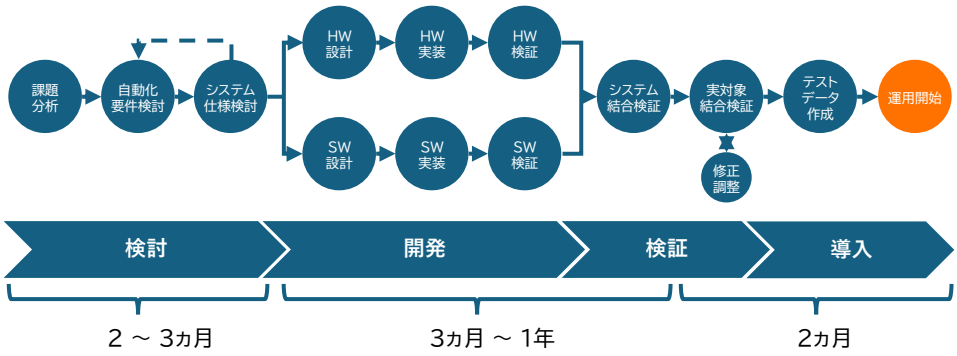
AUTOmealなら

自動テスト環境を1Weekで構築

一般的な自動テスト環境の構築と導入

自動テスト環境を一から開発して導入しようとすると、**自動テスト環境の運用開始までに数か月以上**の時間が必要になります。また、導入以降でも運用面の課題が盛りだくさん。

一般的な導入の流れ

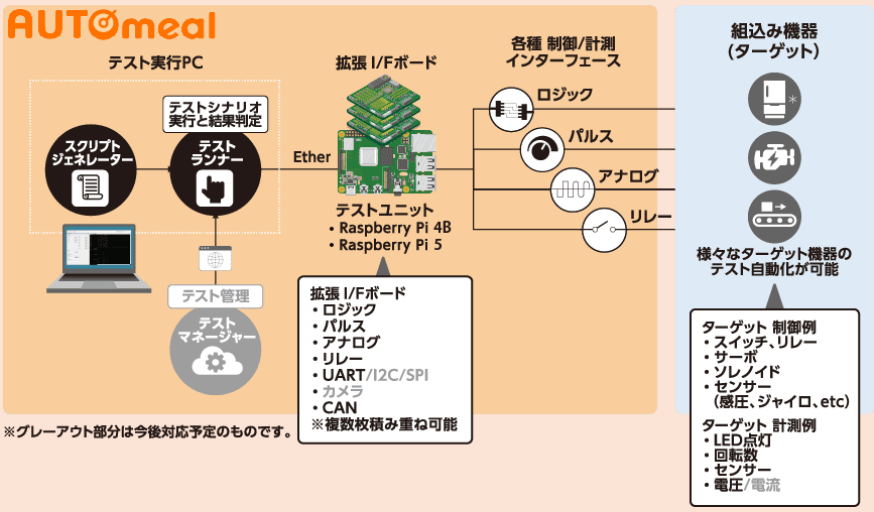


さらに運用課題も	長期運用しないと費用対効果を十分に得られない	テスト自動化環境自体の品質	自動化環境にも定期的にメンテナンスが必要

AUTOmealを使った自動テスト環境の導入

AUTOmealは、お手持ちのラズパイを活用して自動テスト環境を構築します。
環境開発や検証の時間が要らないため、**自動テスト環境を1週間ほどで構築・導入**可能。
その仕組みから、**運用におけるメンテナンスも簡単**です。

ハード環境構築はテスト対象機器のI/Fに適合するHATボードをRaspberry Piに積載して配線するだけ。あとはAUTOmealのWindowsアプリケーションがRaspberry Piを制御してターゲットへのテストを自動実行します。



AUTOmealなら
ノーコードで自動テストを実行

一般的な自動テストのテストスクリプト作成とテスト実行

エクセルなどで管理されているテストシナリオを読み込んで、テスト自動実行のためのテストスクリプトを設計し、手動でコーディングします。
シナリオからテストを設計する能力や、それをコーディングするスキルが必要です。

step 1
テスト手順を基に
テストスクリプト
を設計

step 2
テストスクリプト
のコーディング
デバッグ

step 3
テストスクリプト
の実行環境を構築

step 4
目視による
挙動の正誤判定

スキルやリソースが必要

AUTOmealを使ったテストスクリプト作成とテスト実行

エクセル形式で用意されたフォーマットにそってテストシナリオを記述するだけ。
AUTOmealに読み込ませるとPython形式のテストスクリプトに自動変換されて出力されます。スクリプトはもちろん手動で編集することも可能です。

step 1
テストを基に
エクセル形式の
フォーマットを記入

テストシナリオ1			
No	Operation	Comment	Repeat
1	開始		
2	待機		
3	制御		
4	制御		
5	計測		
6	制御		
7	制御		
8	計測		
9	終了		

