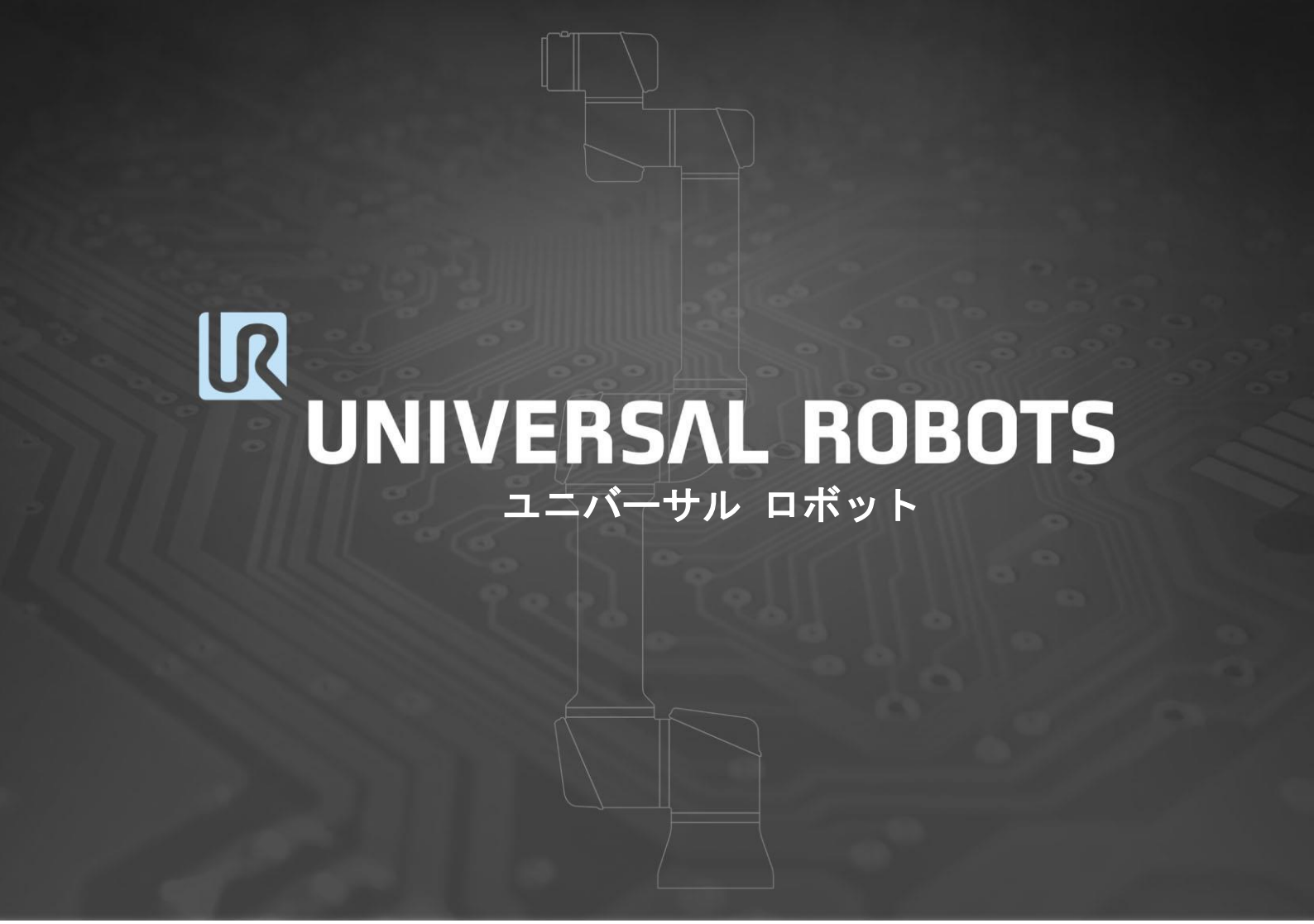

URScript プログラミング言語

バージョン 5.2
2018 年 11 月 21 日

デンマークで設計およびビルド

目次

1 URScript プログラミング言語

1.1 概要

ユニバーサル ロボットは 2 つのレベルで制御できます。

- ●PolyScope またはグラフィック ユーザー インターフェース レベル
- ●スクリプトレベル

スクリプトレベルにおいて、ロボットを制御するプログラミング言語は **URScript** です。**URScript** には変数、型、およびフロー制御ステートメントが含まれます。I/O およびロボットの動作を監視および制御する組み込みの変数および関数もあります。

1.2 URControl への接続

URControl は、コントロール ボックスのミニ ITX PC で実行されている低レベル ロボット コントローラーです。PC の起動時に、URControl がデーモンとして起動し、ローカル TCP/IP 接続を使用して PolyScope またはグラフィック ユーザー インターフェースがクライアントとして接続されます。

（別の PC で実行される）クライアント アプリケーションを作成し、TCP/IP ソケットを使用して URControl に接続することで、*スクリプト レベル*でのロボットのプログラミングを行います。

- ●**ホスト名**：ur-xx（または、ロボットが DNS 内にない場合は PolyScope の **About Dialog-Box** にある IP アドレス）。
- ●**port**：30002

接続が確立されると、ソケット上で URScript プログラムまたはコマンドがクリア テキストで送信されます。各行は「\n」で終了します。テキストは拡張 ASCII 文字のみで構成されることに注意してください。

URControl が正しくスクリプトを解釈するようにするためには、以下の条件を満たす必要があります。

- ●スクリプトは、1 列目に関数定義またはセカンダリ関数定義（「`def`」または「`sec`」キーワード）を配置して始める必要があります。
- ●スクリプトの他のすべての行は、少なくとも 1 つのホワイト スペースでインデントする必要があります。
- ●スクリプトの最後の行は、1 列目から「`end`」キーワードである必要があります。

1.3 数値、変数、および型

URScript では、演算式構文の標準は以下のとおりです。

```
1+2-3
4*5/6
(1+2)*3/(4-5)
"Hello" + ", " + "World!"
```

ブール式では、ブール演算子が使用されます。

```
True or False and (1 == 2)
1 > 2 or 3 != 4 xor 5 < -6
not 42 >= 87 and 87 <= 42
"Hello" != "World" and "abc" == "abc"
```

変数の代入には、イコール記号 = が使用されます。

```
foo = 42
bar = False or True and not False
baz = 87-13/3.1415
hello = "Hello, World!"
l = [1,2,4]
target = p[0.4,0.4,0.0,0.0,3.14159,0.0]
```

変数の基本型は、変数の最初の代入から推測されます。上記の例では、`foo` は `int` 型であり、`bar` は `bool` 型です。`target` は位置と方向の組み合わせである pose 形です。

基本型は以下のとおりです。

- `none`
- `bool`
- `number` - `int` または `float`
- `pose`
- `string`

ポーズは `p[`*x*`,`*y*`,`*z*`,`*ax*`,`*ay*`,`*az*`]` で指定されます。*x*、*y*、*z* は TCP の位置であり、*ax*、*ay*、*az* は軸角度表記で指定される TCP の方向です。

