The SEAL (Servo Algorithm Layer) System

Table of Contents

[2 Introduction 2](#_Toc208134031)

[2.1 Purpose 2](#_Toc208134032)

[2.2 Overview 3](#_Toc208134033)

[2.3 Required support Tools 3](#_Toc208134034)

[3 The drive 4](#_Toc208134035)

[3.1.1 Overview 4](#_Toc208134036)

[3.1.2 Controller Architecture 4](#_Toc208134037)

[3.1.3 SEAL Integration via IPC 4](#_Toc208134038)

[3.1.4 Role in the SEAL Eco System 4](#_Toc208134039)

[4 Modeling Environment 4](#_Toc208134040)

[4.1 SEAL 5](#_Toc208134041)

[4.1.1 Overview 5](#_Toc208134042)

[4.1.2 Use cases: Examples 5](#_Toc208134043)

[4.2 Drive Model 8](#_Toc208134044)

[4.2.1 Overview 8](#_Toc208134045)

[4.3 Scheduler (Harness) Model 8](#_Toc208134046)

[4.3.1 Overview 8](#_Toc208134047)

[4.4 Block diagram 9](#_Toc208134048)

[4.5 The SEAL ABI (Application Binary Interface) 10](#_Toc208134049)

[5 SEAL Export Function Module 11](#_Toc208134050)

[5.1 Overview 11](#_Toc208134051)

[5.2 Management, Scheduling, and Priorities 13](#_Toc208134052)

[5.3 Function Naming and Priorities 15](#_Toc208134053)

[5.4 **Limitations (SEAL Export Function Module)** 16](#_Toc208134054)

[5.4.1 Host / runtime constraints 16](#_Toc208134055)

[5.4.2 SEAL implementation-form constraints 17](#_Toc208134056)

[5.5 The drive control interface data structures 17](#_Toc208134057)

[5.5.1 Drive Interfacing Structures 17](#_Toc208134058)

[5.5.2 G\_DrvCommandBuf — Driver Command Buffer (per-axis) 18](#_Toc208134059)

[5.5.3 Drive Feedback — G\_DrvFeedbackBuf 22](#_Toc208134060)

[5.5.4 Setup Information (Report) — G\_SetupReportBuf 23](#_Toc208134061)

[5.5.5 Version Control — G\_SEALVerControl 25](#_Toc208134062)

[5.6 Communication interfaces 26](#_Toc208134063)

[5.6.1 Overview 26](#_Toc208134064)

[5.6.2 UART Usage 27](#_Toc208134065)

[5.6.3 CAN Usage 29](#_Toc208134066)

[5.7 Building an entry point function (function call subsystem) 31](#_Toc208134067)

[5.7.1 Example 1: Symulink block elements 32](#_Toc208134068)

[5.7.2 Example 2: Matlab code in a Matlab function block 32](#_Toc208134069)

[5.7.3 Example 3: Call for external service functions 33](#_Toc208134070)

[6 Preparing buses and signals 36](#_Toc208134071)

[6.1 Introduction 36](#_Toc208134072)

[6.2 Complex data type: the bus. 36](#_Toc208134073)

[6.3 Generate a parameter instance 37](#_Toc208134074)

[7 Appendix B – Glossary 38](#_Toc208134075)

[7.1 Topics 38](#_Toc208134076)

[7.2 Detail 38](#_Toc208134077)

[8 Appendix: Fixed tables and mappings 44](#_Toc208134078)

[9 Todo: 44](#_Toc208134079)

# Introduction

## Purpose

The purpose of the SEAL echo system is to give users a flexible and accessible way to implement their own control and communication algorithms on a robust servo drive platform. SEAL algorithms are developed in Simulink, where they can be fully simulated and tested in advance, as well as in parallel with deployment to the drive. This approach allows engineers to validate concepts, refine behavior, and ensure reliable performance before committing to hardware. By combining model-based design with seamless integration into the controller, SEAL bridges the gap between simulation and real-world execution, empowering users to bring proprietary functionality into demanding applications with confidence.

## Overview

The SEAL echo system is built around a dual-core controller that manages both drive operation and user-defined extensions. One core is dedicated to real-time control of the servo drivers, while the second core hosts the SEAL module. The SEAL module is automatically generated from Simulink models using MATLAB Coder and is scheduled by the controller alongside standard drive functions.

Within this framework, SEAL has full authority over drive modes and access to all relevant feedback signals. It can synthesize new drive commands, adapt control strategies in real time, and communicate with the external environment through standard communication lines. This enables users not only to implement advanced control algorithms, but also to integrate proprietary communication protocols and application-specific behaviors.

Development and deployment are supported and supervised by the **SolFlow environment** of GFT. SolFlow interrogates the drive model and revision and provides Simulink with the correct framework, including interface definitions and simulation models tailored to the drive under consideration. It also manages the build and download process, invoking the TI toolchain, and ensures that generated code is properly version-controlled.

To support validation, the system includes a scheduler model and drive model for Simulink, allowing complete closed-loop simulation of the controller and plant. Additionally, SEAL supports operation in Simulink “External mode,” enabling live parameter tuning and real-time data logging during hardware execution. In this way, the system ensures a smooth path from model development to hardware deployment, combining flexibility in design with reliability in operation.

## Required support Tools

The SEAL echo system relies on a set of third-party tools to provide the modeling, code generation, and deployment infrastructure:

* **MathWorks MATLAB + Simulink** – the core environment for model-based design.
* **Simulink Coder + Embedded Coder** – required for generating deployable **C code** from the SEAL function-export model.
* **Texas Instruments Code Composer Studio (CCS)** – the toolchain used to compile and link the generated C code into real-time loadable binaries for the dual-core controller.

In addition, the system is supported by **GFT SolFlow**, which is not a third-party product but an integral GFT tool. Beyond providing the correct SEAL frameworks for each drive and supervising build/download processes, SolFlow also serves as the platform for **drive commissioning, configuration, and tuning**.

# The drive

### Overview

GFT offers a family of drives capable of controlling one or more motors under the coordination of a **central controller**. The central controller handles external communication with process management systems and orchestrates the commands delivered to each of its slave drives. This architecture allows scalable deployment, from single-axis to multi-axis systems, under unified supervision.

### Controller Architecture

Each drive includes several processing cores, with different roles assigned to different aspects of system operation. Among them, one core is reserved exclusively for **SEAL**. This dedicated SEAL core is responsible for executing user-defined algorithms and is managed entirely by the SEAL scheduler. Its computational power is fully allocated to the execution of SEAL functions, ensuring isolation from the operational tasks of other cores.

### SEAL Integration via IPC

The SEAL core interacts with the **operational cores** of the drive through the **TI Inter-Processor Communication (IPC) interface**. This IPC channel forms the backbone of SEAL’s integration into the drive environment:

* **Feedback acquisition** – SEAL receives real-time data streams such as currents, positions, velocities, and status indicators.
* **Setup and configuration** – system initialization data and configuration parameters are provided to SEAL through IPC.
* **Command issuance** – SEAL delivers modal selections and reference commands to the operational cores, thereby controlling drive behavior.
* **Parameter exchange** – operational parameters and tuning variables can be transferred between SEAL and the drive cores, ensuring consistent operation across the system.

### Role in the SEAL Eco System

By dedicating an entire core to SEAL and linking it through the IPC mechanism to the operational cores, GFT drives provide a hardware foundation that guarantees both **determinism** and **flexibility**. Users gain a protected and powerful execution environment for their custom algorithms while retaining seamless access to all essential drive functionality.

# Modeling Environment

The modeling environment provides the user with a complete simulation and validation framework that mirrors the operation of the SEAL echo system. It consists of three main components:

1. **The SEAL model** – the user-defined control and communication logic, developed in Simulink and exported as a function-call subsystem.
2. **The drive model** – a representation of the operational core of the drive controller, responsible for executing drive dynamics and feedback behavior.
3. **The scheduler model** – the execution layer that manages SEAL interfaces and coordinates their operation, which can be inserted into Simulink simulations for closed-loop validation.

Together, these models enable seamless development, simulation, and testing of user algorithms prior to and in parallel with deployment to hardware.

## SEAL

### Overview

The SEAL is implemented as a **Simulink function-export model**, consisting of **several callable function-call subsystems**. Within this framework, the user defines proprietary control algorithms, communication strategies, and application logic. Through these subsystems, SEAL is granted full authority over the drive, with direct access to its commands, feedback, and status information. The Simulink model provides the development and validation environment, while the generated **C code**—with certain adaptations described in later sections—will later be deployed to the hardware.

The SEAL environment is supported by the following elements:

* **InitialSetup.m** – a MATLAB script that performs the initial setup of the environment, configuring paths and preparing Simulink for SEAL development.
* **SEALSystemTypes.m** – a definition file that specifies the drive-model-specific control and status structures forming the interface to the operational core.
* **User definition script** – a customizable file in which the user defines the data structures for their own algorithms. This file extends the framework with application-specific variables and logic. Working examples are provided as part of the package to guide users in creating their own definitions.

This layered organization makes SEAL both the container for user functionality and the structured interface to the drive controller, ensuring consistent simulation, validation, and eventual deployment under SolFlow supervision.

### Use cases: Examples

SEAL offers you great flexibility of system design.

Here are some examples.

|  |
| --- |
| The use case below depicts operational work options. They do not refer to the External-Mode connection to Simulink.  External-Mode connection to Simulink is use case agnostic, and done the same way for all the use cases described below |

#### User protocol over GFT-Servo

For this use case the SEAL masters communication lines to the host system.

It interprets the protocol, and issues corresponding commands to the drives chain below.

In this role the SEAL does no motion control: it relays the GFTServo motion commands as desired by the host, and sources feedback from the GFTServo for status reports.

The following block diagram shows this:



Figure : User protocol use case

The command to the servo drivers set can be either a continuous supply of position/speed/torque references or just targets to be profiled into by the drive – see the "Reference Mode" in the driver's command interface.

