

モデルからコードへ 設計レイヤーの違いと機能実装の方法について

MathWorks Japan

アプリケーションエンジニアリング部

はじめに

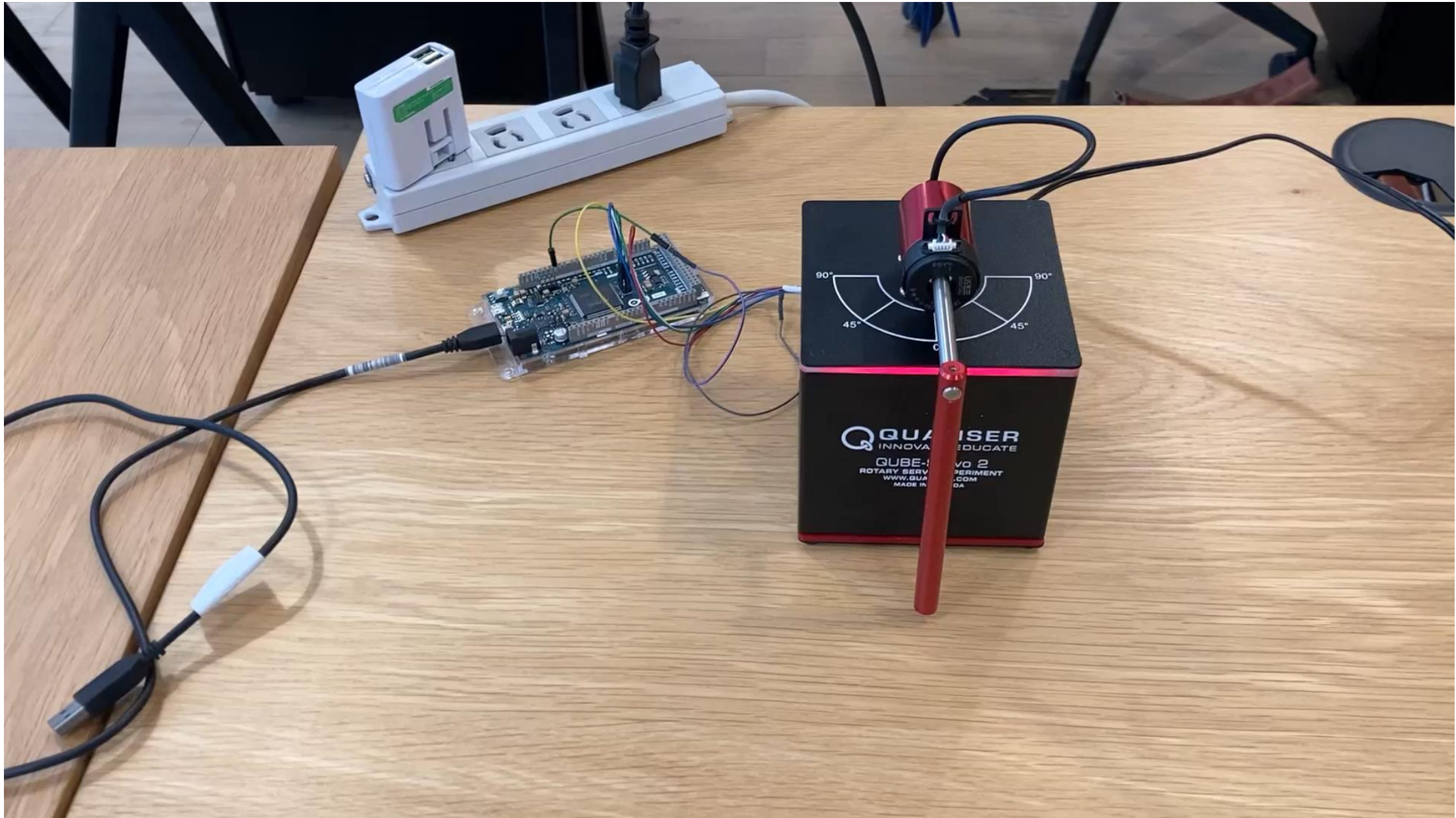
本セッションの対象者

- 制御アルゴリズムをコントローラーに実装したい方

本セッションでお伝えしたいこと

- コントローラー実装に役立つ「レイヤー」の考え方
- 連続モデルを離散化する方法とその必要性について
- **Simulink**モデルから生成した**C/C++**コードを**Arduino**に実装する方法