文字列は基本的にバイト配列であり、含まれている文字に使用されているエンコーディングについては考慮されないことに注意してください。そのため、文字を操作するように見える一部の文字列関数（例：*str_len*）は実際にはバイトを操作しており、文字列にマルチバイトまたは可変長文字のシーケンスが含まれている場合、結果は期待しているものと一致しない場合があります。詳細については、それぞれの関数の説明を参照してください。

1.4 制御フロー

プログラムの制御フローは `if` ステートメントによって変化します。

```
if a > 3:
  a = a + 1
elif b < 7:
  b = b * a
else:
  a = a + b
end
```

また、`while` ループによっても変化します。

```
l = [1,2,3,4,5]
i = 0
while i < 5:
  l[i] = l[i]*2
  i = i + 1
end
```

`break` を使用するとループを完全に停止することができ、`continue` を使用すると一番近い閉ループの次のイテレーションに制御を渡すことができます。

1.4.1 特別なキーワード

- `halt` はプログラムを中止します。
- `return` は関数から戻ります。

1.5 関数

関数は以下のように宣言します。

```
def add(a, b):
  return a+b
end
```

関数は以下のように呼び出すことができます。

```
result = add(1, 4)
```

また、関数の引数のデフォルト値を指定することもできます。

```
def add(a=0,b=0):
  return a+b
end
```

宣言時にデフォルト値を指定する場合、引数を入力することも、以下のようにスキップすることもできます。

```
result = add(0,0)
result = add()
```

関数の呼び出し時は、引数を宣言した順番を遵守することが重要です。順番が定義時と異なると、関数は期待どおりに動作しません。

引数は値（配列を含む）でのみ渡すことができます。すなわち、関数のスコープ内で引数の内容に対して行われた変更は、そのスコープ外では反映されません。

```
def myProg()

  a = [50,100]

  fun(a)

  def fun(p1):
    p1[0] = 25
    assert(p1[0] == 25)
    ...
  end

  assert(a[0] == 50)
  ...
end
```

また、URScript では名前付き引数もサポートされます。

1.6 リモート プロシージャー コール（RPC）

リモート プロシージャー コール（RPC）は、通常の関数呼び出しと同様ですが、関数がリモートで定義および実行されるという点で異なります。リモート サイトでは、呼び出される **RPC** 関数が同じ数のパラメータおよび対応する型で（関数のシグネチャと共に）存在している必要があります。関数がリモートで定義されていない場合、プログラムの実行が停止されます。コントローラーは、XMLRPC 標準を使用してリモート サイトにパラメータを送信し、結果を取得します。**RPC** コール中、コントローラーはリモート関数が完了するのを待ちます。XMLRPC 標準は、C++（xmlrpc-c ライブラリ）、Python、Java などでサポートされています。

カメラを初期化する URScript プログラムを作成すると、スナップショットが撮影され、新しいターゲット ポーズを取得します。

```
  camera = rpc_factory("xmlrpc", "http://127.0.0.1/RPC2")
  if (! camera.initialize("RGB")):
    popup("Camera was not initialized")
  camera.takeSnapshot()
```

```
  target = camera.getTarget()
  ...
```

最初に、rpc_factory(「インターフェース」セクションを参照)は指定された**リモート** サーバーとの XMLRPC 接続を作成します。camera 変数はリモート関数呼び出しのハンドルです。カメラを初期化する必要があるため、camera.initialize("RGB")を呼び出します。関数は、リクエストが成功したかどうかを示すブール値を返します。ターゲット位置を見つけるために、カメラはまず camera.takeSnapshot()を呼び出して写真を取ります。スナップショットが撮影されると、リモート サイトでの画像分析によって、ターゲットの位置が計算されます。プログラムは、関数呼び出し target = camera.getTarget()によって正確なターゲット位置を求めます。リターン時に、target 変数が結果に割り当てられます。camera.initialize("RGB")、takeSnapshot()、および getTarget()関数は、RPC サーバーの担当です。

テクニカル サポート ウェブサイトには、さらに多くの XMLRPC サーバーの例が含まれています。

1.7 スコープのルール

URScript プログラムは、パラメータのない関数として宣言されます。

```
def myProg():

end
```

プログラム内で宣言されたすべての変数にはスコープがあります。スコープは変数が直接アクセスできるテキスト上の領域です。この可視性を変更するために、2 つの修飾子を利用できます。

- local 修飾子は、同じ名前のグローバル変数が存在する場合でも、本当にローカルなものとして関数内の変数を扱うようにコントローラーに指示します。
- global 修飾子は、関数内で宣言された変数がグローバルにアクセスできるように強制します。

各変数について、コントローラーは、変数がグローバルなのかローカルなのかなど、スコープ バインディングを決定します。local または global 修飾子が指定されていない場合(フリー変数とも呼ばれます)、コントローラーはまずグローバルに変数を検索し、見つからなかった場合は変数がローカルとして扱われます。

以下の例では、1 つ目の a がグローバル変数であり、2 つ目の a がローカル変数です。同じ名前であっても、両方の変数は独立しています。

```
def myProg():

  global a = 0

  def myFun():
    local a = 1
    ...
  end
```

```
  ...
end
```

ローカル変数が同じ名前のグローバル変数をマスクするため、関数内からグローバル変数にアクセスできなくなることに注意してください。

以下の例では、1 つ目の a がグローバル変数であり、関数内の変数はプログラム内で宣言されたものと同じ変数です。

```
def myProg():

  global a = 0

  def myFun():
    a = 1
    ...
  end
  ...
end
```

それぞれのネストされた関数では、同じスコープ バインディング ルールが保持されます。以下の例では、1 つ目の a がグローバルに定義され、2 つ目はローカルであり、3 つ目は暗黙的にグローバルとなります。

```
def myProg():

  global a = 0

  def myFun():
    local a = 1

    def myFun2():
      a = 2
      ...
    end
    ...
  end
  ...
end
```

1 つ目と 3 つ目の a は全く同一のものであり、2 つ目の a は独立しています。

global 修飾子がない、または local 修飾子が使用されている場合でも、最初のスコープレベル（最初のインデント）の変数はグローバルとして扱われます。

```
def myProg():

  a = 0

  def myFun():
    a = 1
    ...
```

```
  end
  ...
end
```

変数 a は全く同一のものです。

1.8 スレッド

スレッドはいくつかの特別なコマンドでサポートされます。

新しいスレッドを宣言するには、関数の宣言と同様の構文を使用します。

```
thread myThread():
  # Do some stuff
  return False
end
```

注意すべき点がいくつかあります。まず、スレッドにはパラメータを設定できず、宣言時のかっこは空である必要があります。次に、スレッドでは return ステートメントが許可されていますが、返される値は破棄され、スレッド外からアクセスすることはできません。関数に他の関数を含めることができるのと同様に、スレッドに他のスレッドを含めることができます。言い換えると、スレッドはネストすることができ、スレッド階層を形成することができます。