#### Implement your own controller

For this case the SEAL collects feedback from the servo drive and implements its own algorithms to refer GFTServo lower-level controls.

In this example the SEAL does no motion commanding or profiling: it relies on GFTServo to process motion commands by its native interfaces.



Figure : Feedback control use case

In the above example, the user implements own position control over the native speed controls of DFTServo.

#### Implement your own controller using external sensors by communication

For this case the SEAL collects inertial feedback from an external IMU, and relative feedback from the servo drive and implements its own algorithms to refer GFTServo lower-level controls.

In this example the SEAL does no motion commanding or profiling: it relies on GFTServo to process motion commands by its native interfaces.



Figure : Feedback control with external sensor use case

#### Implement your own controller using external sensors by extension board

For this case the SEAL collects feedback from an extension board, and optionally relative feedback from the servo drive. The information from the extension board comes over the FSI (Fast Serial Interface, down to 50usec period), making it hard real time.

The SEAL implements its own algorithms to refer GFTServo lower-level controls.

In this example the SEAL does no motion commanding or profiling: it relies on GFTServo to process motion commands by its native interfaces.



Figure : Feedback control with extension board use case

#### The fullest use: DIU (Do It Yourself) up to current control

For this case the SEAL collects feedback from an optional extension board, and optional over-the-communication IMU, and optionally relative feedback from the servo drive. The information from the extension board comes over the FSI (Fast Serial Interface, down to 50usec period), making it hard real time.

The SEAL takes over communication lines and interprets the protocol, and issues corresponding reference and profiling commands to the local control algorithms.

The result is a motor current reference to the GFTServo drive.

Figure : The full DIY (Do It Yourself) use case

## Drive Model

### Overview

The drive model is a **fixed-step Simulink model** that represents the operational core of the drive controller. Unlike SEAL, it does not generate deployable code. Instead, it provides the simulated counterpart of the hardware drive, enabling the SEAL module to be exercised and validated in a purely Simulink environment.

The drive model connects to SEAL through the **command and status buses** defined by the system interface. In simulation, it accepts drive commands synthesized by SEAL, applies the modeled drive dynamics, and produces feedback signals. These feedback signals are then routed back to SEAL via the bus interfaces, closing the control loop.

This setup allows the user to simulate realistic drive behavior, including interactions with a motor model, and to validate SEAL algorithms under conditions close to those encountered in deployment. In combination with the **Harness model**—which schedules the callable subsystems of SEAL—the drive model ensures that the entire system can be tested and refined within Simulink prior to hardware integration.

## Scheduler (Harness) Model

### Overview

The **Harness model** is the Simulink embodiment of the **“core-2” host scheduler**. Its purpose is to **emit the function calls** that operate the SEAL **function-export model** and to reproduce—in simulation—the timing, ordering, and event semantics that the controller firmware will enforce on hardware.

* **What it does**
* **Drives callable entry points.** Produces the function-call signals that invoke SEAL’s callable subsystems (e.g., initialization, cyclic control loops, communications/housekeeping).
* **Replicates timing & priorities.** Implements fixed-step scheduling with defined rates, priorities, and deterministic execution order; supports multi-rate operation and interrupt-like (asynchronous) events.
* **Coordinates interfaces.** Connects SEAL’s **command/status buses** to the **Drive model**, ensuring closed-loop exchange of commands and feedback during simulation/validation.
* **Models services the firmware provides.** Stubs or proxies scheduler services that SEAL may call (e.g., timers, timestamping, comms hooks, diagnostics), enabling realistic behavior in Simulink.
* **Handles lifecycle.** Orchestrates startup/shutdown: reset, configuration push, Init call(s), then steady-state cyclic calls at their specified rates.
* **Supports Simulink External mode.** When used with External mode, facilitates **parameter pushes** and **data logging** while SEAL runs in the loop.
* **What it is not**
* **No deployable code.** The Harness is a **simulation/validation artifact**; on target, its role is taken by the controller’s runtime scheduler.
* **No plant substitution.** It schedules SEAL and **coordinates** with the Drive model; it does not implement the drive dynamics itself.

This separation lets you validate algorithm behavior, timing assumptions, and interface correctness in Simulink—before (and in parallel with) generating the **C** code for deployment.

## Block diagram

## Direct control of a single (or two) axes

The following block diagram describes the SEAL ecosystem:

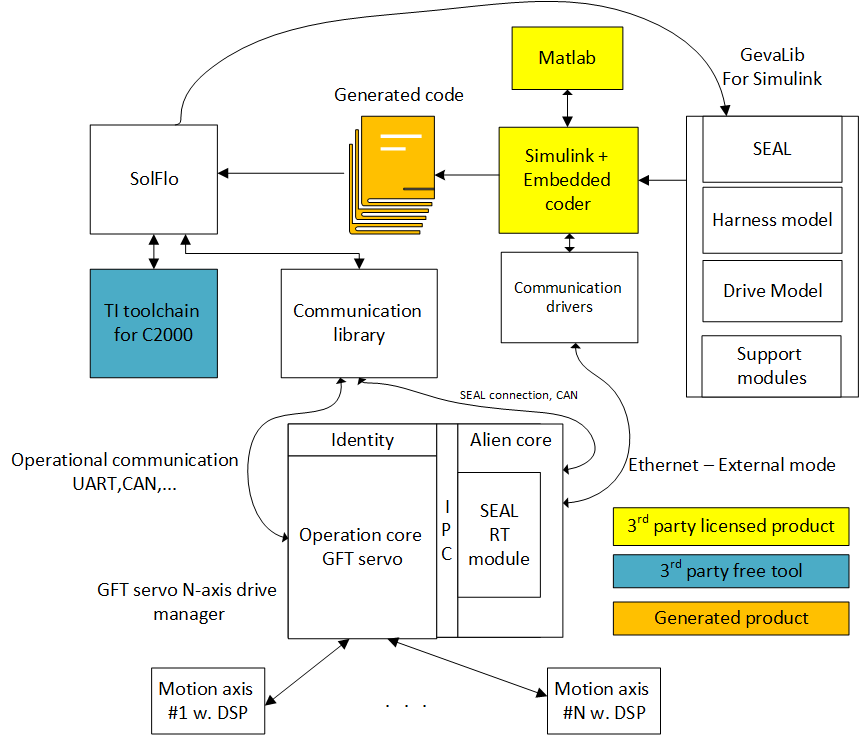


Figure : SEAL Ecosystem – direct axis control

The operational core the axes manager exposes its communication streams to the SEAL through the IPC, so that the SEAL can implement an operational command interpreter and respond communicated commands directly.

The exceptions are:

The exceptions are:

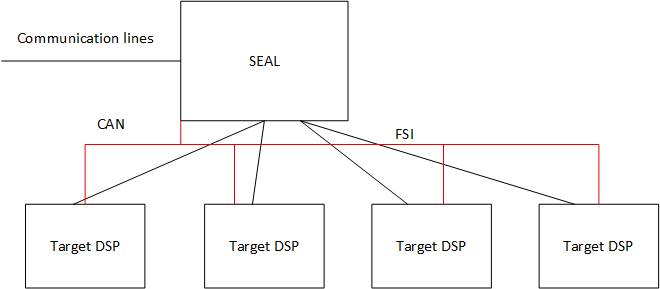
* + If the SEAL manages the UART, then the UART goes into the possession of CPU2 and the interface is not through the IPC
  + CPU2 has its own CAN controller, and SEAL accesses the CAN via the local controller.
  + The Ethernet of the external mode goes directly to Core2

Seal FW downloading shall be in direct speak with CPU2, so that the software loading process of CPU1 shall remain unchanged.

## SEAL Multiple-axis scenario

In this scenario the SEAL core is in a separate CPU that is communicating and coordinating servo axes via external communication lines – here FSI.

The SEAL core has very similar interfaces, only that instead of using the IPC the interface is via FSI lines.



## The SEAL ABI (Application Binary Interface)

**What it is:** The binary-level contract that lets independently built components interoperate without recompilation. It fixes **call/return rules**, **data layouts**, and **type sizes/alignments** so both sides read/write the same bytes the same way.

**In SEAL:** The ABI is the fixed, SolFlow-defined interface between SEAL and the host/drive. It includes:

* **Drive interface buses & arrays**: G\_DrvCommandBuf, G\_DrvFeedbackBuf, G\_SetupReportBuf, G\_SEALVerControl (field names, order, types, dimensions, per-axis array length).
* **Channel/queue contracts** (for comms): channel enumeration, max frame sizes, queue depths, and record layouts.
* **Type system & packing**: exact widths (e.g., int16, uint32, single, double) and struct alignment/padding as generated.
* **Naming-based scheduling surface**: port/class naming that the host parses to derive class/period/sequence (not a C ABI per se, but part of the cross-component contract).

**Versioning & checks:**

* G\_SEALVerControl.Version/SubVersion/UserData are used to **verify ABI compatibility** at startup (Setup).
* On mismatch, the host **refuses to arm motion** (build may compile, but code is not accepted for target loading).

**What counts as ABI-breaking vs. compatible:**

* **Breaking (not allowed without coordinated updates):**  
  – Reorder/add/remove fields in any interface struct  
  – Change field **types/widths** (e.g., int16 → int32) or array lengths (#axes)  
  – Change queue record layouts or channel enumeration values

**Policy:** SolFlow defines and owns the ABI. Users **must not modify** layouts or storage classes. C only; **no dynamic memory** and **no C++**, to preserve determinism and avoid hidden allocations.