【ゴール】 制御機能を実機に実装



アジェンダ

- 背景
- 機能の抽象化レイヤー
- アルゴリズムの離散化
- コード生成と実装

アジェンダ

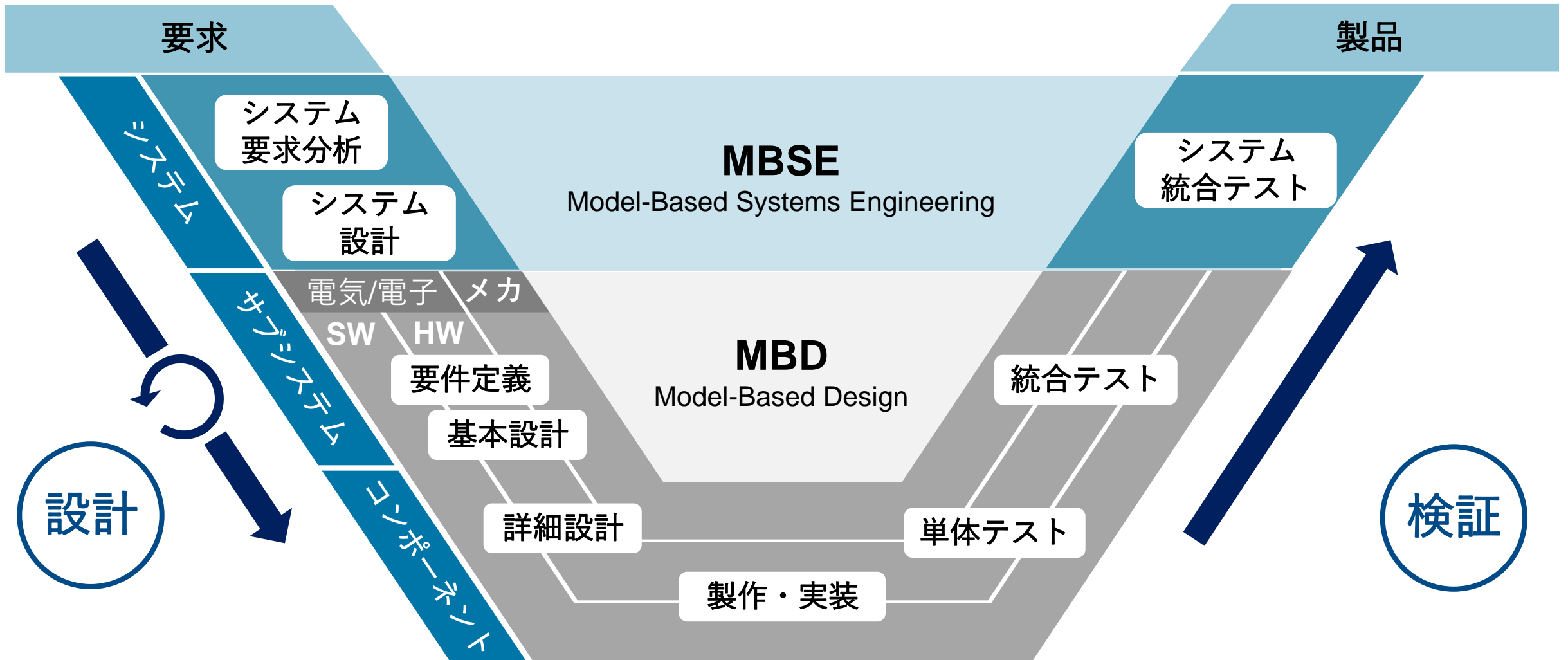
- 背景

- 機能の抽象化レイヤー

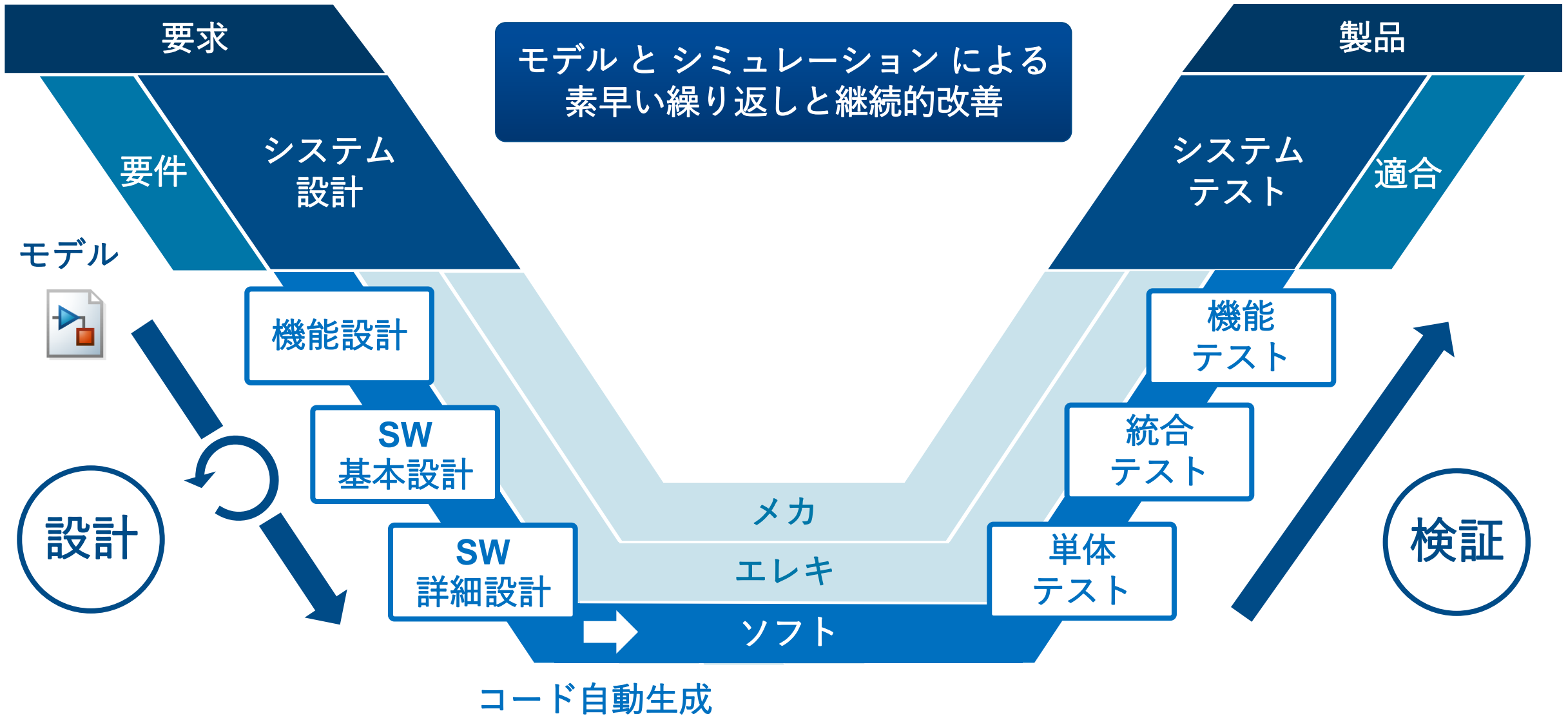
- アルゴリズムの離散化

- コード生成と実装

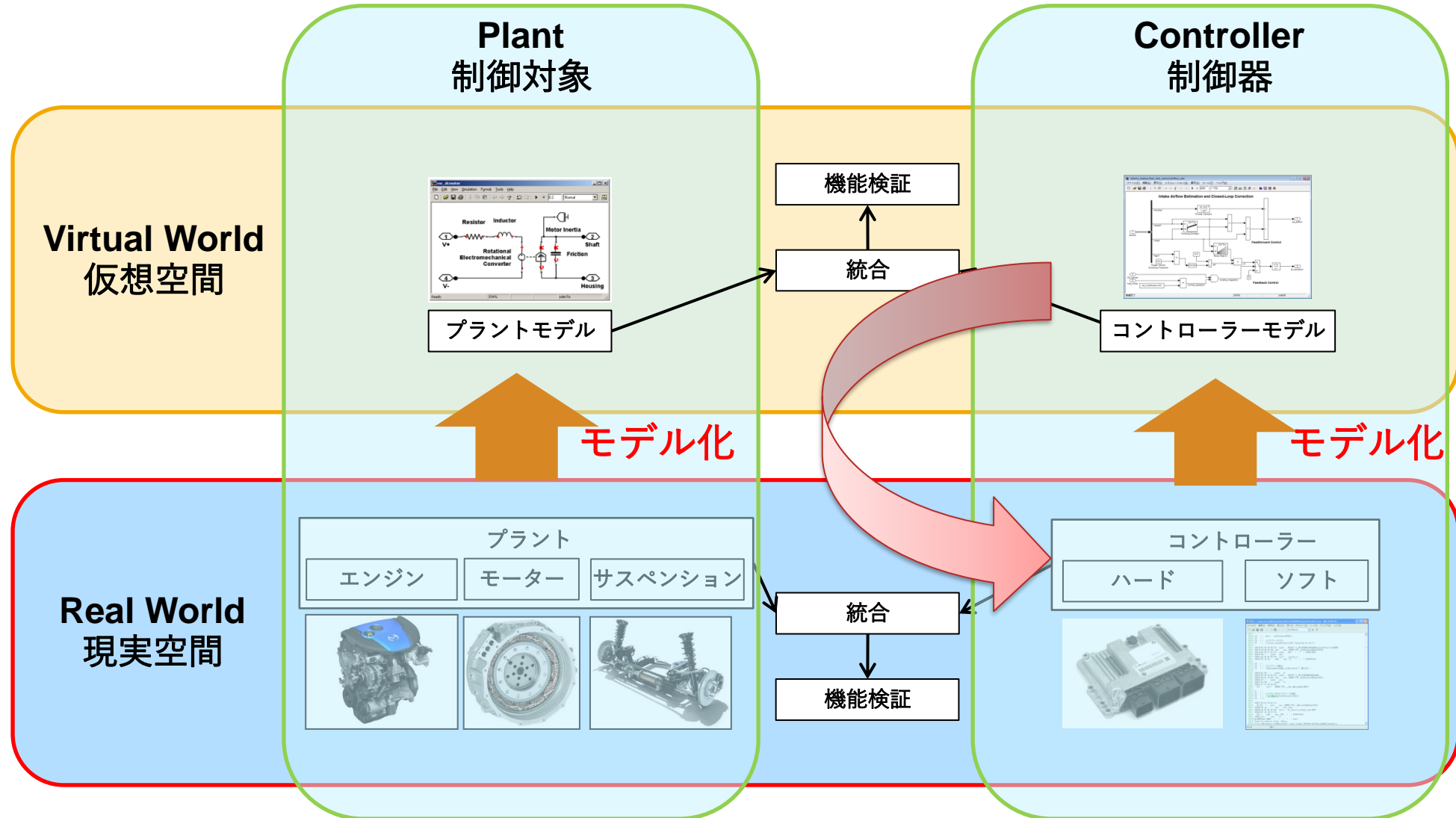
MBDにおけるVプロセス



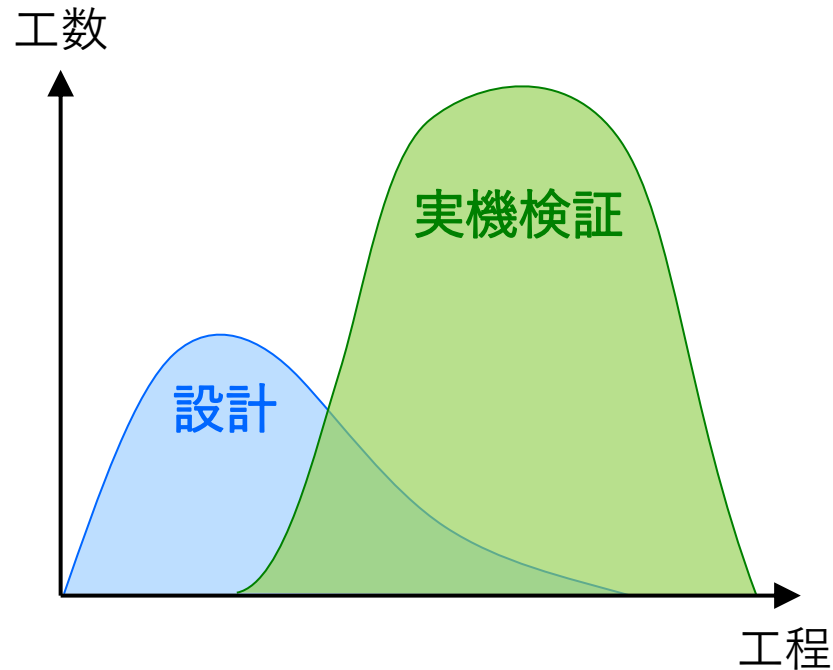
MBDにおけるVプロセス



MBDの例

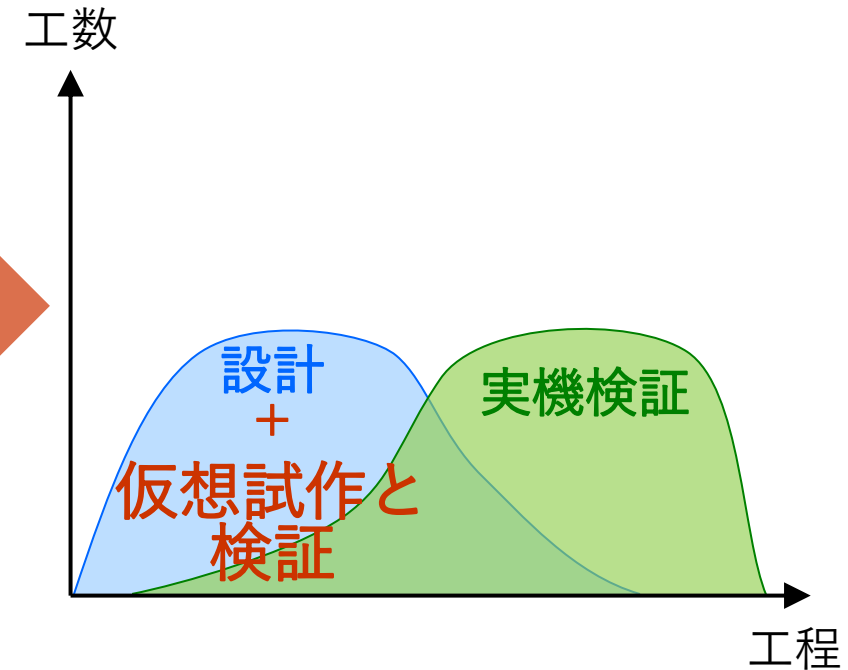


メリットは「検証工程の前倒し」ができること



従来開発（実機試験中心）

- 不具合修正コストが高い
後工程にテストが集中
- 設計抜け漏れが起こり易い



MBD

- 不具合修正コストが低い
開発上流にテストを前倒し
- 設計抜け漏れを早期に発見・修正

重要なポイント

- ハードウェアとソフトウェアが複合されたシステムは複雑である
- 実機を使う前にシミュレーションで設計、検証すると効率が良い
- シミュレーション環境で設計した制御アルゴリズムは、実機に実装し、リアルタイム実行する必要がある

アジェンダ

- 背景

- 機能の抽象化レイヤー

- アルゴリズムの離散化

- コード生成と実装

【参考】OSI参照モデル

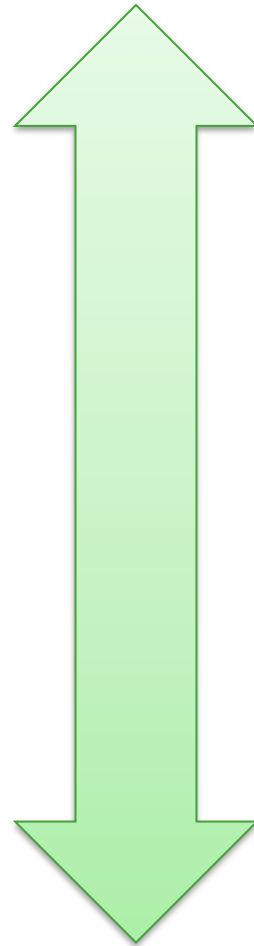
異種間のデータ通信を実現するための、ネットワーク構造の基本的な設計方針