スレッドを実行するには、以下の構文を使用します。

```
thread myThread():
  # Do some stuff
  return False
end

thrd = run myThread()
```

run コマンドによって返される値は、実行中のスレッドに対するハンドルです。このハンドルは、実行中のスレッドを操作するのに使用できます。run コマンドは新しいスレッドから生成され、run 命令に続く命令を実行します。

スレッドは自身によって生成された実行中のスレッドのみを待機できます。実行中のスレッドが終了するのを待つには、join コマンドを使用します。

```
thread myThread():
  # Do some stuff
  return False
end
```

```
thrd = run myThread()

join thrd
```

これは、指定されたスレッドの実行が終了するまで、呼び出し中のスレッドの実行を停止します。スレッドがすでに終了している場合、ステートメントに効果はありません。

実行中のスレッドを強制終了するには、`kill` コマンドを使用します。

```
thread myThread():
  # Do some stuff
  return False
end

thrd = run myThread()

kill thrd
```

kill を呼び出すと、スレッドが停止し、スレッド ハンドルが有効ではなくなります。スレッドに子スレッドが存在する場合、これらも同様に強制終了されます。

競合状態およびその他のスレッド関連の問題を防ぐため、クリティカル セクションのサポートが提供されています。クリティカル セクションを使用すると、別のスレッドが実行を開始する前に、囲まれたコードが実行を終了することが保証されます。クリティカル セクションのスコープ内に時間依存のコマンドが存在しない限り、前のステートメントは常に true です。そのような場合、別のスレッドの実行が許可されます。時間依存コマンドには、sleep、sync、move コマンド、および socketRead が含まれます。そのため、クリティカル セクションは可能な限り短くしておくことが重要です。構文は以下のようになります。

```
thread myThread():
  enter_critical
  # Do some stuff
  exit_critical
  return False
end
```

1.8.1 スレッドとスコープ

スレッドのスコープのルールは関数で使用されるものとまったく同じです。これらのルールの詳細については、1.7 を参照してください。

1.8.2 スレッド スケジューリング

URScript スクリプト言語の第 1 目的はロボットを制御することであるため、スケジューリング ポリシーは主にこのタスクのリアルタイムの要求に基づいています。

ロボットは 500Hz の周波数で制御する必要があります。すなわち、0.002 秒ごとに実行することを指示する必要があります（各 0.002 秒の期間はフレームと呼ばれます）。これを実現するため、各スレッドでは 0.002 秒の「物理的な」（またはロボットの）タイム スライスが使用され、実行可能な状態にあるすべてのスレッドがラウンド ロビン[1]方式でスケジュールされます。

スレッドがスケジュールされるたびに、（ロボットを制御する命令を実行することで）そのタイム スライスの一部を使用したり、ロボットを制御しない命令を実行して「物理的な」時間を使用しなかったりすることができます。スレッドがタイム スライス全体を使い果たすと、そのスレッドは実行不可状態になり、次のフレームが開始されるまで実行が許可されません。スレッドがフレーム内にそのタイム スライスを使用しない場合、フレームが終了する前に実行不可状態に切り替わることが予想されます[2]。この状態の切り替わりの理由は、join 命令によるものであったり、単にスレッドの終了によるものであったりします。

`sleep` 命令でロボットを制御しない場合でも、「物理的な」時間が使用されることに注意する必要があります。`sync` 命令でも同様です。

1.9 プログラム ラベル

最初の記号が「`$`」のプログラム ラベル コード行は、PolyScope によって生成された、プログラムの実行を追跡できるようにする特別な行です。

```
$ 2 "var_1= True "
global var_1= True
```

[1]各フレームが開始される前に、残りのタイム スライスが最大のスレッドが最初にスケジュールされるように、スレッドがソートされます。

[2]この予想が満たされない場合、プログラムが停止します。

2 運動モジュール

2.1 関数

conveyor_pulse_decode(*type, A, B*)

*廃止予定：*ロボット コントローラーにデジタル入力数値 A と B をコンベア エンコーダーのパルスとして扱うよう指示します。デジタル入力 0、1、2、または 3 のみを使用できます。

パラメータ

`type`：A と B の入力を扱う方法を決定する整数

0 はエンコーダーなしであり、パルス デコーディングを無効化します。

1 はクワドラチャ エンコーダーであり、入力 A および B はオフセットが 90 度の矩形波である必要があります。コンベアの方向も決定できます。

2 は信号入力（A）の立ち上がりおよび立ち下がりエッジです。

3 は信号入力（A）の立ち上がりエッジです。

4 は信号入力（A）の立ち下がりエッジです。

コントローラーは最大 40kHz で入力をデコードできます。

`A`： エンコーダー入力 A、0～3 の値はデジタル入力 0～3 です。

`B`： エンコーダー入力 B、0～3 の値はデジタル入力 0～3 です。

廃止予定：この関数は `encoder_enable_pulse_decode` と置き換えられるため、今後は使用しないでください。

```
>>> conveyor_pulse_decode(1,0,1)
```

この例は、入力 A＝digital_in[0]および入力 B＝digital_in[1]でデコードするようにクワドラチャ パルスをセットアップする方法を示しています。

```
>>> conveyor_pulse_decode(2,3)
```

この例は、入力 A＝digital_in[3]でデコードするように立ち上がりおよび立ち下がりエッジ パルスをセットアップする方法を示しています。パラメータ B は設定する必要がないことに注意してください（使用されないため）。

コマンドの例：`conveyor_pulse_decode(1, 2, 3)`

●パラメータの例：

- type＝1 →クワドラチャ エンコーダーであり、入力 A および B はオフセットが 90 度の矩形波である必要があります。コンベアの方向も決定できます。
- A＝2 →エンコーダー出力 A はデジタル入力 2 に接続されます。
- B＝3 →エンコーダー出力 B はデジタル入力 3 に接続されます。

encoder_enable_pulse_decode(*encoder_index, decoder_type, A, B*)

コントローラーのパルス デコーダーに接続されたエンコーダーをセットアップします。

```
>>> encoder_enable_pulse_decode(0,0,1,8,9)
```

この例は、ピン 8 および 9 に接続されたクワドラチャ信号をデコードするようにエンコーダー0 をセットアップする方法を示しています。

パラメータ

`encoder_index`： 定義するエンコーダーのインデックス。0 または 1 である必要があります。

`decoder_type`： A と B の入力を扱う方法を決定する整数

0 はエンコーダーなしであり、パルス デコーディングを無効化します。

1 はクワドラチャ エンコーダーであり、入力 A および B はオフセットが 90 度の矩形波である必要があります。コンベアの方向も決定できます。

2 は信号入力（A）の立ち上がりおよび立ち下がりエッジです。

3 は信号入力（A）の立ち上がりエッジです。

4 は信号入力（A）の立ち下がりエッジです。

コントローラーは最大 40kHz で入力をデコードできます。

`A`： エンコーダー入力 A ピン。8～11 である必要があります。

`B`： エンコーダー入力 B ピン。8～11 である必要があります。

encoder_enable_set_tick_count(*encoder_index, range_id*)

