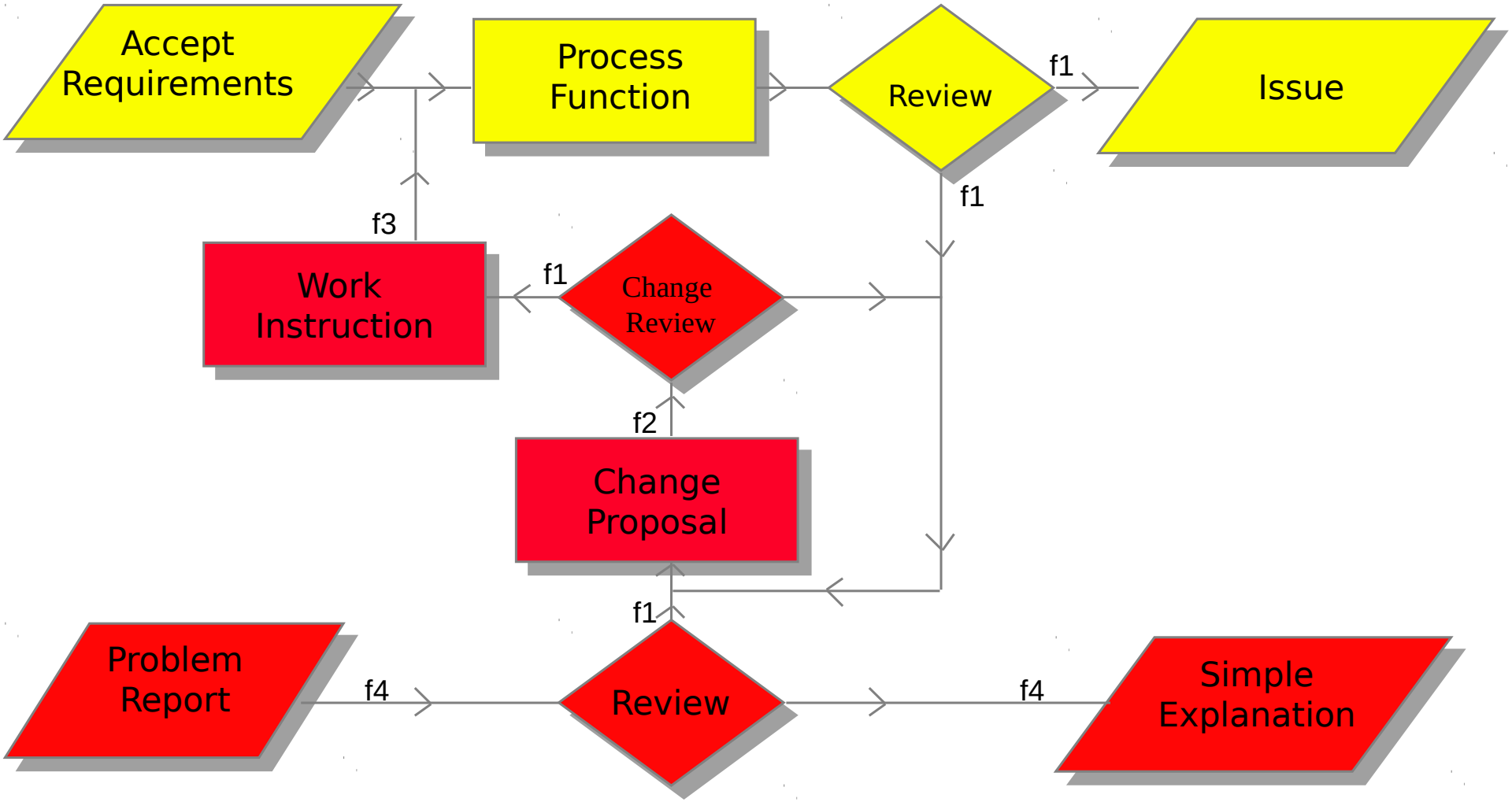
The Document Trail



An Engineering Process Model



An Engineering Process Model

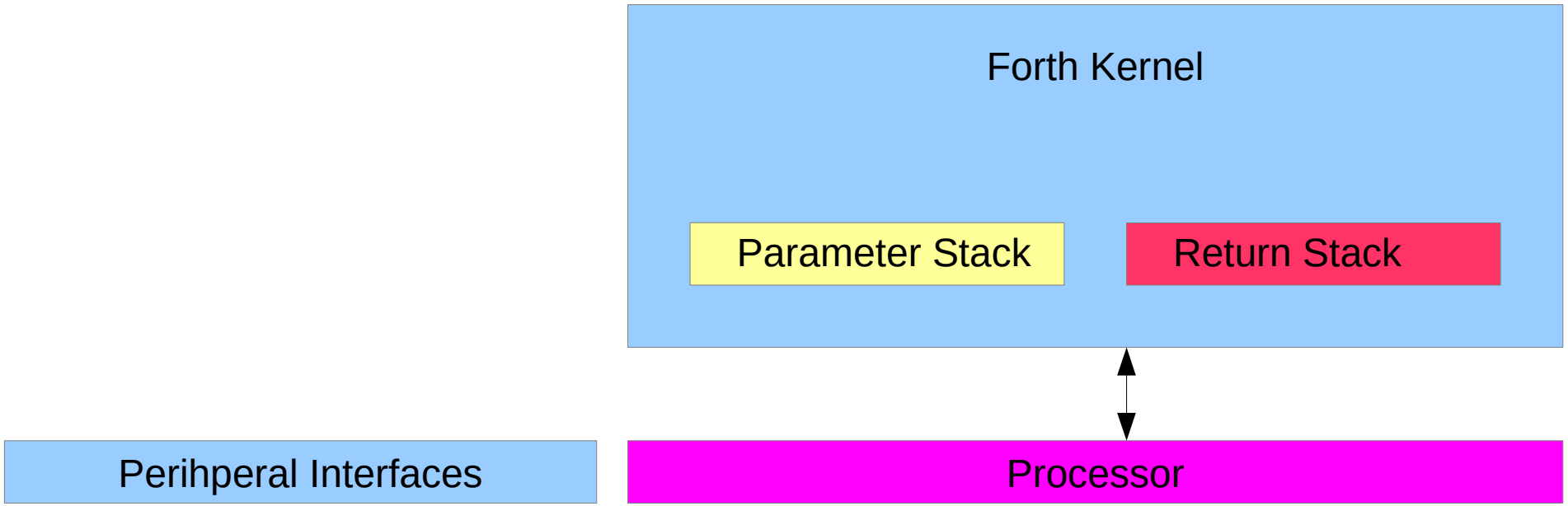


Why Forth?

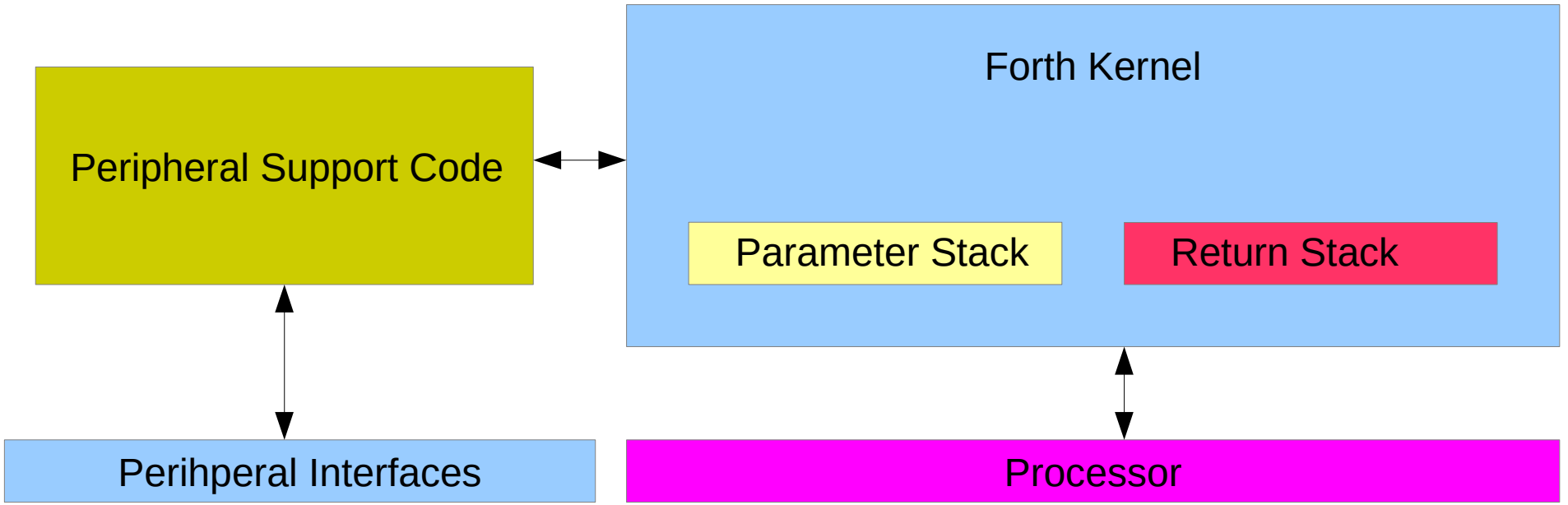
- Stable Virtual Machine (about 40 years)
- Extensible
- Supportive of Structured Programming
- Supportive of Component Oriented Approach
- Does not rely on sub-setting to be “Safe”
- Fully Certifiable