層	名称	規格（プロトコル）	概要
7層	アプリケーション層	HTTP,FTP,POPなど	個々のアプリケーション
6層	プレゼンテーション層	SMTP,FTP,Telnetなど	データの表現形式
5層	セッション層	TLS,NetBIOSなど	通信手段
4層	トランスポート層	TCP,UDP,NetWare/IPなど	エンド間の通信制御
3層	ネットワーク層	IP,ARP,RARP,ICMPなど	データを送る相手を決め 最適な経路で送信
2層	データリンク層	PPP,Ethernetなど	隣接する機器同士の通信を実現
1層	物理層	RS-232,UTP,無線など	物理的な接続、電気信号

【参考】OSI参照モデル

異種間のデータ通信を実現するための、ネットワーク構造の基本的な設計方針

層	名称
7層	アプリケーション層
6層	プレゼンテーション層
5層	セッション層
4層	トランスポート層
3層	ネットワーク層
2層	データリンク層
1層	物理層

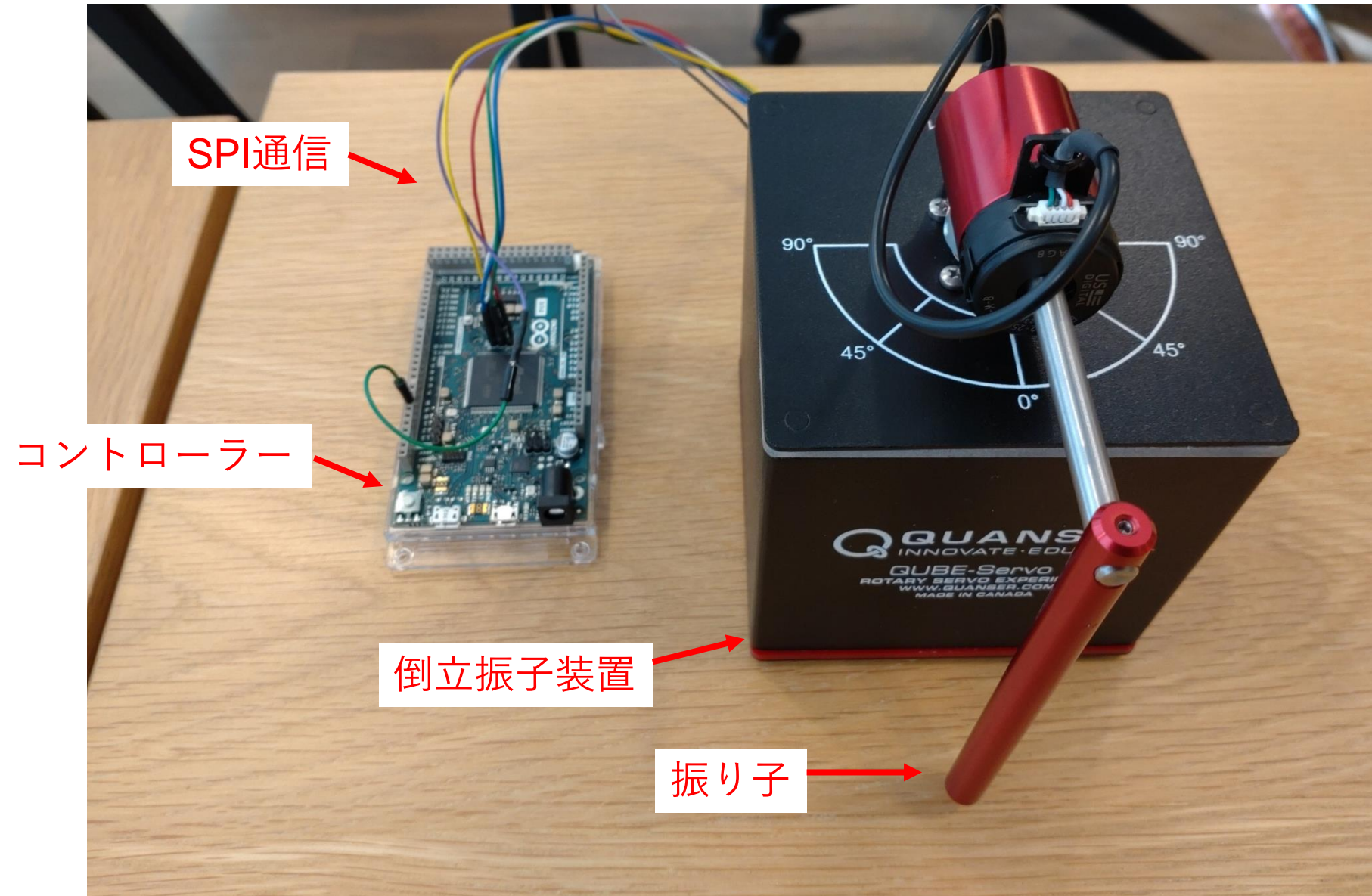


抽象的
ソフトウェア寄り

技術分野や抽象度によってレイヤーを分けると、効率よく設計できる！

具体的
ハードウェア寄り

倒立振子の制御システム 実機構成



倒立振子の制御システムをレイヤー分けしてみる

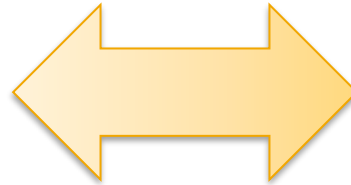
レイヤー	技術領域	項目
アプリケーション	アプリケーションソフトウェア	状態フィードバック制御
OS	システムソフトウェア	カーネル、割り込み
計算装置	電子回路	Atmel SAM3X8E (ARM Cortex-M3)
電気信号	通信	SPI
物理	電気、機械	スイッチング回路素子、モーター、振り子、エンコーダ