関数 `encoder_set_tick_count` によってティック カウントで更新されることが期待されるエンコーダーをセットアップします。

```
>>> encoder_enable_set_tick_count(0,0)
```

この例は、[-2147483648 ; 2147483647]の範囲でカウントを期待するようにエンコーダー0 をセットアップする方法を示しています。

パラメータ

`encoder_index` : 定義するエンコーダーのインデックス。0 または 1 である必要があります。

`range_id` : decoder_index : エンコーダーの範囲（integer）ラッピングを適切に処理するために必要です。

0 は 32 ビット符号付きエンコーダーであり、範囲は[-2147483648 ; 2147483647]です。

1 は 8 ビット符号なしエンコーダーであり、範囲は[0 ; 255]です。

2 は 16 ビット符号なしエンコーダーであり、範囲は[0 ; 65535]です。

3 は 24 ビット符号なしエンコーダーであり、範囲は[0 ; 16777215]です。

4 は 32 ビット符号なしエンコーダーであり、範囲は[0 ; 4294967295]です。

encoder_get_tick_count(*encoder_index*)

指定されたエンコーダーのティック カウントを返します。

```
>>> encoder_get_tick_count(0)
```

この例は、エンコーダー0 の現在のティック カウントを返します。

パラメータ

`encoder_index` : 照会するエンコーダーのインデックス。0 または 1 である必要があります。

戻り値

コンベア エンコーダーのティック カウント（float）

encoder_set_tick_count(*encoder_index, count*)

ロボット コントローラーにエンコーダーのティック カウントを通知します。この関数は、アブソリュート エンコーダーの場合に役立ちます（例：MODBUS）。

```
>>> encoder_set_tick_count(0, 1234)
```

この例は、エンコーダー0 のティック カウントを 1234 に設定します。最初に `encoder_enable_set_tick_count` を使用してエンコーダーが有効化されていることを前提としています。

パラメータ

`encoder_index`： 定義するエンコーダーのインデックス。0 または 1 である必要があります。

`count`： 設定するティック カウント。エンコーダーの範囲内である必要があります。

end_force_mode()

ロボット モードを力モードから通常の動作にリセットします。

これはプログラムの停止時にも実行されます。

end_freedrive_mode()

フリードライブモードの後、ロボットを通常位置制御モードに戻します。

end_teach_mode()

フリードライブモードの後、ロボットを通常位置制御モードに戻します。

force_mode(*task_frame, selection_vector, wrench, type, limits*)

力モードで制御するようにロボットを設定します。

パラメータ

`task_frame`：ポーズ ベクトルによって、ベース フレームを基準にしたカフレームが定義されます。

`selection_vector`：0 と 1 の 6 次元ベクトル。1 は、ロボットがタスク フレームの対応する軸に準拠することを意味します。

`wrench`：ロボットが環境に適用する力／トルク。ロボットは、指定された力／トルクを実現するため、準拠軸に沿って／準拠軸を中心に位置を調整します。非準拠軸については、値の影響はありません。

ジョイントの安全上の制限によって、実際に適用されるレンチは要求されたものよりも小さくなる場合があります。実際の力とトルクは、個別のスレッドで `get_tcp_force` 関数を使用して読み取ることができます。

`type`：ロボットがカフレームを解釈する方法を指定する整数[1;3]

1：カフレームは、y 軸がロボット tcp からカフレームの原点へ向かうベクトルに沿うように変形されます。

2：カフレームは変形されません。

3：カフレームは、x 軸がカフレームの xy 平面に対するロボット tcp 速度ベクトルの投影となるように変形されます。

`limits`：（Float）6 次元ベクトル。準拠軸の場合、これらの値はその軸に沿った／その軸を中心とした最大許容 tcp 速度です。非準拠軸の場合、これらの値は、実際の tcp 位置とプログラムで設定した tcp 位置の間の、軸に沿った／軸を中心とした最大許容偏差です。

注：最初に zero_ftsensor()を呼び出すことで、力トルク センサーの初期状態を計測してください。力モードに移行する前に、準拠軸に対して平行な動きと大きな減速は避けてください（少なくとも 0.02 秒の短い sleep コマンドを挿入することを検討してください）。力制御の正確性が落ちるため、力モードでの大きな加速は避けてください。

force_mode_example()

これは、上記の `force_mode()` 関数の例です。

コマンドの例：`force_mode(p[0.1,0,0,0,0.785], [1,0,0,0,0,0], [20,0,40,0,0,0], 2, [.2,.1,.1,.785,.785,1.57])`

●パラメータの例：

- タスク フレーム＝p[0.1,0,0,0,0.785] →このフレームは、ベースフレームから x 方向への 100mm のオフセットであり、rz 方向に 45 度回転しています。
- 選択ベクトル＝[1,0,0,0,0,0] →ロボットは上記のタスク フレームの x 方向に準拠しています。
- レンチ＝[20,0,40,0,0,0] →ロボットは x 方向に 20N 印加します。また、z 方向への 40N の外的な力も考慮します。
- タイプ＝2 →力フレームは変形されません。
- 制限＝[.1,.1,.1,.785,.785,1.57] →最大 x 速度は 100mm/s、最大 y 偏差は 100mm、最大 z 偏差は 100mm、最大 rx 偏差は 45 度、最大 ry 偏差は 45 度、最大 rz 偏差は 90 度です。

force_mode_set_damping(*damping*)

力モードで制動パラメータを設定します。

パラメータ

`damping`：0〜1：デフォルト値は 0.005 です。

値 1 は完全制動であり、力が存在しない場合、ロボットは速やかに減速します。値 0 は制動なしであり、ロボットは速度を維持します。

この関数が再度呼び出されるまで値は保存されます。これをプログラムの最初に追加すると、力モードに移行する前に呼び出されることが保証されます（そうしない場合、デフォルト値が使用されます）。

force_mode_set_gain_scaling(*scaling*)

カモードでゲインをスケーリングします。

パラメータ

`scaling`：0〜2 のスケーリング パラメータで、デフォルトは 1 です。

値を 1 より大きくすると、カモードを不安定にすることができます（例：硬い表面と衝突または押している場合）。

この関数が再度呼び出されるまで値は保存されます。これをプログラムの最初に追加すると、カモードに移行する前に呼び出されることが保証されます（そうしない場合、デフォルト値が使用されます）。

freedrive_mode()

ロボットをフリードライブモードに設定します。このモードでは、ロボットは「フリードライブ」ボタンを押すのと同じ方法で、手で動かすことができます。このモードでは、ロボットは軌道をたどることはできません（例：movej）。

get_conveyor_tick_count()

*廃止予定：*エンコーダーのティック カウントを通知します。低分解能エンコーダーでより正確な動きを得るため、コントローラーはティック カウントを補間します。

戻り値

コンベア エンコーダーのティック カウント

廃止予定：この関数は `encoder_get_tick_count` と置き換えられるため、今後は使用しないでください。

movec(*pose_via, pose_to, a*=1.2, *v*=0.25, *r*=0, *mode*=0)