Why Forth?

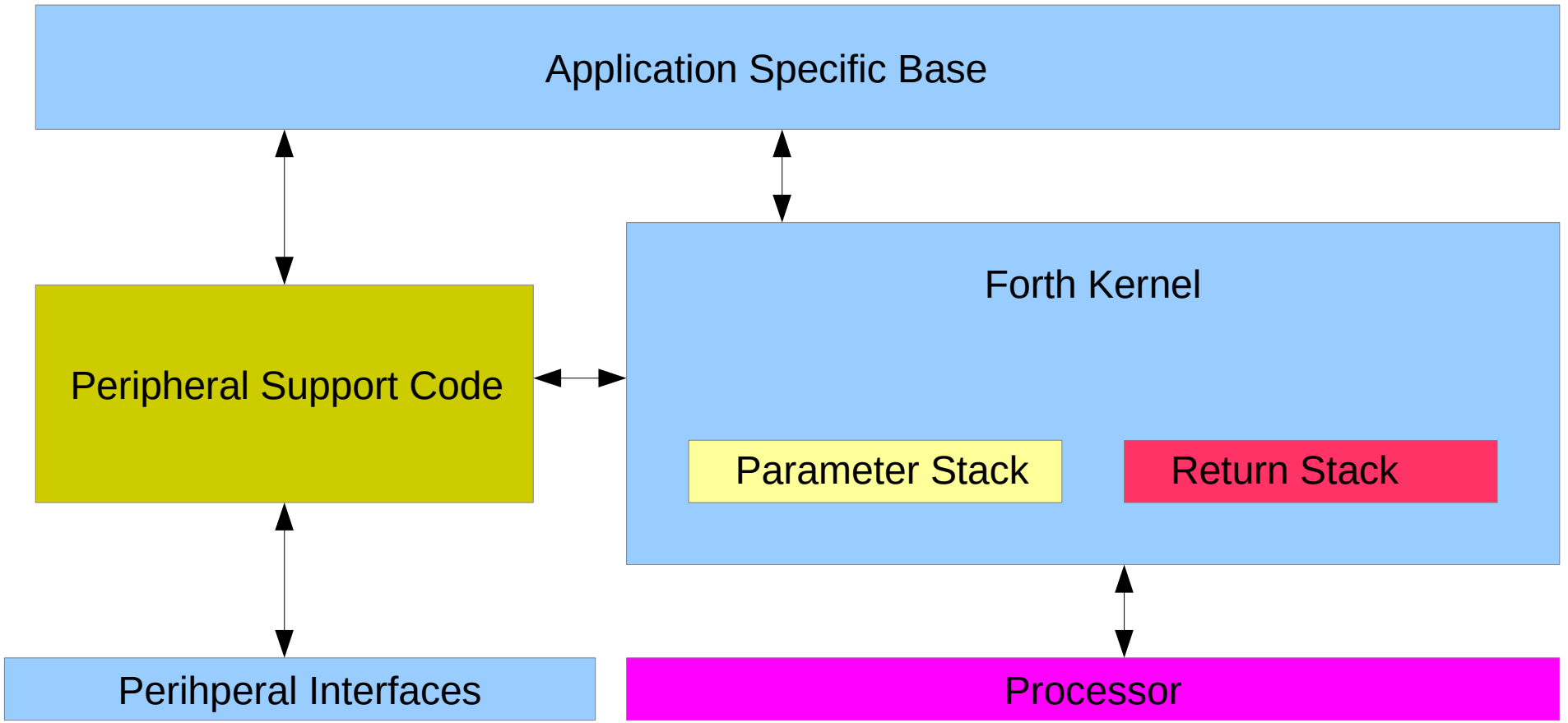
Why Forth?