レイヤーを意識するメリット

システムの一部が変わっても、他のシステムを変えなくてよい。

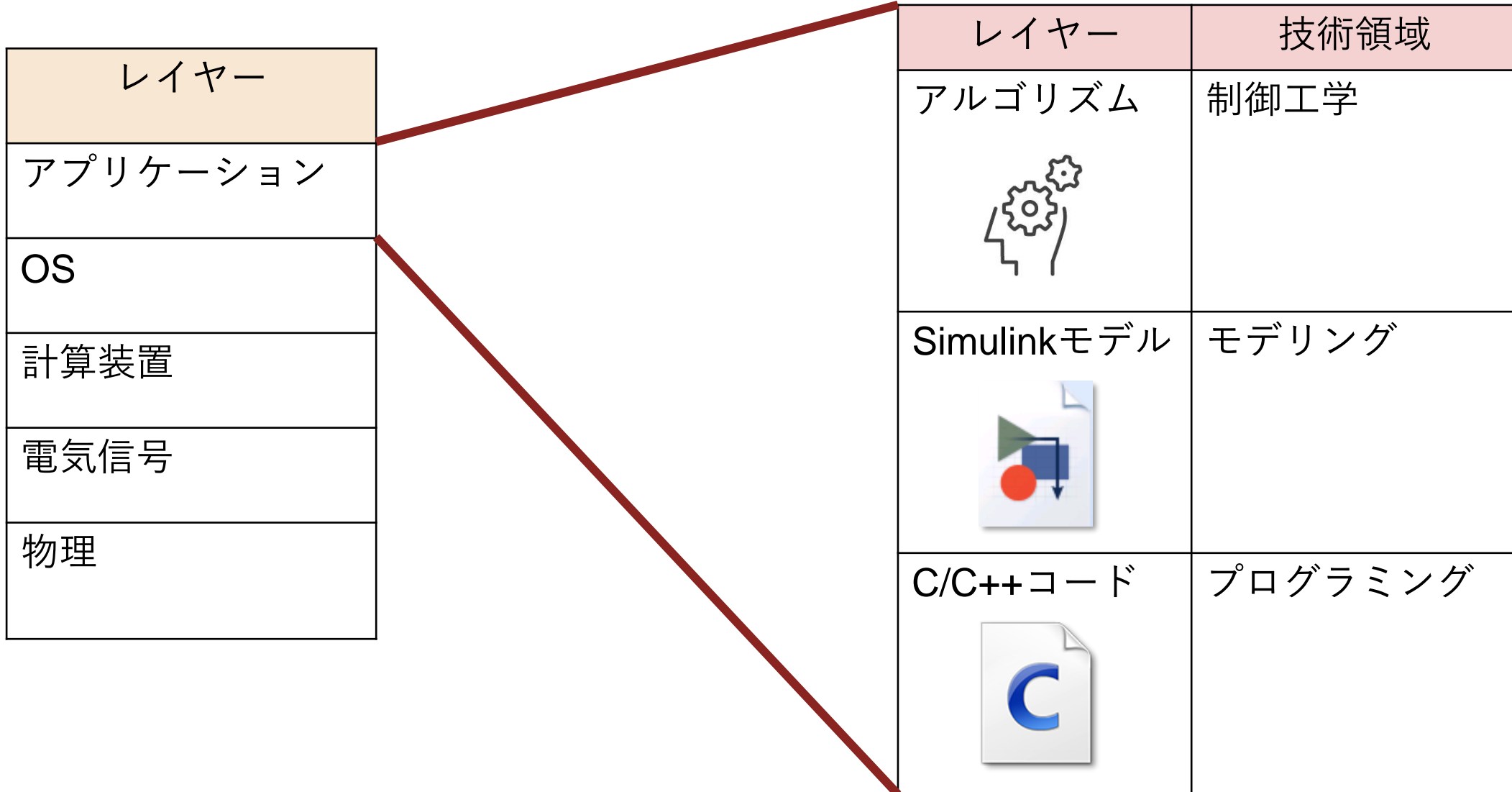
変更点が少ない = 低コスト = 開発効率向上

システムA
状態フィードバック制御
カーネル、割り込み
Atmel SAM3X8E (ARM Cortex-M3)
SPI
スイッチング回路素子、 モーター 、 振り子 、エンコーダ

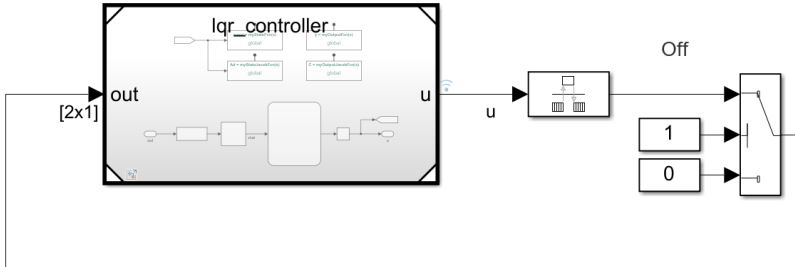


システムB
状態フィードバック制御
カーネル、割り込み
Atmel SAM3X8E (ARM Cortex-M3)
SPI
スイッチング回路素子、 エンジン 、 タイヤ 、エンコーダ

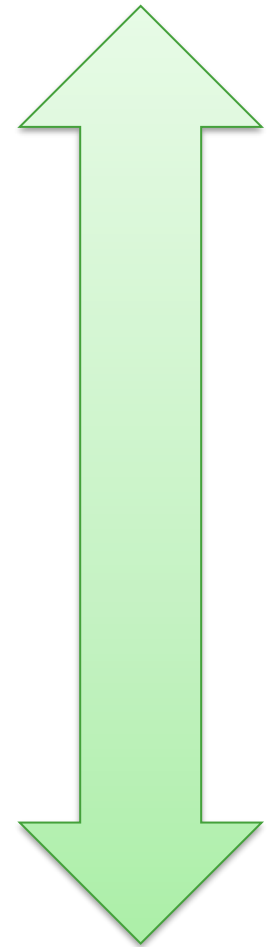
実は、アプリケーションレイヤーも細分化できる



倒立振子制御アルゴリズムのレイヤー

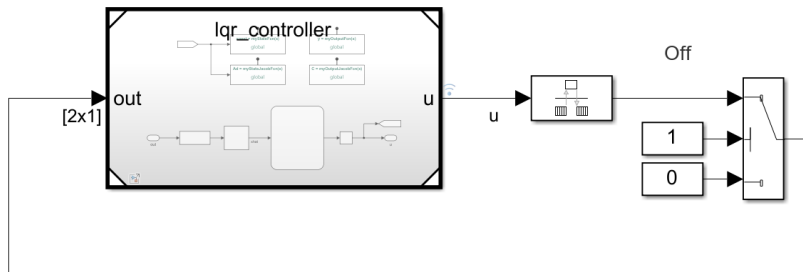
レイヤー	技術領域	項目
アルゴリズム	数式化 (制御工学)	$\dot{x} = Ax + Bu$ <p>のとき、操作量は</p> $u = -R^{-1}B^T Px$
Simulinkモデル	モデリング	
C/C++コード	プログラミング	<pre> if (std::isnan(lqr_controller lambda = (rtNaNF); } else if (std::isinf(lqr_con lambda = (rtNaNF); } else if (lqr_controller_dep lambda = 0.05; </pre>