# SEAL Export Function Module

## Overview

The SEAL is realized as an **Export Function Module**, composed of three main elements:

* **Function Call Subsystems** – Containers for the user application code.
* **Data Stores** – Global data variables serving as the shared memory space.
* **Drive Simulation I/O Ports** – Simulation-only interfaces connecting the data stores to the drive model.
* **Function Call Subsystems**

The SEAL scheduler on Core 2 invokes subsystems according to their class and configured priority. Five classes of subsystems are provided:

* **Setup** – Initialization and one-time configuration routines.
* **ISR** – Time-critical subsystems, scheduled with higher priority to approximate a periodic rate. While named *ISR*, they are not hardware interrupts; Simulink provides no inherent mechanism for thread safety.
* **IdleLoop** – Background subsystems executed as infinite loop when no higher-priority tasks are pending.
* **Exception** – Subsystems reserved for handling abnormal or fault conditions.
* **Abort** – Emergency routines invoked to halt or reconfigure system activity.

Core 2 can run up to **eight instances of each subsystem class**, giving flexibility in structuring the application.

* **Data Stores**

All subsystems operate over **global data variables**, implemented as data stores. These serve as the communication medium between subsystems and are common to the whole module.

* Some structures are **fixed by SolFlow**, defining the drive interface (commands, feedbacks, parameters).
* Others are **user-defined**, allowing the application to maintain its own shared state.
* **Drive Simulation I/O Ports**

The **I/O ports** are special blocks that connect the drive simulation with the SEAL data stores. They serve to:

* Deliver drive feedback values into the SEAL global data space.
* Accept drive command and parameter values generated by the SEAL routines.

These ports **are not intended for user modification**. They exist only in the *simulation model* to emulate the hosting scheduler (Core 2) and the drive environment. They will **not be coded to the target**.



Figure : The SEAL export function module. Not shown: Exception and abort functions

## Management, Scheduling, and Priorities

The SEAL module is orchestrated by the **scheduler of Core 2**, which controls *when* and *in what order* the different function call subsystems are invoked. Scheduling is cooperative: subsystems are invoked as function calls, run to completion, and then return control to the scheduler. No thread preemption or locking is provided by Simulink; thus, **data integrity depends entirely on design discipline**.

* **Scheduler Data Exchange**

Before invoking each subsystem call, the hosting scheduler automatically **refreshes the SEAL data stores** with:

* **Feedback values** from the drive.
* **Setup reports** representing the drive’s current status.

After the subsystem completes, the scheduler automatically **updates the drive commands** from the data stores, reflecting any modifications made by the subsystem logic.

This mechanism is **transparent to the modeler**: the user does not need to code or manage data transfers explicitly.

* **Priorities and Behavior by Class**
* **Setup** – Executed once at initialization to prepare system state. Only a **single Setup function** is permitted. Unlike other subsystems, Setup **hangs until all motors are confirmed off**, ensuring a controlled startup sequence. Setup is also the only entry point for system reset after an **Exception** or **Abort** condition.
* **ISR** – Scheduled with the **highest runtime priority** to approximate a periodic rate. They represent time-critical logic, but remain function calls (not hardware interrupts).
* **IdleLoop** – Scheduled as a **continuous infinite loop**, executing one IdleLoop function after another whenever no higher-priority tasks are pending.
  + Up to eight IdleLoop functions can be defined, not only for separation of functionality but also to **break down the idle slot**. This prevents a single long idle function from blocking ISR execution and ensures responsiveness.
* **Exception** – Triggered either by the user or automatically. Automatic exceptions occur if:
  + The drive is switched off by **electromechanical protection conditions**, such as overspeed, excessive tracking error, overcurrent, or overheating.
  + An ISR exceeds its permitted timing deviation.
  + Other system-defined abnormal conditions arise.  
    During an exception:
  + The **IdleLoop remains active**.
  + **ISRs are stopped**, halting periodic commands.
  + **Drive commands are ignored**, though the drive continues to report updated **feedbacks** and **setup data** into the data stores.
* **Abort** – Automatically invoked following every exception. Once an exception routine completes, **all Abort functions are executed** in sequence, ensuring the drive is safely disengaged or reconfigured. An Abort function may also set the **“Relinquish Control” flag**, which restores control of the drive to its standard interfaces, as if SEAL were not in possession.
* **Instance Management**
* **Setup** is unique: only **one instance** may exist.
* Other subsystem classes (ISR, IdleLoop, Exception, Abort) may each have up to **eight instances**, allowing finer-grained design flexibility.
* **Global Data Access**

All subsystems access the shared **data stores**. Because execution is non-preemptive, priority mainly determines which subsystem’s updates persist. Designers must consider:

* Which variables are sensitive to timing deviations.
* Where **latched copies** or staging buffers are required to preserve data consistency.

## Function Naming and Priorities

The mapping between Simulink and SEAL function execution is determined by the **names of the function call ports**. These port names are transferred verbatim into the generated code, and also dictate both the **function class** and the **scheduling order**.

* **Naming Scheme and Class Determination**
* A function call subsystem inherits its name directly from its **function call port**.
* The same name is assigned to the **handler function** in the generated code.
  + Example: A port named ISR100uControl will produce a generated function ISR100uControl().
* The **class** of each function is determined by the preamble of the name.
  + Example: ISR100uControl → ISR class.
  + Example: Isr50uProfiler → ISR class (class detection is **case-insensitive**).
* Valid class prefixes are: **Setup, ISR, IdleLoop, Exception, Abort**.
* **Sequencing by Port Numbers**

For all classes **except ISR and Setup**, execution order is determined by the **port numbers**:

* Example: Ports IdleLoop1, IdleLoop2, and IdleLoopLast assigned port numbers 7, 1, and 10 will be executed in the order:  
  IdleLoop2 → IdleLoop1 → IdleLoopLast.
* **ISR Naming and Priority Rules**

ISRs follow a stricter naming convention:

* Must be named ISR<number>u<anything>.
* The <number> specifies the **expected period in microseconds**. If the period is not an integer multiple of 1usec, the fraction may be placed following '\_' or 'p'. Remember, however, that ISR period timesmust be integer multiples of the drive basic Ts. Hint: SolFlow exposes the basic sampling time as the Matlab variable SealProjectDescriptor.BaseTs
  + Example: ISR100uControl → nominal rate of 100 μs.
  + Example: ISR100p5uControl → nominal rate of 100.5 μs.
  + Example: ISR100\_5uControl → nominal rate of 100.5 μs.
* ISR priorities are assigned as follows:
  + **Highest priority** goes to the ISR with the **shortest period**.
  + If multiple ISRs have the same period, their relative priority is resolved by **port numbers**.
* **Post-Processing and Validation**

The code generation step includes a **post-processor** that validates function naming and classification:

* If a function name cannot be correctly sorted into its class or timing slot, the model will still compile.
* However, a **warning is issued**, and the generated code will **not be accepted for loading** to the target.

This scheme is necessary because **Simulink does not export scheduling data** unless a main() is generated. SEAL, being hosted by an external scheduler, relies instead on the **naming convention** for class and timing.

* **Harness Model Considerations**

When testing SEAL with a harness model, special care must be taken:

1. **IdleLoop Frequency** – The frequency of IdleLoop scanning is determined by the actual **target processor load and speed** (e.g., a C29xx will run IdleLoop much faster than a C28xx). This cannot be reproduced faithfully in simulation.
   * For better realism, the modeler may read and import **actual execution times** from the target once the system is running.
2. **Function Call Generator Periods** – In the harness model, the periods of the **function call generator** must be matched **manually** to the ISR periods implied by the port names. Otherwise, harness simulation will not reflect actual scheduling.

## **Limitations (SEAL Export Function Module)**

**Scope.** This section lists the limitations of the **SEAL implementation form**. These are **in addition to** general code-generation constraints (e.g., **no solver/continuous-time semantics**, **strict typing**, **no dynamic/variable sizing**), which apply broadly to all embedded cosder artifacts and are not SEAL-specific.

### Host / runtime constraints

* **No dynamic memory (no heap).** Allocation from the heap is **disallowed** for safety and repeatability; we cannot risk memory leaks, fragmentation, or non-deterministic garbage collection.
* **ISR period times** must be integer multiples of the drive basic Ts. Hint: The SolFlow exposes this as the Matlab variable SealProjectDescriptor.BaseTs. Note: This is not a physical limitation but intentional. We want the seal action remain synchronized to the actions of the main CPU. Otherwise timing mismatches will cause hard-to-explain subharmonics phenomena.
* **C++ excluded.** For the same reasons, **C++ is not permitted**: common patterns (including library calls, operators, and hidden temporaries) may allocate/deallocate without the programmer’s awareness, violating the "no heap" constraint. The implementation is **C only**, with static/fixed storage.

### SEAL implementation-form constraints

* **Scheduling inferred from port names and numbers.**  
  Simulink does not emit scheduling metadata without a generated main(). Therefore the scheduler derives **class**, **nominal period**, and **sequence** solely from **function-call port names** and their **port numbers**. If a function cannot be sorted, the model compiles, a **warning** is issued, and the generated code is **rejected for target loading**.
* **All discrete blocks must use *Inherited sample time* (Ts = –1).**  
  Entry call functions are invoked by an external scheduler and are **unaware of their own call rate** inside Simulink; blocks requiring an explicit Ts are incompatible here.
* **Discrete Integrator block is unsupported.**  
  The Simulink discrete integrator implements **Ts·Ki / (z–1)** and requires a known **Ts**. Because Ts must be *inherited* in this implementation form, the block **cannot be made aware of the actual period** and must not be used in SEAL entry functions. You can use an integrator block from the SolFlow library instead.
* **Drive simulation I/O ports are fixed by SolFlow.**  
  These ports emulate Core-2 hosting in simulation, are **not code-generated**, and **must not be modified** by users.
* **Fixed interface ABI (SolFlow).**  
  Drive interface structures (Commands, Feedback, Setup Report, Version) are **defined by SolFlow**, **host-allocated**, and **must not be altered**. Defaults are host-populated: **motor off**, **etc**.