円形移動：位置への移動（ツール空間内で円形）

TCP は、現在のポーズから pose_via を経由して pose_to まで、円弧線分上を移動します。一定のツール速度 v まで加速し、その速度で移動します。方向補間を定義するには、mode パラメータを使用します。

パラメータ

pose_via：経路点（注：位置のみが使用されます）。pose_via はジョイント位置として指定することもでき、対応するポーズの計算には順運動学が使用されます。

pose_to：目標ポーズ（注：固定方向モードでは位置のみが使用されます）。pose_to はジョイント位置として指定することもでき、対応するポーズの計算には順運動学が使用されます。

a：ツール加速度［m/s^2］

v：ツール速度［m/s］

r：（目標ポーズの）ブレンド半径［m］

mode：0：無制限モード。現在のポーズから目標ポーズ（pose_to）への方向を補間します。

1：固定モード。円弧の接線に対して方向を一定に保ちます（現在のポーズから）。

コマンドの例：movec(p[x,y,z,0,0,0], pose_to, a=1.2, v=0.25, r=0.05, mode=1)

●パラメータの例：

- 注：円上の第 1 位置は前のウェイポイントです。
- pose_via＝p[x,y,z,0,0,0] →円上の第 2 位置。
  - ＊注：回転は使用されないため、ゼロのままとすることができます。
  - ＊注：この位置はジョイント角度[j0,j1,j2,j3,j4,j5]として表すこともでき、対応するポーズの計算には順運動学が使用されます。
- pose_to →円上の第 3（最終）位置
- a = 1.2 →加速度は 1.2m/s/s です。
- v = 0.25 →速度は 250mm/s です。
- r = 0 →（pose_to での）ブレンド半径は 50mm です。
- mode = 1 →円弧の接線に対して固定された方向を使用します。

movej(*q*, *a*=1.4, *v*=1.05, *t*=0, *r*=0)

位置への移動（ジョイント空間内で線形）

このコマンドを使用するとき、ロボットは静止しているか、ブレンドを使用して movej または movel から来ている必要があります。速度および加速度パラメータによって、移動の台形速度プロファイルが制御されます。あるいは、t パラメータを使用してこの移動の時間を設定できます。時間設定は速度および加速度設定よりも優先されます。

パラメータ

q： ジョイント位置（q はポーズとして指定することもでき、対応するジョイント位置の計算には逆運動学が使用されます）

a： リード軸のジョイント加速度［rad/s^2］

v： リード軸のジョイント速度［rad/s］

t： 時間［S］

r： ブレンド半径［m］

ブレンド半径が設定されている場合、ロボットがその点で停止しないようにロボット アームの軌道が変更されます。

ただし、この移動のブレンド領域が前後のウェイポイントのブレンド半径と重なっている場合、この移動はスキップされ、「ブレンドが重なっています」という警告メッセージが生成されます。

コマンドの例：`movej([0,1.57,-1.57,3.14,-1.57,1.57], a=1.4, v=1.05, t=0, r=0)`

●パラメータの例：

- q＝[0,1.57,-1.57,3.14,-1.57,1.57] →ベースが 0 度回転、肩が 90 度回転、肘が-90 度回転、手首 1 が 180 度回転、手首 2 が-90 度回転、手首 3 が 90 度回転です。注：ジョイント位置（q はポーズとして指定することもでき、対応するジョイント位置の計算には逆運動学が使用されます）
- a = 1.4 →加速度は 1.4rad/s/s です。
- v = 1.05 →速度は 1.05rad/s です。
- t = 0 →移動する時間（秒）は指定されていません。指定されると、コマンドは a と v の値を無視します。
- r = 0 →ブレンド半径は 0 メートルです。

movel(*pose*, *a*=1.2, *v*=0.25, *t*=0, *r*=0)

位置への移動（ツール空間内で線形）

movej を参照してください。

パラメータ

pose：目標ポーズ（pose はジョイント位置として指定することもでき、対応するポーズの計算には順運動学が使用されます）

a：　ツール加速度［m/s^2］

v：　ツール速度［m/s］

t：　時間［S］

r：　ブレンド半径［m］

コマンドの例：`movel(pose, a=1.2, v=0.25, t=0, r=0)`

●パラメータの例：

- pose＝p[0.2,0.3,0.5,0,0,3.14] →x＝200mm、y＝300mm、z＝500mm、rx＝0、ry＝0、rz＝180 度のベース フレームの位置
- a = 1.2 →加速度 1.2m/s^2
- v = 0.25 →速度 250mm/s
- t = 0 →移動する時間（秒）は指定されていません。指定されると、コマンドは a と v の値を無視します。
- r = 0 →ブレンド半径は 0 メートルです。

movep(*pose*, *a*=1.2, *v*=0.25, *r*=0)

移動プロセス

位置に対して円形にブレンド（ツール空間）し、線形に移動（ツール空間）します。一定のツール速度 v まで加速し、その速度で移動します。

パラメータ

pose：目標ポーズ（pose はジョイント位置として指定することもでき、対応するポーズの計算には順運動学が使用されます）

a：　ツール加速度［m/s^2］

v：　ツール速度［m/s］

r：　ブレンド半径［m］

コマンドの例：movep(pose, a=1.2, v=0.25, r=0)

●パラメータの例：

- pose＝p[0.2,0.3,0.5,0,0,3.14] →x＝200mm、y＝300mm、z＝500mm、rx＝0、ry＝0、rz＝180 度のベース フレームの位置
- a = 1.2 →加速度 1.2m/s^2
- v = 0.25 →速度 250mm/s
- r = 0 →ブレンド半径は 0 メートルです。

position_deviation_warning(*enabled, threshold*=0.8)

有効化すると、この関数は、ロボットが目標位置から逸脱したときにログに警告メッセージを出力します。この関数はプログラムの実行中、任意の時点で呼び出すことができます。戻り値はありません。

```
>>> position_deviation_warning(True)
```

上記の例では、関数が有効になっています。これは、位置が逸脱するといつでもログ メッセージが生成されることを意味しています。任意の「threshold」パラメータを使用して、ログ メッセージをトリガする位置の逸脱のレベルを指定できます。

パラメータ

`enabled`： （Boolean）位置逸脱ログ メッセージを有効化または無効化します。

`threshold`： （Float）範囲[0;1]の任意の値です。0 は位置逸脱なしで、1 は最大位置逸脱です（ロボットの保護停止を発生させる位置逸脱と同じ量です）。ユーザーがしきい値を指定しない場合、デフォルト値 0.8 が使用されます。

コマンドの例：`position_deviation_warning(True, 0.8)`

●パラメータの例：

- Enabled＝True →警告のログ記録をオンにします。
- Threshold＝0.8 →保護停止を発生させる逸脱の 80%で、ログ履歴ファイルに記録される警告を発生させます。

reset_revolution_counter(*qNear*=`[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]`)

オフセットが指定されていない場合、回転計をリセットします。これは、安全制限が「無制限」に設定されているジョイントに適用され、ジョイント角度が制限されている新しい安全設定が適用されたときにのみ適用されます。

```
>>> reset_revolution_counter()
```