抽象的



具体的

レイヤーごとに注力領域が分かれる

レイヤー	項目	設計する内容
アルゴリズム	$\dot{x} = Ax + Bu$ のとき、操作量は $u = -R^{-1}B^T Px$	状態空間モデル、最適レギュレータ、フィードバック制御則
Simulinkモデル		実行周期、計算順序、状態遷移、データ型
C/C++コード	<pre> if (std::isnan(lqr_controller lambda = (rtNaNF); } else if (std::isinf(lqr_con lambda = (rtNaNF); } else if (lqr_controller_dep lambda = 0.05; </pre>	タイマー割り込み、メモリ管理、I/Oデバイス操作

機能の抽象化レイヤーを意識するメリット

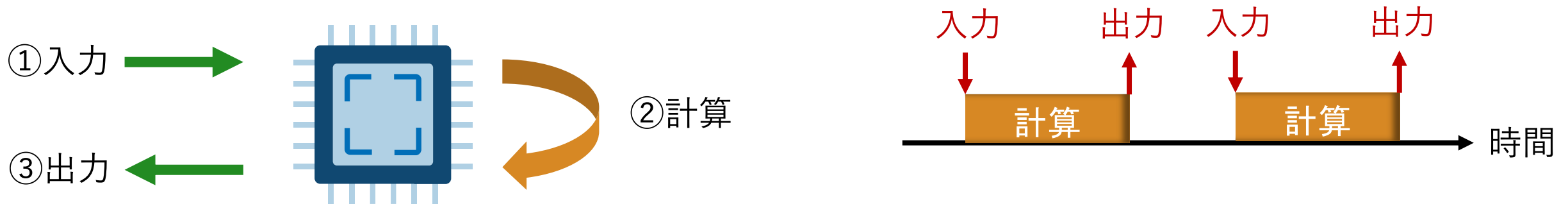
- 設計した要素を他のシステムに転用でき、開発を効率化できる
- レイヤーごとに開発すべき技術内容や観点が異なるため、分けると見通しが良くなる

アジェンダ

- 背景
- 機能の抽象化レイヤー
- アルゴリズムの離散化
- コード生成と実装

離散化の必要性

- 設計した制御アルゴリズムを実現させるには、コントローラーでリアルタイム制御を行う必要がある。
- しかし、デジタルのコントローラーは、連続に動くことができない。
- よって、連続時間で設計したアルゴリズムは、特性を維持したまま離散化して、コントローラーに実装する必要がある。

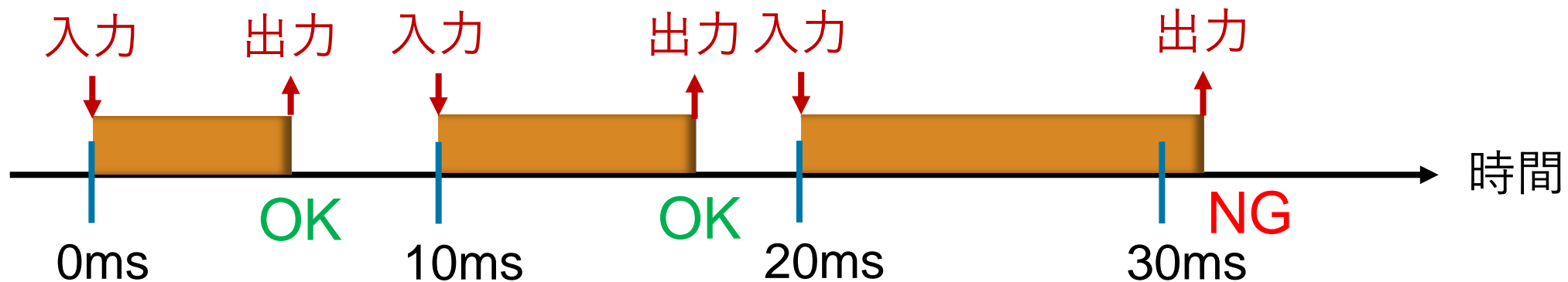


コントローラーは、これを繰り返すだけ

リアルタイム制御とは

実行周期10msの場合

計算開始  計算終了



決められた時間内に確実に計算を終え、結果を出力できること。
計算が速いことではない。

リアルタイム制御を実現するには

- 一定の時間間隔でセンサー値を取得し、計算し、結果を出力するために、タイマー割り込みを実装する。

```
void setup() {  
  pinMode(LED_BUILTIN, OUTPUT);  
  TimeManager::start_timer(TC1, 0, TC3_IRQn,  
  .....  
  TimeManager::SAMPLING_TIME_STEP_MSEC);  
}
```

タイマー割り込みの
設定を最初に1回実行する

```
void TimeManager::start_timer(Tc *tc, uint32_t channel, IRQn_Type irq_type,  
  .....  
  uint32_t sampling_time_step_msec) {  
  pmc_enable_periph_clk((uint32_t)irq_type);  
  TC_Configure(tc, channel,  
  .....  
  TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC | TC_CMR_TCCLKS_TIMER_CLOCK1);  
  uint32_t rc = (VARIANT_MCK / 2 / 1000) * sampling_time_step_msec;  
  TC_SetRC(tc, channel, rc);  
  TC_Start(tc, channel);  
  tc->TC_CHANNEL[channel].TC_IER = TC_IER_CPCS;  
  tc->TC_CHANNEL[channel].TC_IDR = ~TC_IER_CPCS;  
  NVIC_EnableIRQ(irq_type);  
}
```

※このコードはArduino Due用のコードです。
マイコン開発環境によってコードの内容が変わります。

```
void TC3_Handler() {  
  /* タイマー割り込みの制御処理 */  
}
```

タイマー割り込みで呼び出される関数部分

離散化

- 連続時間（ラプラス変換のs領域）で表現されたモデルはz変換により離散化できる。
- 古典制御と現代制御においては、制御器に含まれる時間微分と時間積分を離散化するだけでよい。

例：後退オイラー法

$$y = \frac{K_i}{s} u \quad \longrightarrow \quad y[k] = y[k-1] + K_i \cdot ts \cdot u[k]$$

$$y = K_d s u \quad \longrightarrow \quad y[k] = K_d \cdot \frac{u[k] - u[k-1]}{ts}$$

K_i は積分ゲイン、 K_d は微分ゲイン、 $y[k]$ は積分器、微分器の出力、 $u[k]$ は積分器、微分器の入力、 ts はサンプリングタイムステップ、 k は今のステップの値、 $k-1$ は前回のステップの値

離散化すると、アルゴリズムをコードで表現できる

積分

$$y[k] = y[k - 1] + K_i \cdot ts \cdot u[k]$$

```
double integrate_u(double u) {  
    · double Ki = 2.0;  
    · double ts = 0.01;  
  
    · static double y = 0.0;  
  
    · y = y + Ki * ts * u;  
    · return y;  
}
```

微分

$$y[k] = K_d \cdot \frac{u[k] - u[k - 1]}{ts}$$

```
double differentiate_u(double u) {  
    · double Kd = 2.0;  
    · double ts = 0.01;  
    · double y = 0.0;  
  
    · static double u_1 = 0.0;  
  
    · y = Kd * (u - u_1) / ts;  
    · u_1 = u;  
  
    · return y;  
}
```


【参考】状態空間モデルの連続と離散

連続時間の状態方程式： $\dot{x}(t) = \underline{A}x(t) + \underline{B}u(t)$



離散化：
$$\frac{x[k+1] - x[k]}{ts} = Ax[k] + Bu[k]$$



離散時間の状態方程式： $x[k+1] = \underline{(I + A \cdot ts)}x[k] + \underline{(B \cdot ts)}u[k]$

連続時間と離散時間で、 x と u の係数行列が異なるので注意。

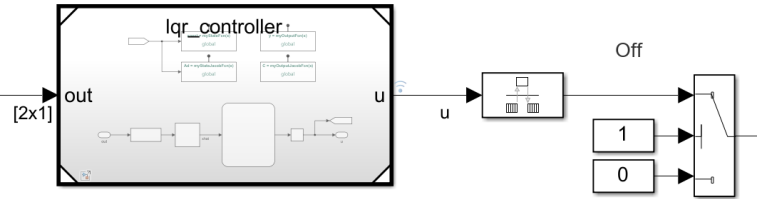
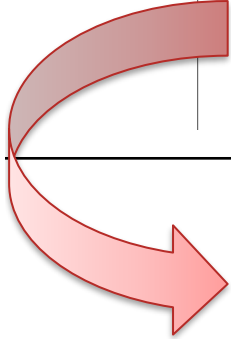
【参考】 離散化手法は何を使えばいい？