## The drive control interface data structures

### Drive Interfacing Structures

SEAL exchanges control and feedback data with the drive via **fixed interface structures** defined and provisioned by **SolFlow** for the specific **drive model/identity**. These structures are carried in data stores and come in four families:

1. **Driver Command** – SEAL’s requested actions.
2. **Feedback Information** – live measurements from the drive.
3. **Setup Information (Report)** – state, capabilities, configuration.
4. **Version Control** – ABI/signature and compatibility markers.
5. **Authoritative definitions (no user edits)**

* **All bus/type definitions originate from SolFlow** and are bound to the drive identity.
* **Users must not alter** bus layouts, names, dimensions, or storage classes.
* The **array length** (multi-axis) is set by the drive identity and provided by the host.

1. **Allocation & lifecycle**

* **Allocation is performed by the host** (not by the user model).
* **Default values are populated in advance** by the host before SEAL functions run.
* In simulation, these same definitions are provided; users still **must not** attempt to recreate/override them.
* **Documentation note"**

**The drive interface structures, like the G\_DrvCommandBuf schema** below, are generated by SolFlow for each drive identity. The table above documents the **baseline** fields present across models and the semantics of optional groups. Your build ships a header (e.g., SealInterface.h) produced by SolFlow that is the **single source of truth** for field order, types, and enums. Applications **must not** modify the field layout and must adhere to the mode/reference pairing rules.

### G\_DrvCommandBuf — Driver Command Buffer (per-axis)

* **Ownership:** Allocated by host, schema generated by **SolFlow** for the specific drive model/identity.
* **Lifecycle defaults (host-side init):**
  + Motor **disabled**
  + **Highest loop** configured (per drive) is selected
  + All numeric fields **zero**, except **position reference initialized to actual feedback position**
* **Concurrency:** Produced by the host, consumed by the SEAL scheduler over TI IPC.
* **Multiplicity:** G\_DrvCommandBuf[NumAxes] (one element per controlled axis).
* **Write cadence:** Host updates once per control cycle (or per your host scheduler); SEAL samples at its scheduler boundaries.
* **Do not change schema** in code—use the SolFlow package that defines the bus.
* **1) Baseline fields (stable across models)**

| **Field** | **Type** | **Units / Domain** | **Direction** | **Default** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| AxisId | uint16 | index (0…NumAxes-1) | Host→Drive | model | Redundant with array index, but useful for sanity. |
| Enable | boolean | on/off | Host→Drive | false | Motion/drive power enable interlock. |
| FaultReset | boolean | pulse | Host→Drive | false | Write true for one cycle to request fault clear. |
| Mode | uint16 | enum | Host→Drive | **Highest loop** | Control mode selection (e.g., Current/Velocity/Position). Values come from the model’s SolFlow enum. |
| PositionCommand | double | mechanical units (e.g., rev, rad, mm) | Host→Drive | Feedback.Position | Absolute position reference in **position mode**. |
| SpeedCommand | double | mech units/s | Host→Drive | 0 | Velocity reference in **speed mode**. |
| CurrentCommand | double | A (phase/torque-producing) | Host→Drive | 0 | If FOC: this is typically **Iq** request when in current/torque mode. |
| FeedForwardPosition | double | same as PositionCommand | Host→Drive | 0 | Optional additive term used by the position loop. |
| FeedForwardVelocity | double | same as SpeedCommand | Host→Drive | 0 | Optional additive term for velocity loop. |
| FeedForwardTorque | double | N·m (or A) | Host→Drive | 0 | Optional additive torque/Iq feed-forward. |
| JerkLimit | double | units/s³ | Host→Drive | model | Max jerk for profiler (if enabled). |
| AccelLimit | double | units/s² | Host→Drive | model | Max accel for profiler. |
| VelocityLimit | double | units/s | Host→Drive | model | Max velocity for profiler & safety. |
| TorqueCurrentLimit | double | N·m or A | Host→Drive | model | Soft torque/current clamp. |
| SoftLimitsEnable | boolean | on/off | Host→Drive | model | Enables enforcement of soft travel limits. |
| SoftLimitMin | double | position units | Host→Drive | model | Lower travel bound (abs coordinates). |
| SoftLimitMax | double | position units | Host→Drive | model | Upper travel bound (abs coordinates). |
| Halt | boolean | on/off | Host→Drive | false | If true, profiler ramps to stop with configured decel. |
| EmergencyStop | boolean | on/off | Host→Drive | false | Hard stop; implementation model-specific. |
| CommandSeq | uint32 | monotonically increasing | Host→Drive | 0 | Lets drive detect dropped/out-of-order writes. |
| CommandTimestamp | uint64 | host ticks or µs | Host→Drive | 0 | Host time of issue (sync/diagnostics). |
| Checksum | uint32 | CRC32 | Host→Drive | 0 | Optional integrity guard; model decides enforcement. |
| Reserved[] | — | padding/forward-compat | — | 0 | For alignment and future extensions. |

**Notes on ‘Mode’**  
Typical enum (actual values come from the SolFlow model):

* 0: Torque/Current
* 1: Velocity
* 2: Position
* 3+: Model-specific (e.g., Homing, Indexing, Gearing/Following, Cyclic Synchronous modes)
* **2) Optional motion-command group (profiler & targets)**

These appear when the model includes an internal profiler or trajectory block. When present, they override raw PositionCommand/SpeedCommand as the *target inputs*.

| **Field** | **Type** | **Units** | **Direction** | **Default** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| MoveProfiled | boolean | — | Host→Drive | false | If true, use profiled move block. |
| TargetPosition | double | pos units | Host→Drive | Feedback.Position | Destination for profiled move. |
| TargetVelocity | double | units/s | Host→Drive | 0 | For velocity holds or blended moves. |
| TargetAccel | double | units/s² | Host→Drive | model | Per-move override (else use AccelLimit). |
| TargetDecel | double | units/s² | Host→Drive | model | Per-move override (else use limit). |
| TargetJerk | double | units/s³ | Host→Drive | model | Per-move override (else use limit). |
| ProfileAbort | boolean | pulse | Host→Drive | false | Abort current profile (ramps per decel). |

* **3) Optional homing & reference group (if enabled in model)**

| **Field** | **Type** | **Units** | **Direction** | **Default** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| HomeRequest | boolean | pulse | Host→Drive | false | Start homing sequence. |
| HomeMethod | uint16 | enum | Host→Drive | model | Per model (switch to index, etc.). |
| HomeVelocity | double | units/s | Host→Drive | model | Search speed. |
| HomeOffset | double | pos units | Host→Drive | 0 | Applied after home latch. |

* **4) Optional electronic gearing / following (model-dependent)**

| **Field** | **Type** | **Units** | **Direction** | **Default** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| FollowerEnable | boolean | — | Host→Drive | false | Enable electronic gearing. |
| MasterSource | uint16 | enum | Host→Drive | model | What the axis follows (encoder, bus, virtual). |
| GearNumerator | int32 | — | Host→Drive | 1 | Ratio numerator. |
| GearDenominator | int32 | — | Host→Drive | 1 | Ratio denominator. |
| GearPhaseOffset | double | pos units | Host→Drive | 0 | Phase shift. |

* **5) Optional FOC current-loop details (if model exposes them)**

Some models expose split current commands; others only expose torque/Iq.

| **Field** | **Type** | **Units** | **Direction** | **Default** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| IqCommand | double | A | Host→Drive | 0 | Torque-producing current. |
| IdCommand | double | A | Host→Drive | 0 | Flux current (rarely nonzero for PM motors). |

* **6) Command validity & masks**

| **Field** | **Type** | **Direction** | **Default** | **Notes** |
| --- | --- | --- | --- | --- |
| ValidMask | uint32 | Host→Drive | 0 | Bitmask indicating which fields to consume this cycle (if model enforces). |
| AppliedMask | uint32 | Drive→Host (echo) | — | Last applied fields (if the model echoes it in a feedback buffer). |

**Alignment & ABI**  
SolFlow emits the bus with explicit ordering and padding. Treat the layout as **ABI-stable for the model version**. Do not assume natural packing—use the Simulink bus accessors or code-gen’d headers.

* **Usage contract (what the application can safely do)**
* **Set enable / mode** and write the **one active reference** that corresponds to the selected mode (PositionCommand *or* SpeedCommand *or* CurrentCommand), plus any optional feed-forward.
* Respect **limits** by writing per-axis clamps (VelocityLimit, AccelLimit, TorqueCurrentLimit) or leave them at model defaults.
* For profiled motion, drive MoveProfiled = true and write the Target\* fields.
* Never add/remove fields; if a field is absent for your model, it’s not supported there.
* **Example (illustrative) C struct**

⚠️ **Illustrative only** — exact types/order come from your SolFlow-generated header.

typedef struct {

uint16\_t AxisId;

bool Enable;

bool FaultReset;

uint16\_t Mode;

double PositionCommand;

double SpeedCommand;

double CurrentCommand;

double FeedForwardPosition;

double FeedForwardVelocity;

double FeedForwardTorque;

double JerkLimit, AccelLimit, VelocityLimit, TorqueCurrentLimit;

bool SoftLimitsEnable;

double SoftLimitMin, SoftLimitMax;

bool Halt;

bool EmergencyStop;

// Optional profiled move

bool MoveProfiled;

double TargetPosition, TargetVelocity, TargetAccel, TargetDecel, TargetJerk;

// Optional homing

bool HomeRequest;

uint16\_t HomeMethod;

double HomeVelocity, HomeOffset;

// Optional gearing

bool FollowerEnable;

uint16\_t MasterSource;

int32\_t GearNumerator, GearDenominator;

double GearPhaseOffset;

// Optional FOC split

double IqCommand, IdCommand;

uint32\_t ValidMask;

uint32\_t CommandSeq;

uint64\_t CommandTimestamp;

uint32\_t Checksum;

uint32\_t \_reserved[8];

} DrvCommandBuf\_T;

extern volatile DrvCommandBuf\_T G\_DrvCommandBuf[NUM\_AXES];

### Drive Feedback — G\_DrvFeedbackBuf

**Role**  
Per-axis feedback buffer supplying real-time measurements and status to SEAL. **Refreshed by host before each subsystem call.** Continues updating during **Exception**. Read-only to SEAL.