パラメータ

`qNear`： 任意のパラメータであり、指定された qNear ジョイント ベクトルに近い方に回転計をリセットします。定義しない場合、ジョイントの実際の回転数が使用されます。

コマンドの例：`reset_revolution_counter(qNear=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0])`

●パラメータの例：

- qNear = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0] →任意のパラメータであり、UR3 ロボットのゼロに対する手首 3 の回転計を、qNear で表されるジョイントの回転に最も近いゼロ位置にリセットします。

servoc(*pose*, *a*=`1.2`, *v*=`0.25`, *r*=`0`)

円形サーボ制御

位置へのサーボ制御（ツール空間内で円形）一定のツール速度 v まで加速し、その速度で移動します。

パラメータ

`pose`： 目標ポーズ（pose はジョイント位置として指定することもでき、対応するポーズの計算には順運動学が使用されます）

`a`： ツール加速度［m/s^2］

`v`： ツール速度［m/s］

`r`： （目標ポーズの）ブレンド半径［m］

コマンドの例：`servoc(p[0.2,0.3,0.5,0,0,3.14], a=1.2, v=0.25, r=0)`

●パラメータの例：

- pose＝p[0.2,0.3,0.5,0,0,3.14] →x＝200mm、y＝300mm、z＝500mm、rx＝0、ry＝0、rz＝180 度のベース フレームの位置
- a = 1.2 →加速度 1.2m/s^2
- v = 0.25 →速度 250mm/s
- r = 0 →目標位置でのブレンド半径は 0 メートルです。

servoj(*q, a, v, t*=0.002, *lookahead_time*=0.1, *gain*=300)

位置へのサーボ制御（ジョイント空間内で線形）

ロボットのオンライン制御に使用されるサーボ関数。先読み時間およびゲインを使用して軌道を平滑化または鋭化できます。

注：ゲインを大きくしたり、先読み時間を短くすると、安定性が低下する場合があります。各ステップで新しい設定値（q）を使用してこの関数を呼び出すことをお勧めします（デフォルトは t=0.002）。

パラメータ

q：	ジョイント位置［rad］
a：	現在のバージョンでは使用されません。
v：	現在のバージョンでは使用されません。
t：	コマンドがロボットを制御している時間。関数は時間 t［s］の間ブロックします。
lookahead_time：	時間［s］、範囲[0.03,0.2]によって、この先読み時間で軌道を平滑化します。
gain：	次の目標位置の比例ゲイン、範囲[100,2000]

コマンドの例：`servoj([0.0,1.57,-1.57,0,0,3.14], 0, 0, 0.1, 0.1, 300)`

●パラメータの例：

- q＝[0.0,1.57,-1.57,0,0,3.14] →ベース、肩、肘、手首 1、手首 2、手首 3 の回転を表すジョイント角度（ラジアン単位）
- a＝0 →現在のバージョンでは使用されません。
- v＝0 →現在のバージョンでは使用されません。
- t＝0.1 →コマンドがロボットを制御している時間。関数は時間 t［s］の間ブロックします。
- lookahead time＝0.1 →時間［s］、範囲[0.03,0.2]によって、この先読み時間で軌道を平滑化します。
- gain＝300 →次の目標位置の比例ゲイン、範囲[100,2000]

set_conveyor_tick_count(*tick_count, absolute_encoder_resolution*=0)

廃止予定：ロボット コントローラーにエンコーダーのティック カウントを通知します。この関数は、アブソリュート エンコーダーの場合に役立ちます。インクリメンタル エンコーダーのセットアップには conveyor_pulse_decode()を使用します。円形コンベアの場合、値は 0 から回転あたりのティック数までの範囲内である必要があります。

パラメータ

`tick_count`：コンベアのティック カウント（Integer）

`absolute_encoder_resolution`：エンコーダーの分解能（ラッピングを適切に処理するために必要です）。（Integer）

0 は 32 ビット符号付きエンコーダーであり、範囲は[-2147483648 ; 2147483647]です（デフォルト）

1 は 8 ビット符号なしエンコーダーであり、範囲は[0 ; 255]です。

2 は 16 ビット符号なしエンコーダーであり、範囲は[0 ; 65535]です。

3 は 24 ビット符号なしエンコーダーであり、範囲は[0 ; 16777215]です。

4 は 32 ビット符号なしエンコーダーであり、範囲は[0 ; 4294967295]です。

廃止予定：この関数は `encoder_set_tick_count` と置き換えられるため、今後は使用しないでください。

コマンドの例：`set_conveyor_tick_count(24543, 0)`

●パラメータの例：

- Tick_count＝24543 →アブソリュート エンコーダーによって更新される MODBUS レジスタなどから読み取られる値
- Absolute_encoder_resolution＝0 →0 は 32 ビット符号付きエンコーダーであり、範囲は[-2147483648 ; 2147483647]です（デフォルト）

set_pos(*q*)

シミュレートされたロボットのジョイント位置を設定します。

パラメータ

`q`： ジョイント位置

コマンドの例：`set_pos([0.0,1.57,-1.57,0,0,3.14])`

●パラメータの例：

- q＝[0.0,1.57,-1.57,0,0,3.14] →ベース、肩、肘、手首 1、手首 2、手首 3 の回転を表すジョイント角度（ラジアン単位）によってシミュレートされたロボットの位置

set_safety_mode_transition_hardness(*type*)

通常モード、減速モード、安全停止間の遷移硬度を設定します。

パラメータ

`type`：遷移硬度を指定する整数。

0 は非常停止と同様に、最大トルクを使用して、モード間の遷移が急です。

1 はモード間の遷移が緩やかです。

speedj(*qd, a, t*)

ジョイント速度

ジョイント空間で線形に加速し、一定のジョイント速度を維持します。時間 t は任意です。指定されると、目標速度に達したかどうかにかかわらず、関数は時間 t 後に戻ります。時間 t が指定されない場合、目標速度に達すると関数が戻ります。

パラメータ

`qd`： ジョイント速度［rad/s］

`a`： ジョイント加速度［rad/s^2］（リード軸）

`t`： 関数が戻るまでの時間［s］（任意）

コマンドの例：`speedj([0.2,0.3,0.1,0.05,0,0], 0.5, 0.5)`

●パラメータの例：

- qd →ベース＝0.2rad/s、肩＝0.3rad/s、肘＝0.1rad/s、手首 1＝0.05rad/s、手首 2 および手首 3＝0rad/s のジョイント速度
- a＝0.5rad/s^2 →リード軸の加速度（この場合は肩）
- t＝0.5s →関数が戻るまでの時間

speedl(*xd, a, t, aRot=*'a')

ツール速度

デカルト空間で線形に加速し、一定のツール速度を維持します。時間 t は任意です。指定されると、目標速度に達したかどうかにかかわらず、関数は時間 t 後に戻ります。時間 t が指定されない場合、目標速度に達すると関数が戻ります。