step 2
AUTOmealで
Python形式の
テストスクリプトに変換

```
import lib.test_manager as tm

def main():
    try:
        res = tm.push("Init")
        res = tm.send_rs232c(["Sample1"])

        # ToDo : 任意のエラー判定
        if not res[0] is 0x00:
            raise Exception("Error")

    except Exception as e:
        tm.push("ALL '0'")

if __name__ == '__main__':
    main()
```

step 3
AUTOmealで
自動テストを実施

スキルやリソースが不要

Pythonスクリプトを手動で編集可
なのでちょっとしたメンテナンスも簡単

AUTOmealなら

CAN通信対応！

シリアル通信のテストを自動化

一般的なシリアル通信のテスト

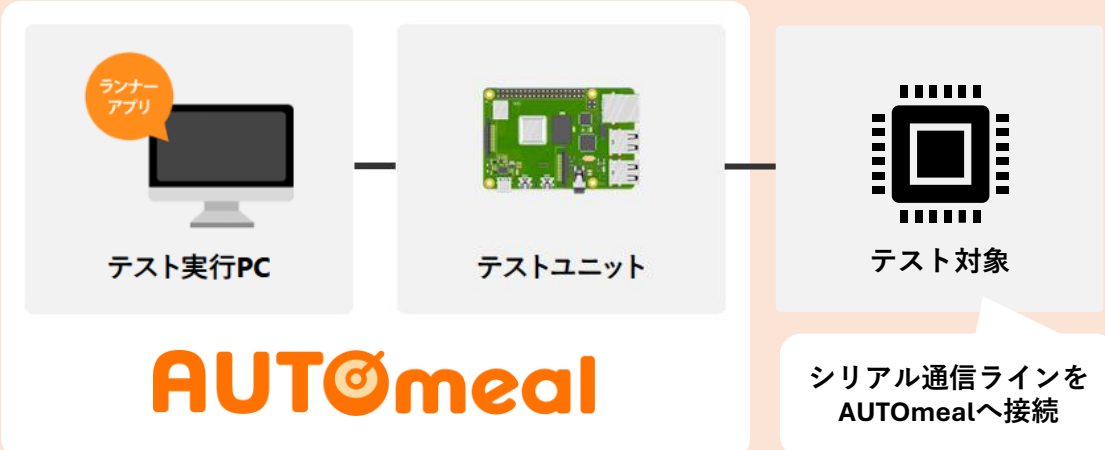
以下のような方法がありますが、作業者には負担がかかり、テストスキル起因の属人化が発生することも。

RAMモニタで	実際に通信を行い、正しく処理されたかどうかをメモリの内容から判定	通信内容とRAM内容の正しい把握したうえでの目視確認が必要
オシロやラインモニタで	シリアル通信ラインから電気的なHi/Loを取得して判定	プロービングのためのハード的なスキル、およびツールを使った目視確認が必要
シミュレーターで	シミュレーターをターゲット基板に接続して実際に通信させて判定	シミュレーター自体やスタブを作製できるエンジニアが必要

AUTOmealを使ったシリアル通信のテスト

AUTOmealがシリアル通信相手に成り代わることでシリアル通信のテストを実行します。
手動による通信制御は必要なく、自動でテストの実行から結果の判定までを行います。

AUTOmealで	AUTOmealをターゲット基板に接続して実際に通信させて判定する	AUTOmealによるテストスクリプト生成と自動テストの実行



DT+による

実機デバッグや動作検証の効率化

一般的な動作検証の手法

実機上でのデバッグや動作検証を2種類に大別するならば、print文デバッグに代表されるようなログデバッグか、デバッガによるステップ実行でしょう。
それぞれに向き不向きがあります。

ログデバック

■ print文をコンソール等で確認

○ 処理の流れを把握しやすい
○ 止められない箇所などでも使える
× ログの準備・解析の手間が多い

ステップ実行

■ JTAG、開発環境上の機能

○ 1ステップずつ確認でき、
変数などの状況も分かりやすい
× 処理を止められない場合は使えない

動的テストによる動作検証の手法

動的テスト＝print文デバッグのような「ログ解析による挙動解析」をリッチにしたもの
デバッグのみならず、パフォーマンス測定やカバレッジ計測も行えます。

step 1

ソースコードに
ログ出力処理を埋込

step 2

Runさせてログ収集

step 3

ログ分析で
ソフト挙動を解析

動的テストツールDT+でできること

DT+は、ログを取得すればワンクリックで以下のような解析を実施できます。
エンジニアのスキルに依存せず、手間を大きく削減したデバッグや動作検証が可能です。

✓ 不具合の原因解析	✓ パフォーマンスの測定	✓ テスト時のカバレッジの確認
✓ ベースコードの解析	✓ 設計値の変化の可視化	✓ 通信ラインの可視化
✓ タスクやスレッドの挙動把握	✓ 変数値の変化の可視化	✓ SWログとHW波形の同時解析
✓ 複数コアの動作を一挙に可視化	✓ 複数CPUの動作確認	✓ 映像との同期