**Definition & sizing**

* **Struct and array length are defined by SolFlow** for the drive identity.
* Multi-axis systems use an **array of structs** (one element per axis).
* **Do not** modify layout, names, dimensions, or storage class.
* **Fields (as defined by SolFlow)**

The following members constitute the feedback ABI. **Memory order is the authoritative SolFlow definition**; do not rely on any reordered presentation here.

* EncoderMain : int32 — The main encoder sensor.
* EncoderSecondary : int32 — The secondary encoder sensor.
* EncoderMainSpeed : single — Speed of main encoder sensor.
* EncoderSecondarySpeed : single — Speed of secondary encoder sensor.
* Iq : single — Q-axis current (A).
* Id : single — D-axis current (A). *(Note: original text labeled this “Q-channel”; treated as D-axis here.)*
* DcBusVoltage : single — DC bus voltage (V).
* PowerStageTemperature : single — Power stage temperature (°C).
* FieldAngle : single — Motor electrical field angle (units per drive config).
* LoopConfiguration : int16 — Control loop configuration (reported).
* ReferenceMode : int16 — Reference mode (reported).
* MotorOn : int16 — Motor on report.
* HallCode : int16 — Code of Hall sensors.
* STODisable : int16 — 1 if disabled by STO (Safe Torque Off).
* StatusBitField : int16 — Status bit field.
* ErrorCode : uint32 — Motor failure report.
* ConfirmRelinquishControl : int16 — Confirms release of drive from SEAL control.
* **Access & timing rules (summary)**
* **Host/scheduler writes**, **SEAL reads**.
* Host **refreshes** this buffer **before each SEAL function call**.
* During **Exception**, this buffer **continues to update**; drive commands are ignored until recovery via **Setup**.

### Setup Information (Report) — G\_SetupReportBuf

**Role**  
Per-axis **configuration and limits report** provided by the drive/host. SEAL **reads** this to understand capabilities, safety thresholds, communication settings, and sampling periods.

**Definition & sizing**

* Struct layout and array length are **defined by SolFlow** for the drive identity.
* Multi-axis systems use an **array of structs** (one element per axis).
* Users **must not** alter layout, names, dimensions, or storage class.

**Refresh rule**

* The host scheduler **refreshes G\_SetupReportBuf before each subsystem call**.
* During **Exception**, this buffer **continues to update** (commands remain ignored until recovery via Setup).
* **Fields (as defined by SolFlow)**
* MaximumPositionReference : double — Maximum allowed position reference.
* MinimumPositionReference : double — Minimum allowed position reference.
* HighPositionException : double — Position threshold that triggers an exception (high).
* LowPositionException : double — Position threshold that triggers an exception (low).
* AbsoluteSpeedLimit : single — Absolute speed limit.
* PositionModulo1 : double — Modulo count for position sensor #1.
* PositionModulo2 : double — Modulo count for position sensor #2.
* OverSpeed : single — Speed level that triggers an overspeed exception.
* AbsoluteAccelerationLimit : single — Absolute acceleration limit.
* ContinuousCurrentLimit : single — Continuous current limit.
* PeakCurrentLimit : single — Peak current limit.
* PeakCurrentDuration : single — Allowed duration at peak current.
* OverCurrent : single — Current level that triggers an over-current exception.
* UARTBaudRate : uint32 — UART baud rate.
* CANBaudRate : uint32 — CAN baud rate.
* IsPosSensorModulo1 : uint16 — 1 if position sensor #1 is modulo (wraps).
* IsPosSensorModulo2 : uint16 — 1 if position sensor #2 is modulo (wraps).
* CANId11bit : uint16 — 11-bit CAN identifier.
* Ts : single — Profiler sampling time.

**Notes & usage**

* Units, encodings, and any enumerations are **per the SolFlow drive project**.
* Exception thresholds (e.g., OverSpeed, OverCurrent, high/low position) are used by the host/drive to raise **automatic exceptions**.
* Modulo fields (PositionModulo\*, IsPosSensorModulo\*) describe sensor wrap behavior and counts for correct reference/feedback handling.
* Ts (profiler sampling time) can inform ISR tuning and timing diagnostics.

### Version Control — G\_SEALVerControl

**Role**  
System-wide **version/compatibility handshake** between SEAL and the host/drive configuration. Used to verify that the generated model matches the SolFlow database for the target drive identity before motion is enabled.

**Definition & scope**

* **Defined by SolFlow**; users must not alter layout, names, or storage class.
* **Allocated and populated by the host** (typically at initialization).
* Treated as **read-only** by SEAL.
* Unless otherwise specified by SolFlow, this is a **single (global) struct**, not per-axis.

**Update/refresh**

* Populated at startup; generally **static** during runtime.
* Read by **Setup** to validate compatibility prior to enabling the system; also available to diagnostic/telemetry paths.
* **Fields (as defined by SolFlow)**
* Version : uint16 — **SEAL database version** (major).
* SubVersion : uint16 — **SEAL database sub-version** (minor/patch).
* UserData : uint32 — **Support data** associated with the database (semantics defined by SolFlow; may encode build ID, feature flags, or integrity checks).

**Usage & policy notes**

* SEAL **Setup** (and/or the host loader) checks Version/SubVersion against the model’s expected values.
* On **mismatch**, the system must not transition to active control; per policy, the host may issue a warning and refuse to load/arm motion.
* UserData provides SolFlow-defined auxiliary signaling (e.g., capability flags or CRC), enabling stricter ABI checks when required.

If you’re ready, we can proceed to the next document section (e.g., “Building an Entry Point Function — examples”) or circle back to add any missing glossary entries (we already added **data store** and **entry call function**).

## Communication interfaces

### Overview

**Scope & topology.** The **operational core (Core 1)** of the axes manager owns the physical communication interfaces. The **number and arrangement of links are implementation-dependent** (per drive identity), typically drawn from **UART, CAN, Ethernet, and EtherCAT**. Core 1 exposes these links to SEAL through an inter-core bridge.

**IPC role and payload handling.** Core 1 and Core 2 communicate via **TI IPC**. Because the IPC mailbox is **too small for full payloads**, it is used only for **notifications and small data chunks**. The bulk data moves through **host-managed queues**:

* **RX path (into SEAL):** Core 1 captures raw bytes/frames from the line → signals Core 2 via IPC → the **Core 2 host process** collects the payload into **fixed-size, bounded queues** (per channel) and exposes them to SEAL for reading.
* **TX path (from SEAL):** SEAL enqueues payloads into **Core 2 TX queues** → notifies Core 1 via IPC → Core 1 immediately drains from IPC/queue into its **Core 1 transmit queues** and pushes frames on the wire when the line is available.

**Determinism & safety policy.**

* **No dynamic memory:** All buffers/queues are **statically allocated**; sizes are defined by the drive identity (SolFlow).
* **Non-blocking semantics:** Queue operations are **lockless and bounded**. On RX overflow or TX saturation, the enqueue/dequeue **fails without blocking** and increments a per-channel **drop/overflow counter**.

**How SEAL should use it.**

* **ISR routines** must **not** parse or copy large payloads. At most, they can set lightweight flags or read a small header.
* **IdleLoop routines** should perform the **bulk RX consumption, parsing, and TX preparation** in short, sliced chunks to preserve ISR responsiveness.
* **Exception/Abort:** Communications remain available for diagnostics and telemetry; SEAL may still read RX queues and publish status.

**Ownership & configuration.**

* Link enablement, bit rates, IDs, and EtherCAT/Ethernet stack configuration are **owned by Core 1/SolFlow** and exposed to SEAL **as-is**. SEAL **does not reconfigure** physical links at runtime.
* Channel enumeration, maximum frame sizes, and queue depths are **part of the SolFlow ABI** for the given drive identity.

**Latency model (at a glance).**

1. **Wire → Core 1** (driver/DMA)
2. **Core 1 → Core 2** (IPC notify; host pulls into RX queue)
3. **SEAL** consumes from RX queue during scheduled calls
4. **SEAL TX** enqueues → **Core 1** drains → **Wire**

This design keeps inter-core traffic deterministic, avoids heap usage, and lets SEAL interact with multiple communication lines without ever blocking the control schedule.

### UART Usage

* **Policy & ownership**The SEAL may define the use of UART for its own purposes. The UART resource is shared with core #1 (GFT Servo). Thus if SEAL grabs the UART it will not be available later for GFT-Servo.   
  The defintion is done through the UserInfo parameter, where you flag UART use and state the desired baud rate. The other parameters will be no-parity, one stop bit, and 8-bits byte. UART possession is automatic when SEAL starts.
* **Sole endpoint.** When SEAL uses UART, **SEAL is the only node** authorized to interpret UART traffic at the application level.
* **Disable Core-1 interpretation.** SEAL must **inform the axes controller (Core 1)**—via a **project-specific flag/field in G\_DrvCommandBuf defined by SolFlow**—that Core 1 must **not** interpret UART payloads. (Exact bit/field is part of the drive identity ABI.)
* **Where to set it.** Set/clear this flag in **Setup** (and keep it consistent thereafter). Wait for the appropriate **feedback/report confirmation** before accepting UART commands from the line.
* **Execution placement**
* **Parsing & response building happen in the IdleLoop.**  
  ISR code must not parse long frames or copy large buffers; use ISR only for light signalling if ever needed. The **IdleLoop** consumes RX, interprets commands, and prepares TX responses in **short slices** to maintain responsiveness.
* **Data stores & bus type**

SEAL and the host exchange UART payloads through two **data stores** that use the same **bus type**:

* **Receiver (host → SEAL):** G\_UartCyclicBuf\_in
* **Transmitter (SEAL → host):** G\_UartCyclicBuf\_out
* **Bus type:** Bus: UartCyclicBuf\_T (SolFlow-defined; do not alter)

**UartCyclicBuf\_T fields (as provided):**

* PutCounter : uint16 — Index where the **next character will be written** into UARTQueue.
* FetchCounter : uint16 — Index of the **next character to read** from UARTQueue.
* UartError : uint16 — UART error/status code (overflow, framing, parity, etc.; encoding per SolFlow).
* TxFetchCounter : uint16 — Index of the **next TX character to read** (used by the TX consumer).
* UARTQueue : uint16[1x256] — Software ring buffer holding UART characters (one character per element).