パラメータ

`xd`： ツール速度［m/s］（空間ベクトル）

`a`： ツール位置加速度［m/s^2］

`t`： 関数が戻るまでの時間［s］（任意）

`aRot`： ツール加速度［rad/s^2］（任意）。位置加速度 a が定義されてない場合に使用されます。

コマンドの例：`speedl([0.5,0.4,0.0,0.,1.57,0,0], 0.5, 0.5)`

●パラメータの例：

- qd →x＝500mm/s、y＝400mm/s、rx＝90deg/s、ry および rz＝0mm/s のツール速度
- a＝0.5rad/s^2 →リード軸の加速度（この場合は肩）
- t＝0.5s →関数が戻るまでの時間

stop_conveyor_tracking(*a*=`20`)

track_conveyor_linear()または track_conveyor_circular()で開始されたコンベアのトラッキングを停止し、すべてのジョイント速度をゼロに減速します。

パラメータ

`a`： ジョイント加速度［rad/s^2］（任意）

コマンドの例：`stop_conveyor_tracking(a=15)`

●パラメータの例：

- a = 15rad/s^2 →ジョイントの加速度

stopj(*a*)

停止（ジョイント空間内で線形）

ジョイント速度をゼロに減速します。

パラメータ

`a`： ジョイント加速度［rad/s^2］（リード軸）

コマンドの例：`stopj(2)`

●パラメータの例：

- a＝2rad/s^2 →リード軸の減速率。

stopl(*a, aRot=*'`a`')

停止（ツール空間内で線形）

ツール速度をゼロに減速します。

パラメータ

`a`： ツール加速度［m/s^2］

`aRot`： ツール加速度［rad/s^2］（任意）。位置加速度 a が定義されてない場合に使用されます。

コマンドの例：`stopl(20)`

●パラメータの例：

- a＝20m/s^2 →ツールの減速率。
- aRot →ツール減速度［rad/s^2］（任意）。位置加速度 a が定義されてない場合に使用されます。すなわち、減速度「a」の代わりに使用されます。

teach_mode()

ロボットをフリードライブモードに設定します。このモードでは、ロボットは「フリードライブ」ボタンを押すのと同じように、ロボットを手で動かすことができます。このモードでは、ロボットは軌道をたどることはできません（例：movej）。

track_conveyor_circular(*center, ticks_per_revolution, rotate_tool=*'`False`', *encoder_index=*`0`)

ロボットの動き（movej()など）を円形コンベアに追従させます。

```
>>> track_conveyor_circular(p[0.5,0.5,0,0,0,0],500.0,
>>> false)
```

コードの例では、ロボットがロボット ベース座標系の p[0.5,0.5,0,0,0,0]を中心として円形コンベアを追従するようにしています。エンコーダーの 500 ティックが円形コンベアの中心周辺の 1 回転に対応しています。

パラメータ

`center`：	ロボットのベース座標系におけるコンベアの中心を決定するポーズ ベクトル。
`ticks_per_revolution`：	コンベアが 1 回転したときにエンコーダーによって確認されるティック数。
`rotate_tool`：	ツールはコンベアと一緒に回転するか、軌道によって指定された方向を維持する（movel()など）必要があります。
`encoder_index`：	コンベア トラッキングに関連付けられているエンコーダーのインデックス。0 または 1 である必要があります。これは任意の引数であり、デフォルトは 0 であることに注意してください。この引数を省略しても、既存のプログラムは動作し続けます。また、追跡するコンベアが 1 つだけの場合も、この引数を省略することを検討してください。

コマンドの例：

```
track_conveyor_circular(p[0.5,0.5,0,0,0,0], 500.0, false)
```

●パラメータの例：

- center＝p[0.5,0.5,0,0,0,0] →コンベアの中心位置。
- ticks_per_revolution＝500 →コンベアが 1 回転したときにエンコーダーによって確認されるティック数。
- rotate_tool＝false →ツールはコンベアと一緒に回転しませんが、軌道によって指定された方向を維持する（`movel()`など）必要があります。

track_conveyor_linear(*direction, ticks_per_meter, encoder_index*=0)

ロボットの動き（movej()など）を線形コンベアに追従させます。

```
>>> track_conveyor_linear(p[1,0,0,0,0,0],1000.0)
```

コードの例では、ロボットがロボット ベース座標系の x 軸でコンベアを追従するようにしています。エンコーダーの 1000 ティックが x 軸の 1m に対応しています。

パラメータ

direction：	ロボットのベース座標系におけるコンベアの方向を決定するポーズ ベクトル。
ticks_per_meter：	コンベアが 1m 移動したときにエンコーダーによって確認されるティック数。
encoder_index：	コンベア トラッキングに関連付けられているエンコーダーのインデックス。0 または 1 である必要があります。これは任意の引数であり、デフォルトは 0 であることに注意してください。この引数を省略しても、既存のプログラムは動作し続けます。また、追跡するコンベアが 1 つだけの場合も、この引数を省略することを検討してください。

コマンドの例：`track_conveyor_linear(p[1,0,0,0,0,0], 1000.0)`

●パラメータの例：

- direction＝p[1,0,0,0,0,0] →ロボットのベース座標系におけるコンベアの方向を決定するポーズ ベクトル。
- ticks_per_meter＝1000. → コンベアが 1m 移動したときにエンコーダーによって確認されるティック数。

2.2 変数

名前	説明
__package__	**値**：'Motion'
a_joint_default	**値**：1.4
a_tool_default	**値**：1.2
v_joint_default	**値**：1.05
v_tool_default	**値**：0.25

3 内部モジュール

3.1 関数

force()

TCP にかかる力を返します。

現在 TCP に外部からかけられている力を返します。力は get_tcp_force()を使用して計算された Fx、Fy、Fz のノルムです。

戻り値

ニュートン単位の力（float）

get_actual_joint_positions()

すべてのジョイントの実際の角度位置を返します。

実際の角度位置はラジアン単位で表され、長さ 6 のベクトルとして返されます。特に加速中や重荷重時、出力は get_target_joint_positions()の出力と異なる場合があることに注意してください。

戻り値

現在の実際のジョイント角度位置ベクトル（ラジアン単位）。[ベース, 肩, 肘, 手首 1, 手首 2, 手首 3]

get_actual_joint_speeds()

すべてのジョイントの実際の角速度を返します。

実際の角速度はラジアン／秒単位で表され、長さ 6 のベクトルとして返されます。特に加速中や重荷重時、出力は get_target_joint_speeds()の出力と異なる場合があることに注意してください。

戻り値

現在の実際のジョイント角速度ベクトル（ラジアン／秒単位）。[ベース, 肩, 肘, 手首 1, 手首 2, 手首 3]

get_actual_tcp_pose()

現在の計測されたツール ポーズを返します。

ベース フレームで指定されたツール位置および方向を表す 6 次元ベクトルを返します。このポーズの計算は実際のロボット エンコーダーの読み取り値に基づきます。

戻り値

現在の実際の TCP ベクトル[X, Y, Z, Rx, Ry, Rz]

get_actual_tcp_speed()