Forth Kernel
Parameter Stack
Return Stack
Perihperal Interfaces
Processor

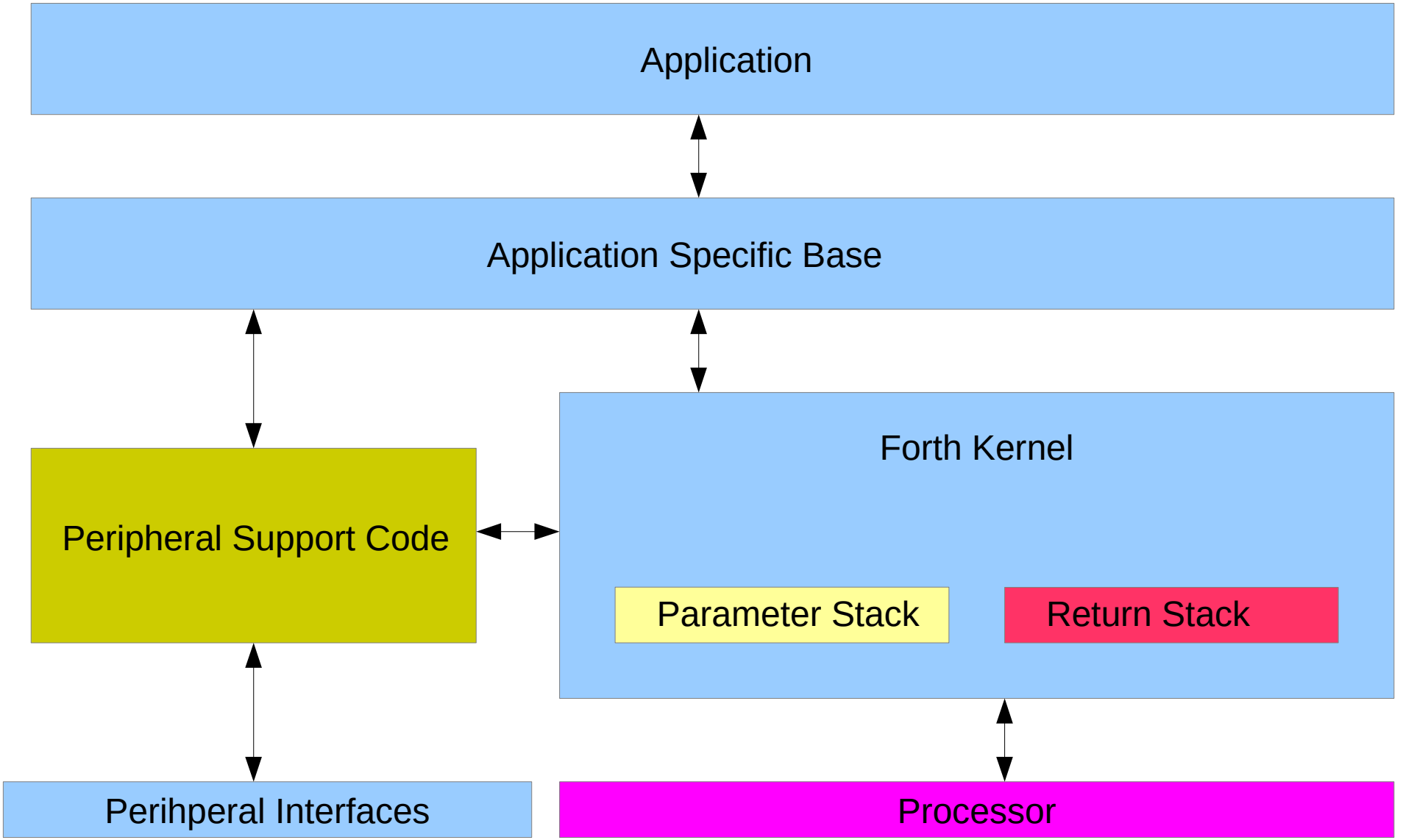
Why Forth?