- PID制御や最適レギュレータ制御則の場合は、シンプルな手法でよい。
 - 積分値、微分値が真値と異なっていたことによる制御器の性能の変化は、現実のプラントモデルをモデル化した際のモデル化誤差で引き起こされる性能の変化と比較した場合、ほとんどの場合でモデル化誤差の影響の方が大きい。
 - 従って、離散化誤差はモデル化誤差に埋もれて、全く目立たなくなるのである。
- ローパスフィルターなどのフィルターアルゴリズムの場合は、離散化の手法によって性能が変わるため、適切な手法をよく検討する必要がある。

アジェンダ

- 背景
- 機能の抽象化レイヤー
- アルゴリズムの離散化
- コード生成と実装

レイヤーを超えるための「コード生成」

レイヤー	技術領域	項目
アルゴリズム	制御工学	$\dot{x} = Ax + Bu$ のとき、操作量は $u = -R^{-1}B^T Px$
Simulinkモデル	モデリング	
C/C++コード	プログラミング	<div style="display: flex; align-items: center;"> <div style="writing-mode: vertical-rl; transform: rotate(180deg); color: red; font-weight: bold; margin-right: 10px;">コード生成</div>  <div style="background-color: black; color: white; padding: 10px; font-family: monospace;"> <pre> if (std::isnan(lqr_controller lambda = (rtNaNF); } else if (std::isinf(lqr_con lambda = (rtNaNF); } else if (lqr_controller_dep lambda = 0.05; </pre> </div> </div>

コード生成のメリット

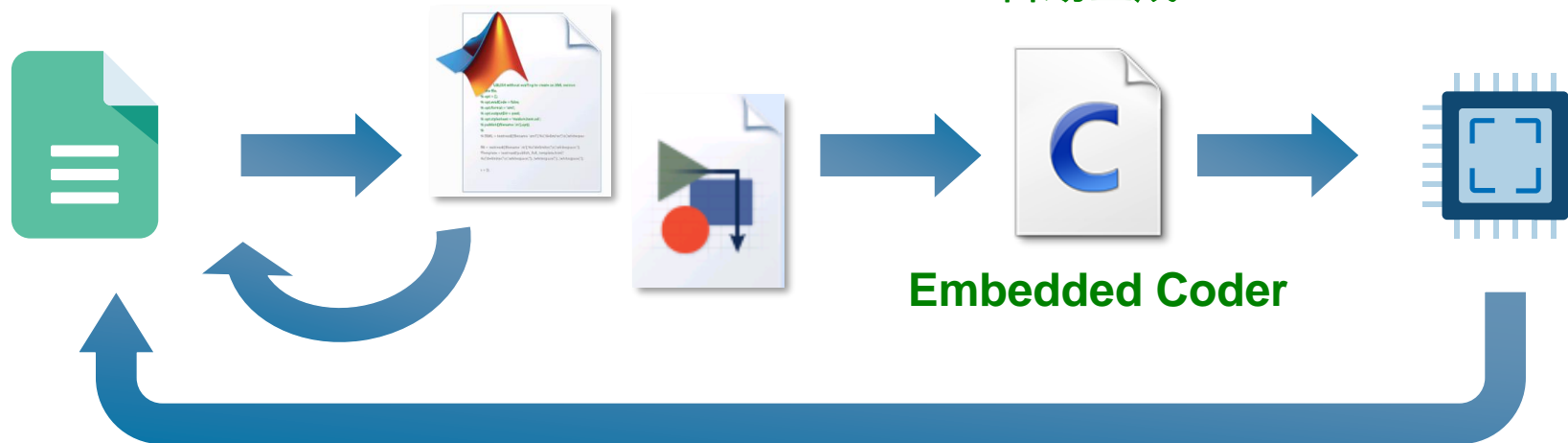
モデル→ハンドコーディングで発生する
コーディング時間を短縮、誤解釈混入リスクを解消、モデル・コード間の乖離を防止

アルゴリズムや
要求仕様

Simulinkモデル
MATLABプログラム

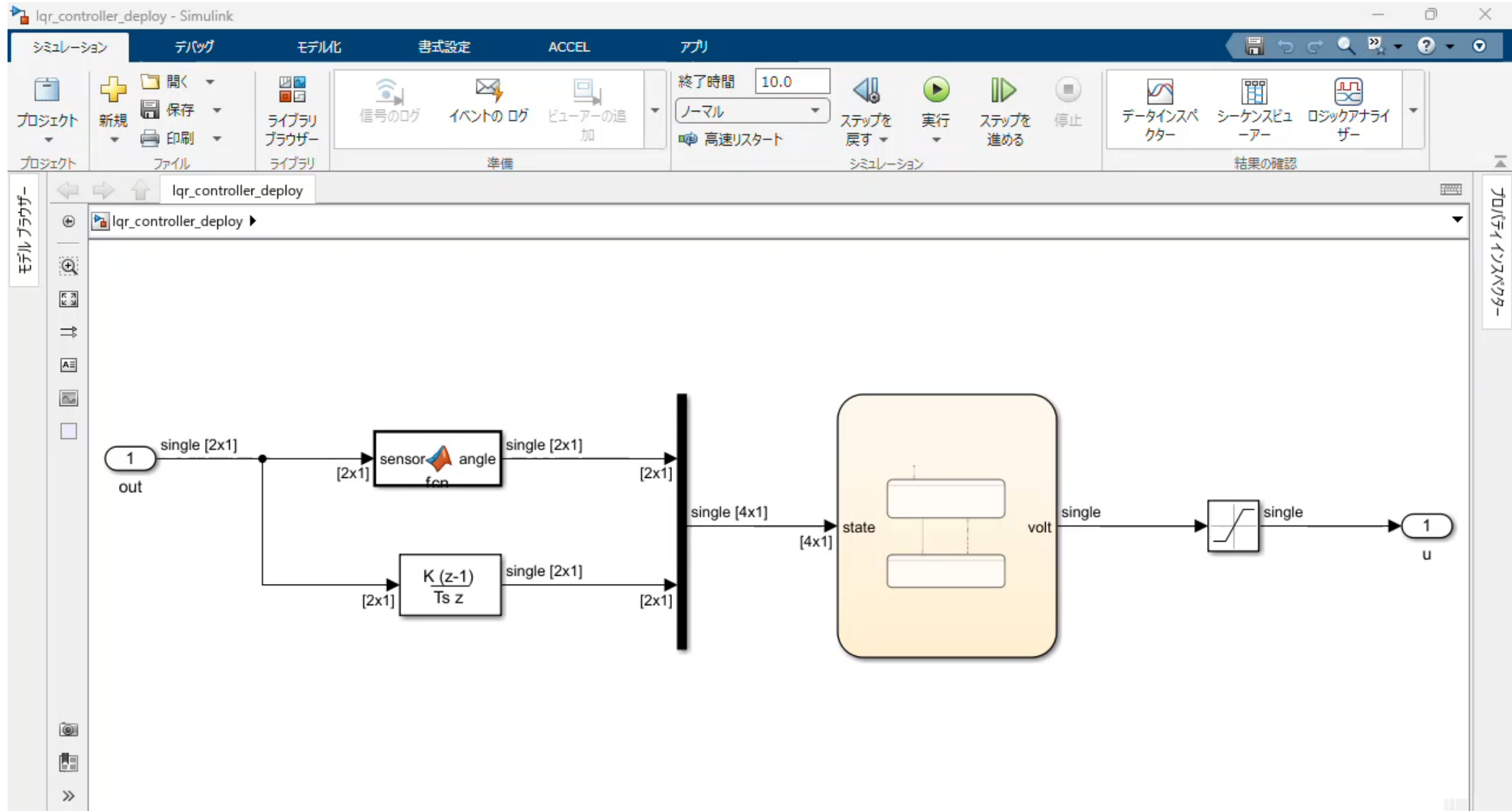
C/C++コードを
自動生成

マイコン/ECU



モデル成果物をシームレスに実装することで開発の修正ループを高速化

SimulinkモデルからC/C++コードを生成



生成したコードの利用方法

周期/非周期処理からモデルロジック実行関数を呼び出す
(割り込みルーチンやRTOSタスク等)