**Multi-axis note:** UART buffers are typically **global**, not per-axis. The interpretation of a communication is application specific.

* **Producer/consumer roles (lock-free ring)**
* **G\_UartCyclicBuf\_in (RX path):**
  + **Producer:** Core 1/host updates UARTQueue and advances **PutCounter** as bytes arrive.
  + **Consumer:** SEAL reads bytes and advances **FetchCounter**.
  + **TxFetchCounter** is unused in RX.
* **G\_UartCyclicBuf\_out (TX path):**
  + **Producer:** SEAL writes bytes into UARTQueue and advances **PutCounter**.
  + **Consumer:** Core 1 drains for transmission and advances **TxFetchCounter**.
  + **FetchCounter** is unused in TX.
* **Wrap, bounds, and overflow**
* **Counters wrap modulo queue length** (256).
* **Non-blocking semantics:** On **RX overflow** (producer would overrun consumer), host increments UartError and drops incoming bytes. On **TX saturation**, SEAL must detect “no space” and **defer** enqueuing (do not block); optionally increment a local drop counter.
* **Orderly access pattern:**
  1. Snapshot producer/consumer counters,
  2. Compute available bytes/space with modulo arithmetic,
  3. Move a **small bounded chunk**,
  4. Publish the updated counter.
* **Determinism & safety**
* **No dynamic memory.** Queues are **statically allocated**; sizes come from the drive identity.
* **Do not modify** the bus or store definitions; they are **SolFlow-owned ABI**.
* **Typical flow (at a glance)**

1. **Core 1** receives bytes from the UART hardware, writes them into G\_UartCyclicBuf\_in.UARTQueue, advances PutCounter, and signals Core 2 via IPC.
2. **SEAL IdleLoop** reads from G\_UartCyclicBuf\_in, interprets commands, prepares a reply, writes into G\_UartCyclicBuf\_out.UARTQueue, advances PutCounter.
3. **Core 1** drains G\_UartCyclicBuf\_out using TxFetchCounter and transmits when the line is available.

* **Arbitration reminder**
* Ensure the **“Core 1 do not interpret UART”** directive is set before SEAL starts accepting UART commands; otherwise, both ends might respond.
* Clearing this directive (e.g., during **Abort** with **Relinquish Control**) returns UART interpretation to the drive’s native firmware per SolFlow policy.

|  |
| --- |
| The shipped Seal example model includes an example of correct UART RX and TX queues use. It also implements an example minimal UART interpreter |

### CAN Usage

* **Channels & exposure**SEAL has its own dedicated CAN-Bus controller, which may or may not share physical lines with GFT-Servo, according to the hardware case. If the CAN-bus is shared with GFT-Servo, the baud rate is set by the GFT-Servo. If the implementation has separate CAN lines, you can set the baud rate, and the use of CAN-FD as you wish.   
  You can select message acceptance filters via the UserInfo parameters; you can specify up to 4 standard 11-bit filters (ID and mask) and in addition up to 4 extended (29 bit) acceptance filters.
* **Bus type:** CANCyclicBuf\_T (SolFlow-defined; do not alter).
* **Data stores:**
  + **RX (Core1 → SEAL):** G\_CANCyclicBuf\_in
  + **TX (SEAL → Core1):** G\_CANCyclicBuf\_out
* **Scope:** Typically global (not per-axis). Exact scope and sizes are part of the drive identity ABI.
* **Struct layout (CANCyclicBuf\_T)**

Memory layout and encodings are owned by SolFlow; below mirrors the provided definition.

* PutCounter : uint16 — Index of the **next message slot** to write into the queue.
* FetchCounter : uint16 — Index of the **next message slot** to read from the queue.
* CANError : uint16 — Error/status (overflow, bus-off, etc.; encoding per SolFlow).
* CANQueue : uint32[128 x 2] — Payload store (two 32-bit words per message slot).
* TxFetchCounter : uint16 — Index of the **next message slot** to transmit (used by TX consumer).
* CANID : uint32[64 x 1] — Identifier per message slot (standard 11-bit or extended 29-bit).
* DLenAndAttrib : uint16[64 x 1] — DLC and attributes per slot (RTR/IDE/FD flags per SolFlow).

**Slot mapping.** One message **slot** uses **two** uint32 words in CANQueue. With CANQueue sized [128 x 2], there are **64 slots** total. CANID and DLenAndAttrib therefore have length 64—**one entry per slot**.  
**Counters wrap modulo 64** (the number of slots). Producers/consumers advance their respective counters **per slot**, not per 32-bit word.

* **Producer/consumer roles**
* **RX buffer (G\_CANCyclicBuf\_in)**
  + **Producer:** Core 1 places received frames into the next free slot:
    - Write payload into CANQueue[slot,\*] (two words)
    - Write metadata CANID[slot], DLenAndAttrib[slot]
    - Advance PutCounter (mod 64) and notify Core 2 via IPC
  + **Consumer:** SEAL reads the slot and advances **FetchCounter** (mod 64)
* **TX buffer (G\_CANCyclicBuf\_out)**
  + **Producer:** SEAL writes payload/metadata into the next free slot and advances **PutCounter** (mod 64)
  + **Consumer:** Core 1 transmits from the slot and advances **TxFetchCounter** (mod 64)

**Non-blocking semantics.** On RX overflow or TX saturation (producer would overrun consumer), the operation **does not block**; the producer drops the message and increments CANError. SEAL should keep its own diagnostics if needed.

* **Addressing & filters**
* **SEAL must own two CAN IDs:** exactly **one standard (11-bit)** and **one extended (29-bit)**.
* SEAL **reports its IDs to Core 1** via the SolFlow-defined mechanism (part of the ABI).
* **Core 1 exposes upstream (to Core 2/SEAL)** only:
  + Messages addressed to **SEAL’s own ID(s)**, and
  + **Broadcast** frames (per project policy).
* **ID value 0 is disqualified** and must not be used. If reported as 0, Core 1 will not route frames to SEAL.
* **Execution placement & policy**
* **IdleLoop** performs CAN parsing and response building in **short slices** to preserve ISR responsiveness.
* **ISR** should not handle bulk CAN work; at most, set light flags or counters.
* **No dynamic memory; C only.** All queues are **statically allocated** with fixed sizes from the drive identity.
* SEAL **does not reconfigure** physical CAN parameters at runtime (bitrate, timing, filters); these are owned by Core 1/SolFlow and exposed read-only to SEAL via setup/feedback.
* **Quick read of a slot (conceptual)**

1. Snapshot PutCounter/FetchCounter (mod 64).
2. If FetchCounter == PutCounter, queue is empty.
3. Else read CANID[slot], DLenAndAttrib[slot], CANQueue[slot,\*].
4. Advance FetchCounter (mod 64) and return.

This fixed-queue design keeps inter-core CAN traffic deterministic, avoids heap usage, and ensures SEAL processes only the frames that matter (its IDs and broadcast).

|  |
| --- |
| The CAN service implements message queues only but not any specific protocol like CanOpen or J1939. A modeler that wishes these stacks must provide them. |

|  |
| --- |
| The shipped Seal example model includes an example of correct CAN RX and TX queues use. It also implements an example which simply transmits back everything it receives. |

|  |
| --- |
| The Seal example (see e.g. Setup handler) provides ways to access the drive's get/set object dictionary. This is implemented by direct addressing the object dictionary. No physical CAN lines are involved and no use is made of the CAN queue system presented above. |

## Building an entry point function (function call subsystem)

Entry call functions are where your application logic lives. SEAL doesn’t prescribe algorithms; it enforces **naming**, **class**, and **priority**. The patterns below are proven ways to structure functions that play nicely with the scheduler and data stores.

1. **Common skeleton (applies to all classes)**

**At call entry (automatic by scheduler):** feedback + setup reports are refreshed into data stores.  
**At call exit (automatic by scheduler):** command data stores are pushed to the drive.

**Inside your function:**

1. **Snapshot**: Copy the few feedback/setup values you need into locals (prevents mid-function inconsistency if you change your mind).
2. **Compute**: Do bounded, deterministic math (fixed widths, saturations).
3. **Commit**: Write only the command/flags you own. Keep write scope minimal

### Example 1: Symulink block elements

Consider the following block diagram:



This diagram implements a simplistic cascaded gain over PI controller.

The principle is:

* + Take whatever data you want from the data stores using bus selectors (here the reference is take from the command buffer, and the feedback from the feedback buffer)
  + Use Simulink symbols (refer to the above section on limitations) to draw your algorithm graphically.
  + Use a bus merger block to merge the calculated commands into the command buffer.

### Example 2: Matlab code in a Matlab function block

Consider the following block diagram:



This is an implementation of a position command profiler as a Matlab function block. The output of the function is a vector of position speed and torque commands; they are merged as output to the command buffer using a bus merger block.