Forth Kernel
Peripheral Support Code
Parameter Stack
Return Stack
Perihperal Interfaces
Processor

Why Forth?


Application Specific Base
Forth Kernel
Peripheral Support Code
Parameter Stack
Return Stack
Perihperal Interfaces
Processor

Why Forth?


Application
Application Specific Base
Forth Kernel
Peripheral Support Code
Parameter Stack
Return Stack
Perihperal Interfaces
Processor

Component Oriented

- All systems are constructed from components
- Components have Datasheets describing their attributes, functionaility and limitations.
- Components are complete in themselves
- Components can be certified for compliance with their specification.
- Non-compliance becomes obvious upon proper inspection.

Certifiying Software Components

- NPL have been running approximately 5,000 random C compilations per evening on a selection of C compilers. So far there have been no matches observed at the object code level.
- Forth already has at least two fully certified compiler implementations for High Integrity Applications.
- Choosing Forth made such certification effort much easier to complete.

Producing High Integrity Code

- Think of writing the comments first (use the comments as a statement of what you expect to be achieved).
- Review the comments to establish the state the true intent for the code you have yet to write.
- Write the code to meet the statement of requirements expressed by the comments.
- Statically Inspect the code to ensure implementation matches intent
- Perform a functional test of all logical paths in the code.
- Perform a limitations test (trying to make the code fail).
- 100% Path and Function Coverage is possible.

Code Inspection Sample

```
\ DSQRT                                        (c) PEB    28/10/05

: DSQRT ( ud -- u )
(G u is the nearest integer value of the square root of the    )
(   unsigned number ud. Results are rounded down.              )
   -1 >R
   BEGIN  R> 1+ >R
          R@ 2* 1+ S>D D-
          2DUP D0>= NOT
   UNTIL  2DROP R>
;
```

Code Inspection Sample

```
\ DSQRT                                    (c) PEB    28/10/05

: DSQRT ( ud -- u )
(G u is the nearest integer value of the square root of the    )
(   unsigned number ud. Results are rounded down.              )
   -1 >R
   BEGIN  R> 1+ >R
          R@ 2* 1+ S>D D-
          2DUP D0>= NOT
   UNTIL  2DROP R>
;
```

The above code fails certification as the word's glossary comment does not match the actual action of the code below it. The result returned is only valid if the input value is a positive signed number (31 bits instead of 32 - based on system with 16 bit width).

Code Inspection Sample

```
\ DSQRT                                        (c) PEB    28/10/05

: DSQRT ( +dn -- +n )
(G +n is the nearest positive integer value of the square root of )
(   the positive double length integer +dn. Results are rounded  )
(   down to the nearest positive integer.                         )
   -1 >R
   BEGIN  R> 1+ >R
          R@ 2* 1+ S>D D-
          2DUP D0>= NOT
   UNTIL  2DROP R>
;
```

After analysis, the code was deemed suitable for the application it was intended for but the glossary entry had to be re-worded to properly document its intention and limitations.

Summary

- You need a development process to at least CMM level 3 capability.
- Component Oriented Approaches keep the problems bounded.
- Forth is a very good Component Oriented Development Environment
- Forth code can be as certifiable as hardware.