ソフト初期化
のタイミング
で初期化関数
を呼び出す

ソフトウェア

```
Main()
{
    ADC_Init();
    Serial_Init();
    PWM_Init();
    Controller_initialize();
    while(1) { }
}
```

```
InterruptServiceRoutine_20ms()
{
    ADC_Read();
    Serial_Read();
    Controller_step();
    PWM_Write();
}
```

モデル



モデル生成コードにはモデルロジック実行関数と初期化関数が含まれる

初期化処理の記述

```
void setup() {  
  pinMode(LED_BUILTIN, OUTPUT);  
  TimeManager::start_timer(TC1, 0, TC3_IRQn,  
    ..... TimeManager::SAMPLING_TIME_STEP_MSEC);  
  invert_pendulum_controller.initialize();  
  pinMode(SS0PIN, OUTPUT);  
  digitalWrite(SS0PIN, HIGH);  
  SPI.begin();  
}
```

← 初期化コマンド

setup()関数は、マイコン実行開始してから最初に1度だけ実行される。

タイマー割込み処理の記述

```
void TC3_Handler() {
    TC_GetStatus(TC1, 0);

    static Quanser_Info previous_sensor_data;

    double now_time = TimeManager::get_time();

    if (((int)now_time % 2) == 1) {
        digitalWrite(LED_BUILTIN, LOW);
    } else {
        digitalWrite(LED_BUILTIN, HIGH);
    }

    /* sensor input */
    float input_for_controller[2] = {};
    input_for_controller[0] = previous_sensor_data.theta;
    input_for_controller[1] = previous_sensor_data.phi;

    /* control */ 入力、制御実行、出力
    invert_pendulum_controller.setout(input_for_controller);
    invert_pendulum_controller.step();
    double volt = invert_pendulum_controller.getu();

    /* Protect volt input */
    if (now_time >= EXPERIMENT_END_TIME) {
        volt = 0.0;
    }
}
```

```
/* Convert */
uint8_t SPI_data[OUTPUT_DATA_SIZE] = {0};
SetEncoderSignal_Quanser.create_set_encoder_signal(now_time);
SPICommunicator.convert_volt_to_SPI(
    volt, SetEncoderSignal_Quanser.set_encoder_0,
    SetEncoderSignal_Quanser.set_encoder_1, SPI_data);

/* Communicate with SPI */
SPI.beginTransaction(spiB);
digitalWrite(SS0PIN, LOW);
SPI.transfer(SPI_data, OUTPUT_DATA_SIZE);
digitalWrite(SS0PIN, HIGH);

/* Get received data */
Quanser_Info quanser_info;
SPICommunicator.convert_data_to_Quanser_Info(SPI_data, &quanser_info);

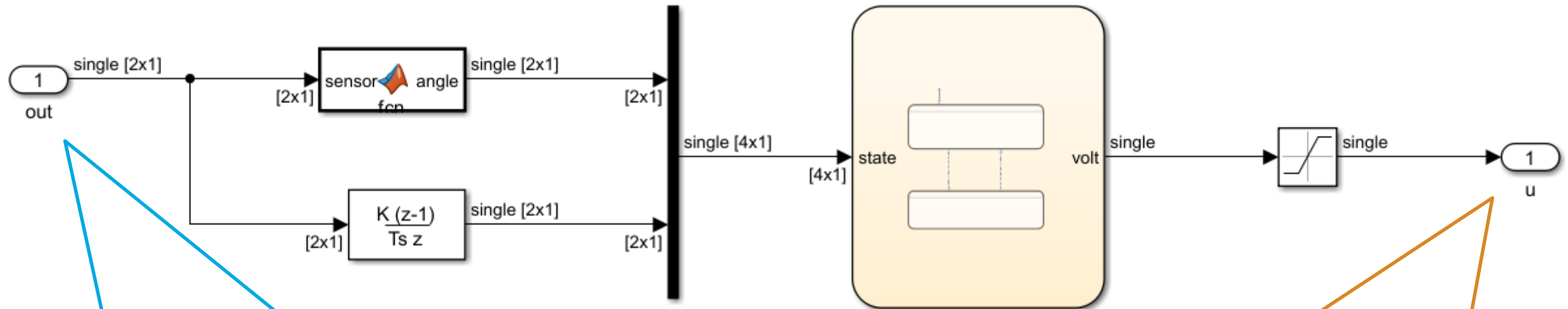
previous_sensor_data.theta = quanser_info.theta;
previous_sensor_data.phi = quanser_info.phi;
previous_sensor_data.status = quanser_info.status;
previous_sensor_data.current = quanser_info.current;

/* Count up time */
TimeManager::count_time();
}
```

信号の変換

Simulinkの入出力信号に対応するコード

実装する制御モデルの最上位階層



```
struct ExtU_lqr_controller_deploy_T {
    float out[2]; // '<Root>/out'
};

ExtU_lqr_controller_deploy_T lqr_controller_deploy_U;

void lqr_controller_deploy::setout(float localArgInput[2])
{
    lqr_controller_deploy_U.out[0] = localArgInput[0];
    lqr_controller_deploy_U.out[1] = localArgInput[1];
}
```

```
struct ExtY_lqr_controller_deploy_T {
    float u; // '<Root>/u'
};

ExtY_lqr_controller_deploy_T lqr_controller_deploy_Y;

float lqr_controller_deploy::getu() const
{
    return lqr_controller_deploy_Y.u;
}
```

【参考】 データの渡し方について

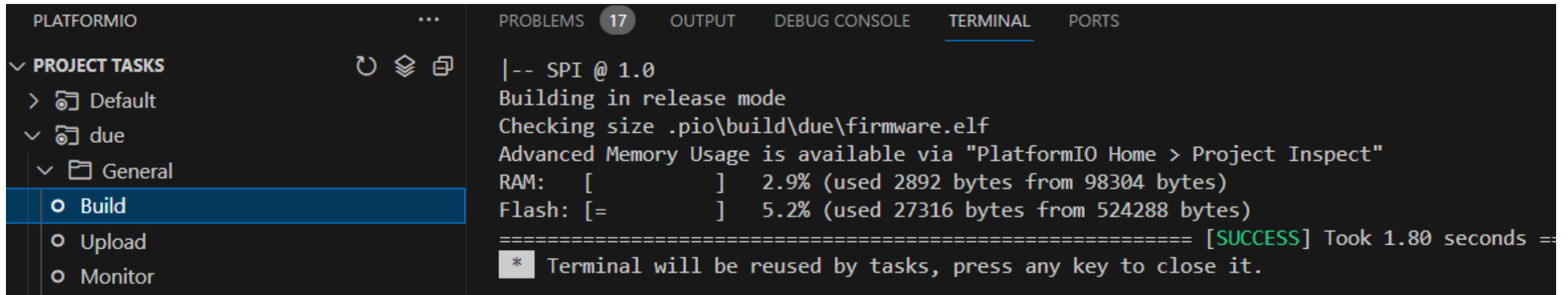
- モデルの出力が複数ある場合は、ポインタで渡す関数が生成される場合もある

```
void my_func(double *a, double *b) {  
    *a = 1.0;  
    *b = 2.0;  
}
```

```
void main(void) {  
    double a = 0.0;  
    double b = 0.0;  
  
    my_func(&a, &b);  
  
    // aが1.0、bが2.0になっている。  
}
```

← 出力を受け取るための変数を定義し、
my_funcの引数にその変数アドレスを渡し、
計算結果を受け取る形となる。

ビルドをして実行ファイルをマイコンに書き込む



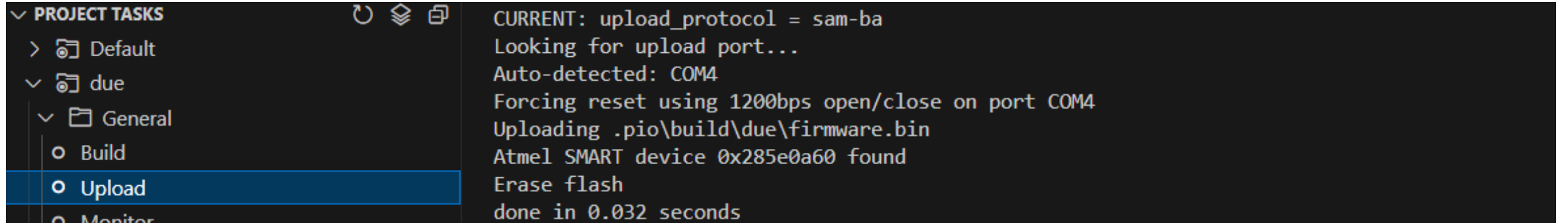
PLATFORMIO ...