The Matlab function block itself is:

function y = fcn

% User Data stores

global G\_PosProfilerState

global G\_PosProfilerData

% System Data stores

global G\_SetupReportBuf

% Application code

Ts = double( G\_PosProfilerData.Ts) ;

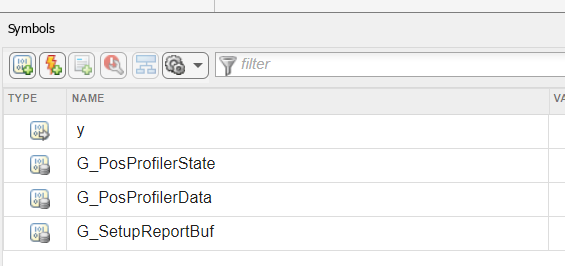
…. Additional algorithm code you can find in example system

% Output

y = [NewFiltState1 ; (NewFiltState1-G\_PosProfilerState.FiltState(1))/ max(Ts,1e-5); 0 ] ;

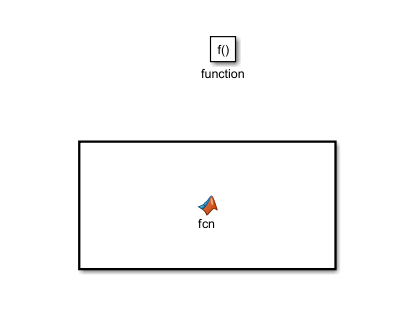
end

Note that the data stores (global variables) in use here are declared as globals in the Matlab function. These data stores must be declared and typed in the symbols pane as data stores to be recognized.



### Example 3: Call for external service functions

Consider the following setup block:



It looks like a Matlab function with no I/O.

Take a look inside:

function fcn

% Use a data store as a global

global G\_UserInfo

% Object Initialization code

SetObject2Drive( uint16(hex2dec('60f0')), uint16(1) , single(150000) ,VarDataTypes.T\_uint32) ;

% Get object from file

G\_UserInfo.junk1 = single( GetObjectFromDrive( uint16(hex2dec('60f0')), uint16(1) , VarDataTypes.T\_uint32) ) ;

end

function retCode = SetObject2Drive( Index, subIndex ,value , datatype )

% call an external C function

retCode = int32(0);

coder.ceval('SetObject2Drive', uint16(Index), uint16(subIndex), double(value) , uint16(datatype) ,coder.ref(retCode));

end

function [value,retCode] = GetObjectFromDrive( Index, subIndex , datatype)

retCode = int32(0);

value = double(0) ;

coder.ceval('GetObjectFromDrive', uint16(Index), uint16(subIndex), uint16(datatype) , coder.ref(value), coder.ref(retCode));

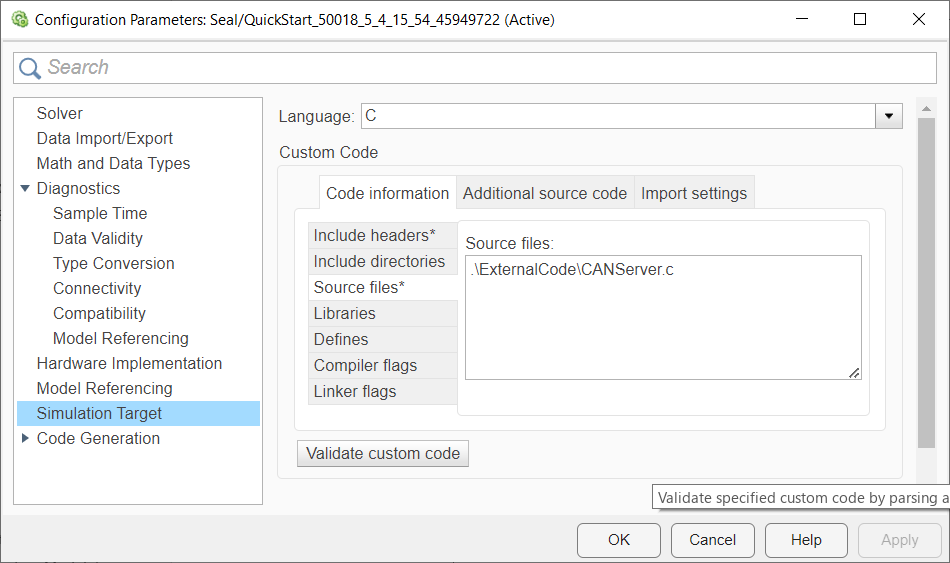
end

The data store G\_UserInfo is used within the function as global; don’t forget to declare it as data store, see above example.

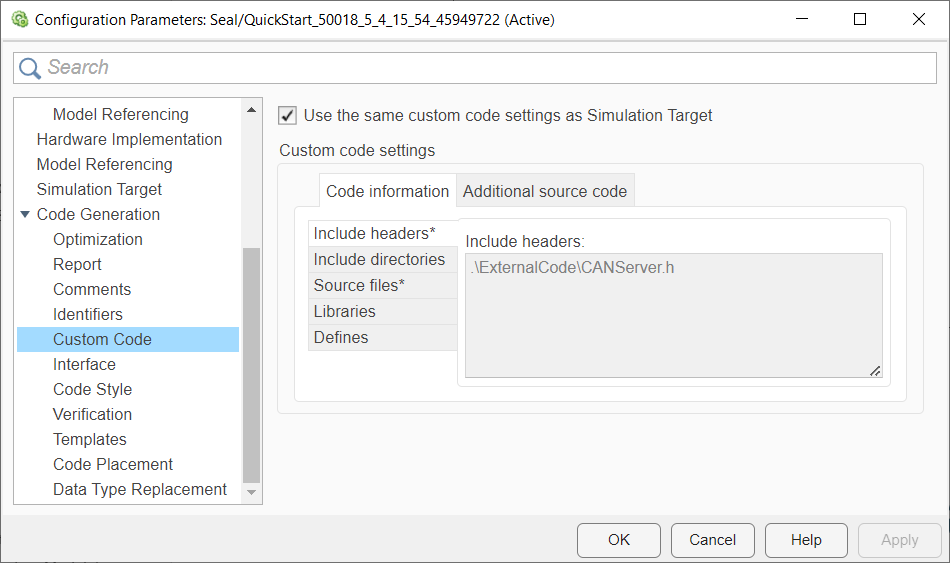
This setup function sets and reads objects in the drive's object dictionary. It uses host services that are addressed as C function, using the coder.ceval Simulink scheme.

Notes:

* + The C code may be defined or not in the replacement table. The post processor will replace this function by the host service anyway.
  + The C code must be defined as additional code in the model properties:



And don’t forget in code the generation custom code window, to ask the use of the same custom code setting (this is already done in the SEAL example provided)



# Preparing buses and signals

## Introduction

The modeler must make Simulink coder aware of the parameters you insert, and of any non-scalar data types in use.

The model is aware to custom data type and to parameter instances through the SLDD (Simulink Data Dictionary) attached to the model.

Seal comes with the dictionary SEALGenericTypes\_1.sldd.

This SLDD is defined in **InitialSetup.m.**

The SLDD has sections: The one of interest here is Design Data

*DesignDataSection = getSection(SlddHandle,'Design Data');*

Note that a data dictionary is a handle class, so it is passed to functions as reference. Changes made to an SLDD argument inside a function will persist when the function exits.

## Complex data type: the bus.

A bus is equivalent to a C language struct. Here is a typical bus definition for profiler parameters:

*% Define the bus elements, their data type and their help*

*pelems(8,1) =Simulink.BusElement ;*

*pelems(1) = SetBusElement('PositionTarget','single',"Final position to arrive" ) ;*

*pelems(2) = SetBusElement('ProfileSpeed','single',"Maximum speed" ) ;*

*pelems(3) = SetBusElement('ProfileAcceleration','single',"Maximum Profile acceleration" ) ;*

*pelems(4) = SetBusElement('ProfileDeceleration','single',"Maximum Profile deceleration" ) ;*

*pelems(5) = SetBusElement('ProfileFilterDen','double',"Filter monic polynomial for profile filtering",[4,1] ) ;*

*pelems(6) = SetBusElement('ProfileFilterNum','double',"Filter numerator for profile filtering",[1,1] ) ;*

*pelems(7) = SetBusElement('ProfileDataOk','uint16',"Flag that profiler data is consistent",[1,1] ) ;*

*pelems(8) = SetBusElement('Ts','single',"Profiler sampling time " ) ;*

*% Generate a generic bus*

*PosProfilerData\_T = Simulink.Bus;*

*% and populate it with the elements we defined*

*PosProfilerData\_T.Elements = pelems;*

*% Insert the bus definition into the SLDD*

*assignin(DesignDataSection,'PosProfilerData\_T',PosProfilerData\_T);*

Note:

We defined a data structure, not any instance of it. For instantiation, see below.

## Generate a parameter instance

When Simulink model has a parameter, it can take it from the SLDD, or if none is found, from the Matlab base workspace. Definition of a parameter in the SLDD is safer, as it cannot be accidentally overridden.

First let's create a scalar parameter for a PI controller.

For this purpose, Seal comes with a ready function (which you can examine), in the line below it defines a parameter named Kp with the value of 5.

*SetSealParameter(DesignDataSection,'Kp',5) ;*

Instantiating a bus is a little more complicated, take for example the profiler data whose type we created above:

*% Generate an instance of the bus in the Matlab base worspace*

*PosProfilerDataStructPrototype = Simulink.Bus.createMATLABStruct('PosProfilerData\_T');*

*% And initialize its values*

*PosProfilerDataStructPrototype.PositionTarget = 0 ;*

*PosProfilerDataStructPrototype.ProfileSpeed = 1 ;*

*PosProfilerDataStructPrototype.ProfileAcceleration = 1 ;*

*PosProfilerDataStructPrototype.ProfileDeceleration = 1 ;*

*PosProfilerDataStructPrototype.ProfileFilterDen = [1.000000000000000 ; -0.874052257056399 ; 0.280840263500717 ; -0.024477523271653 ] ;*

*PosProfilerDataStructPrototype.ProfileFilterNum = 0.382310483172666 ;*

*PosProfilerDataStructPrototype.ProfileDataOk = 0 ;*

*% Then use the instance in the workspace to generate an instance in the SLDD*

*% For this we create a parameter (again in the base workspace)*

*% and assign it with data type, initialization value and storage class.*

*PosProfilerData\_init = Simulink.Parameter;*

*PosProfilerData\_init.DataType = 'Bus: PosProfilerData\_T';*

*PosProfilerData\_init.Value = PosProfilerDataStructPrototype ;*

*PosProfilerData\_init.StorageClass = 'ExportedGlobal';*

*% And finally put it to the SLDD*

*assignin(DesignDataSection,'PosProfilerData\_init',PosProfilerData\_init);*

**CAVEAT:**

Simulink works against the base workspace. If you want to make the above statements inside a function you will fail, as you will not be working in a scope capable of interacting with Simulink. If you want to encapsulate the technics, your function will have to relay actions to the base space. For an example observe the SetSealParameter() function internals.

# Appendix B – Glossary

The glossary explains terms in use in this document.

Some of the explained items are documented in the Simulink manual set. These terms are described here for convenience, heavily biased towards their use in SEAL. Of course, the description in the Simulink manual set is more comprehensive and accurate.

## Topics

* ABI
* Bus Signal
* Code Replacement
* Data Stores
* Entry Point functions
* Enumerated Data Types
* External Mode
* Function-Call Subsystem
* Function-Export Model
* Harness Model
* InitialSetup.m
* SEALSystemTypes.m
* Simulink Parameter
* Simulink Signal
* SLDD (Simulink Data Dictionary)
* SolFlow
* Storage Class
* User Definition Script

## Detail

* **ABI (Application Binary Interface)**

**What it is:** The binary-level contract that lets independently built components interoperate without recompilation. It fixes **call/return rules**, **data layouts**, and **type sizes/alignments** so both sides read/write the same bytes the same way.

* **Bus Signal**

In Simulink, a bus groups multiple signals together into a single line for organizational clarity and structured data handling. In the context of **Embedded Coder**, a bus is a **strongly typed entity**: it is explicitly defined (usually in an SLDD) with fixed field names, data types, and dimensions. During code generation, a bus is translated directly into a corresponding **C-language struct**, preserving its hierarchy and ensuring a one-to-one mapping between the model definition and the generated code interface. This makes buses the standard mechanism for defining structured interfaces such as command/feedback data between SEAL and the drive.

* **Code Replacement**

In **Embedded Coder**, code replacement is the process of substituting certain generated code constructs with alternative implementations that are more efficient, hardware-specific, or environment-compliant. It does not affect data storage (that is governed by storage class), but instead alters the way operations and services are invoked.

Examples include:

* Replacing high-level math functions (e.g., sin()) with optimized intrinsics or assembly-access implementations (e.g., \_\_sin()).
* Redirecting calls from generic simulation drivers to service routines provided by the hosting scheduler.
* Mapping standard library calls to project-specific certified equivalents.

This mechanism ensures that generated code integrates seamlessly into the target environment while preserving the intended Simulink semantics.

1. **Data Stores**

All subsystems operate over a set of **global data variables**, implemented as **data stores**. These represent the common memory pool accessed across Setup, ISR, IdleLoop, Exception, and Abort routines.

Two categories of global data structures are used:

1. **Drive Interface Structures** – Fixed by SolFlow. These provide the standardized interface between the SEAL module and the drive hardware, carrying feedback values, command references, and parameter data.
2. **User-Defined Data Structures** – Optional application-specific items defined by the user. These allow application logic to maintain state, temporary buffers, or custom interface data.

The data store mechanism ensures consistency across all SEAL routines, allowing asynchronous subsystems (such as ISR and Exception) to safely exchange state with the rest of the application.

**Entry Call Function**  
A function generated from a Simulink **function call port**. Its name matches the port name exactly, and it serves as the entry point invoked by the SEAL scheduler on Core 2. Entry call functions are classified into **Setup, ISR, IdleLoop, Exception, or Abort** based on their naming preamble. Their execution order and priority are determined by port naming conventions and port numbering.

**Enumerated Data Types**

An **enumerated data type** defines a variable that can take one of a limited set of named values, rather than arbitrary numbers. In Simulink, enumerations improve clarity by replacing numeric codes with descriptive names (e.g., DriveMode.Standby, DriveMode.Run, DriveMode.Fault).

In **Embedded Coder**, enumerations may be translated into either:

* A C enum, listing symbolic names with assigned integer values, or
* A set of #define constants, depending on generation settings and target constraints.

Enumerations are typically stored in the SLDD, ensuring consistency across models and code. They are widely used for defining drive modes, state machines, and interface protocols.

* **External Mode**

A Simulink feature that allows quasi real-time communication between a running target (such as the TI C2000 controller hosting SEAL) and the Simulink model on the host computer. External Mode provides parameter tuning and signal monitoring while the target code is executing. It relies on a communication channel (normally Ethernet but could be also UART or CAN) implemented via an **rtiostream** interface.

It is a very limited mode: The Simulink clock is synchronized to the target clock. Simulink can only deal with one External Mode target, and it doesn’t simulate anything: it just relays parameters and listens signal reports.

* **Function-Call Subsystem**

A Simulink subsystem block that executes only when explicitly “called” by a function-call signal, instead of running continuously. In a function-export model, the system consists of several such function-call subsystems, each representing a different callable routine (e.g., initialization, cyclic control, communications). In generated code, these become callable C functions.

* **Function-Export Model**

A special type of Simulink model for code generation, where the model’s logic is organized into callable entry points rather than continuous execution. Each callable entry point corresponds to a **function-call subsystem** in the model, which becomes a callable function in generated code. This structure is essential when the execution order and timing are dictated by an external scheduler, such as the drive controller firmware.

* **Harness Model**

In Simulink, a harness model is a test environment linked to a component model (such as SEAL). It provides inputs, schedules execution, and observes outputs in a controlled setup. In the SEAL environment, the harness model serves as the **scheduler model**: it supplies function-call signals to activate SEAL’s subsystems and includes the drive model, thus reproducing in simulation the interactions expected in the deployed system.

* **InitialSetup.m**

A MATLAB script included with SEAL that performs the initial configuration of the development environment. It sets up paths, parameters, and ensures that Simulink can compile and execute the SEAL model.

* **SEALSystemTypes.m**

A MATLAB file that defines drive-specific control and status data structures. These structures formalize the interface between the SEAL module and the drive model, ensuring alignment between simulation and hardware integration.

* **Simulink Parameter**

A **design-time or tunable variable** that defines constants, coefficients, or thresholds used in a Simulink model (e.g., controller gains, saturation limits). Parameters are static in nature — they configure the system rather than represent changing runtime data.

In **Embedded Coder**, parameters map into code depending on their **storage class**:

* **Compile-time constants** → #define macros or const variables.
* **Tunable parameters** → global variables accessible at runtime.
* **Custom-mapped parameters** → placed into specific memory sections or calibration layers.

Parameters represent the **configurable constants** of the system.

* **Simulink Signal**

A **runtime entity** representing values that flow between Simulink blocks (e.g., measured speed, torque, current). Unlike parameters, signals vary with time and represent the dynamic state of the system.

In **Embedded Coder**, signals map into code based on their **storage class**:

* As local variables (default).
* As global variables for external access or logging.
* As struct fields when defined within a bus.

Signals represent the **time-varying data flow**, in contrast to parameters, which define fixed or tunable configuration values.

* **SLDD (Simulink Data Dictionary)**

A **Simulink Data Dictionary** is a centralized repository for storing data definitions used by Simulink models. An SLDD ensures consistency, traceability, and version control of all modeling entities.

It typically contains:

* Parameters with values, types, and storage classes.
* Signal definitions.
* Bus and struct definitions.
* Enumerations and data types.

In Embedded Coder workflows, the SLDD is the backbone of large projects, ensuring that multiple models share a consistent definition of data and interfaces.

* **SolFlow**

The GFT development environment that supervises SEAL integration. SolFlow interrogates the drive model and revision, delivers the correct interface definitions and simulation models to Simulink, and manages build and download via the TI toolchain. It also maintains version control of the generated SEAL code, ensuring traceability.

* **Storage Class**

In **Embedded Coder**, the storage class of a data object (signal, parameter, or bus) determines how it is represented in generated C code. It controls:

* **Scope** – local, global, or externally defined.
* **Representation** – macro, const, RAM variable, or struct field.
* **Integration** – placement in memory sections, linkage to calibration systems, or adherence to coding standards.

Storage class is the primary mechanism for aligning generated code with project coding guidelines and hardware integration needs.

* **User Definition Script**

A MATLAB script provided by the user to extend SEAL with application-specific structures and variables. It defines the data required by proprietary algorithms. Working examples are supplied to help users create their own definition files.

# Appendix: Fixed tables and mappings

The following address tables serve as fix location to communicate between SEAL and its host scheduler in Core 2.

# Practice

## List of functions for the user’s benefit

function CreateProtectedBus(DesignDataSection,busName,DataStoreName,IsProtected,InitName,InitValue,Description)

## Caveats

Do not use nested bus with CreateProtectedBus

Do not use very long bus

Beware of the uint yupes (roll over properties)

# Todo:

Watch that function names do not use reserved keywords

Pass CAN / UART data to Idle Loop routines

Analyze protected structs, prep BG (double) versions.

Dirty markers + Dirty copiers

Analyze BG functions that call protected structs, implement for them routines for protection

**SetupEmpty**

Checks :

C:\Projects\SEAL\Hfile2String\HfileAsString

Compare the H files in the target to the H file internal

Report Target contents on communication

A routine was written

Document:

How SolFlow decides drive and associated SEAL environment

Harness interfaces for passing CAN and UART data

Debugging instructions

Models

Harness model

Drive model

Gross Level

Communication interfaces are actual HW. SEAL has its own core 2 services both for CAN and for UART. When seal has them, CPU2 has granted permission to use different interfaces. It will use UART directly, **CAN will have routing to second CAN device!**

Set communication interfaces to pointers, and allocate them in the HOST so communication buffers are hosted (for DLOAD)