PROJECT TASKS

- Default
- due
 - General
 - Build**
 - Upload
 - Monitor

PROBLEMS 17 OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```
-- SPI @ 1.0
Building in release mode
Checking size .pio\build\due\firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM: [          ] 2.9% (used 2892 bytes from 98304 bytes)
Flash: [=         ] 5.2% (used 27316 bytes from 524288 bytes)
===== [SUCCESS] Took 1.80 seconds =====
* Terminal will be reused by tasks, press any key to close it.
```

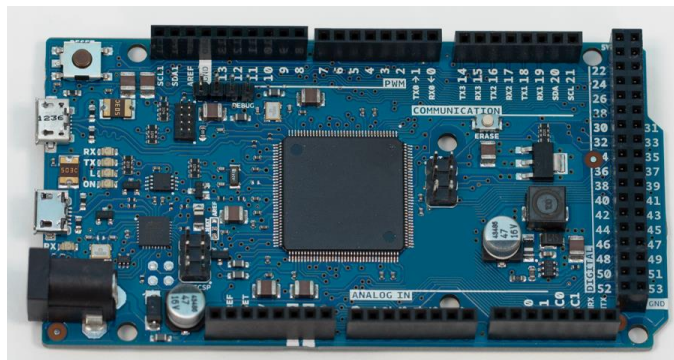


PROJECT TASKS

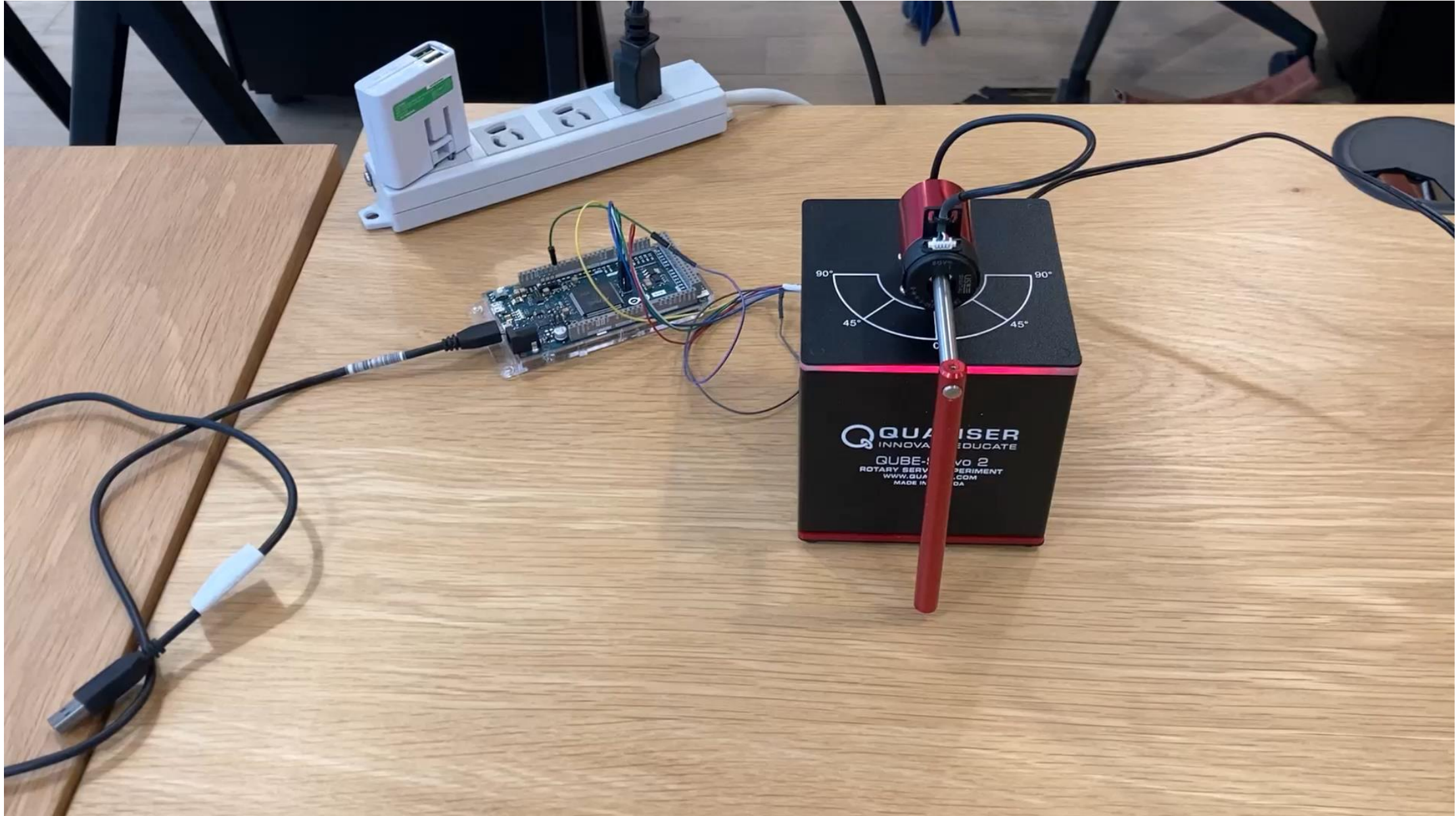
- Default
- due
 - General
 - Build
 - Upload**
 - Monitor

CURRENT: upload_protocol = sam-ba
Looking for upload port...
Auto-detected: COM4
Forcing reset using 1200bps open/close on port COM4
Uploading .pio\build\due\firmware.bin
Atmel SMART device 0x285e0a60 found
Erase flash
done in 0.032 seconds

書き込み



制御機能を実機に実装し、制御を実行



モデリング・コード生成ワークフローの良い点

- モデリングのレイヤーで機能を作り込めるため、C言語の難しい仕様に思考のリソースを取られないで済む
- モデルで作った制御機能は汎用的なコードとなり、ハードウェア依存のコードと切り離され、影響を受けない
- アルゴリズムのコア部分をモデル化することで、他の開発プロジェクトにも流用できる

モデリング・コード生成ワークフローの良い点

Simulink

制御モデル

プラントモデル

Arduinoに実装するコード全体

モデルから生成されたコード

データの受け渡し処理

デバイス操作の処理

SPI通信

タイマー割込み

まとめ

まとめ

- 機能の抽象化軸に沿ってレイヤーを分けることで、複雑なシステムを見通し良く、効率的に設計できる
- Simulinkのモデルはアルゴリズムの具体化と検証に役立ち、コード生成して実機に実装できる
- 制御アルゴリズムをデジタルのコントローラーに実装するには、離散化とリアルタイム制御が重要



Accelerating the pace of engineering and science

© 2024 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

參考資料

初心者のための Embedded Coder : 設定Tipsとワークフローをまとめて解説

<https://blogs.mathworks.com/japan-community/2024/04/17/embedded-coder-for-beginner/>

目次:

1. コード生成とは
 - 1.1. コード生成のありがたみ
 - 1.2. Embedded Coder の特徴
2. 今回の目標
 - 2.1. 振子の振れ止め制御
 - 2.2. Arduino 開発環境
 - 2.3. タイマー割込みを設計する
3. Simulink モデル構築
 - 3.1. コード生成のための Simulink モデリング
 - 3.2. コンフィギュレーションパラメーター設定方法
 - 3.3. Simulink モデルから生成されたコードをライブラリ的に使いたい場合
 - 3.4. Embedded Coder 設定 Tips
4. 実装と実機試験
 - 4.1. Arduino 開発環境へ統合
 - 4.2. 実機試験
5. まとめ

1. コード生成とは

このセクションでは、コード生成とは何かについて説明します。

1.1. コード生成のありがたみ

モデルベースデザイン（モデルベース開発、MBD）は、制御開発の上流工程において、コントローラーに実装する制御モデルと、制御対象となる物理システムモデルを組み合わせた全体システムをシミュレーションで表現し、ふるまいを検証する手法です。詳細については、[こちらのページ](#)をご確認ください。

モデルベースデザインの開発ワークフローは、以下の「V字モデル」で表現されます。



Embedded Coderの使い方について、初心者にも分かりやすく解説しています。