現在の計測された TCP 速度を返します。

ポーズ構造で返される TCP の速度。最初の 3 つの値は x、y、z に沿ったデカルト速度であり、後の 3 つは現在の回転軸 rx、ry、rz を定義し、長さ[rz,ry,rz]によって角速度（ラジアン／秒）が定義されます。

戻り値

現在の実際の TCP 速度ベクトル[X, Y, Z, Rx, Ry, Rz]

get_actual_tool_flange_pose()

現在の計測されたツール フランジ ポーズを返します。

ベース フレームで指定されたツール フランジ位置および方向を表す、ツール中心点オフセットのない 6 次元ベクトルを返します。このポーズの計算は実際のロボット エンコーダーの読み取り値に基づきます。

戻り値

現在の実際のツール フランジ速度。[X, Y, Z, Rx, Ry, Rz]

注：TCP オフセットを含む実際の 6 次元ポーズについては、`get_actual_tcp_pose` を参照してください。

get_controller_temp()

コントロール ボックスの温度を返します。

ロボット コントロール ボックスの温度（°C）を返します。

戻り値

温度（°C）（float）

get_forward_kin(*q=*'current_joint_positions', *tcp=*'active_tcp')

校正したロボット運動学を使用して、順運動学変換（ジョイント空間→ツール空間）を計算します。ジョイント位置ベクトルが指定されていない場合、ロボット アームの現在のジョイント角度が使用されます。tcp が指定されていない場合、コントローラーの現在アクティブな tcp が使用されます。

パラメータ

q： ジョイント位置ベクトル（任意）

tcp： tcp オフセット ポーズ（任意）

戻り値

ツール ポーズ

コマンドの例：`get_forward_kin([0.,3.14,1.57,.785,0,0], p[0,0,0.01,0,0,0])`

●パラメータの例：

- q＝[0.,3.14,1.57,.785,0,0] →j0＝0 度、j1＝180 度、j2＝90 度、j3＝45 度、j4＝0 度、j5＝0 度のジョイント角度。
- tcp＝p[0,0,0.01,0,0,0] →x＝0mm、y＝0mm、z＝10mm の tcp オフセットおよび rx＝0 度、ry＝0 度、rz＝0 度の回転ベクトル。

get_inverse_kin(*x, qnear, maxPositionError*=`1e-10`, *maxOrientationError*=`1e-10`, *tcp*=*'*`active_tcp`*'*)

逆運動学変換（ツール空間→ジョイント空間）を計算します。qnear が定義されている場合、qnear に最も近い解が返されます。そうでない場合、現在のジョイント位置に最も近い解が返されます。tcp が指定されていない場合、コントローラーの現在アクティブな tcp が使用されます。

パラメータ

`x`：	ツール ポーズ
`qnear`：	ジョイント位置のリスト（任意）
`maxPositionError`：	最大許容位置誤差（任意）
`maxOrientationError`：	最大許容方向誤差（任意）
`tcp`：	tcp オフセット ポーズ（任意）

戻り値

ジョイント位置

コマンドの例：`get_inverse_kin(p[.1,.2,.2,0,3.14,0], [0.,3.14,1.57,.785,0,0])`

●パラメータの例：

- x＝p[.1,.2,.2,0,3.14,0] →位置が x＝100mm、y＝200mm、z＝200mm、回転ベクトルが rx＝0 度、ry＝180 度、rz＝0 度のポーズ。
- qnear＝[0.,3.14,1.57,.785,0,0] →解は j0＝0 度、j1＝180 度、j2＝90 度、j3＝45 度、j4＝0 度、j5＝0 度のジョイント角度に近い必要があります。
- maxPositionError のデフォルトは 1e-10m です。
- maxOrientationError のデフォルトは 1e-10 ラジアンです。

get_joint_temp(*j*)

ジョイント j の温度を返します。

ゼロからカウントしたジョイント j のジョイント ハウスの温度。j＝0 はベース ジョイントで、j＝5 はツール フランジ前の最後のジョイントです。

パラメータ

`j`： ジョイント番号（int）

戻り値

温度（°C）（float）

get_joint_torques()

すべてのジョイントのトルクを返します。

（重力や摩擦など）ロボット自体の移動に必要なトルクによって補正されたジョイントのトルク。長さ 6 のベクトルとして返されます。

戻り値

ジョイント トルク ベクトル（Nm 単位）[ベース, 肩, 肘, 手首 1, 手首 2, 手首 3]

get_steptime()

ロボット時間ステップの期間を返します（秒単位）。

すべての時間ステップで、ロボット コントローラーは計測したジョイント位置と速度をロボットから受信し、目的のジョイント位置および速度をロボットに送信します。これは、定期的な間隔で周期的に発生します。この間隔の長さはロボット時間ステップです。

戻り値

ロボット ステップの期間（秒単位）

get_target_joint_positions()

すべてのジョイントの目的の角度位置を返します。

目標の角度位置はラジアン単位で表され、長さ 6 のベクトルとして返されます。特に加速中や重荷重時、出力は get_actual_joint_positions()の出力と異なる場合があることに注意してください。

戻り値

現在の目標のジョイント角度位置ベクトル（ラジアン単位）。[ベース, 肩, 肘, 手首 1, 手首 2, 手首 3]

get_target_joint_speeds()

すべてのジョイントの目的の角速度を返します。

目標の角速度はラジアン／秒単位で表され、長さ 6 のベクトルとして返されます。特に加速中や重荷重時、出力は get_actual_joint_speeds()の出力と異なる場合があることに注意してください。

戻り値

現在の目標のジョイント角速度ベクトル（ラジアン／秒単位）。[ベース, 肩, 肘, 手首 1, 手首 2, 手首 3]

get_target_payload()

アクティブなペイロードの重さを返します。

戻り値

現在のペイロードの重さ（キログラム単位）

get_target_payload_cog()

アクティブなペイロードの重心座標を返します。

このスクリプトは、ツール フランジに関して、アクティブなペイロードの重心座標を返します。

戻り値

重心の 3 次元座標[CoGx, CoGy, CoGz]（メートル単位）

get_target_tcp_pose()

現在の目標ツール ポーズを返します。

ベース フレームで指定されたツール位置および方向を表す 6 次元ベクトルを返します。このポーズの計算は現在の目標ジョイント位置に基づきます。

戻り値

現在の目標 TCP ベクトル[X, Y, Z, Rx, Ry, Rz]

get_target_tcp_speed()

現在の目標 TCP 速度を返します。

ポーズ構造で返される TCP の目的の速度。最初の 3 つの値は x、y、z に沿ったデカルト速度であり、後の 3 つは現在の回転軸 rx、ry、rz を定義し、長さ[rz,ry,rz]によって角速度（ラジアン／秒）が定義されます。

戻り値

TCP 速度（pose）

get_tcp_force()

TCP のレンチ（力／トルク ベクトル）を返します。

この関数は p[Fx(N), Fy(N), Fz(N), TRx(Nm), TRy(Nm), TRz(Nm)]を返します。Fx、Fy、Fz はニュートン単位で計測されたロボット ベース座標系の軸の力であり、TRx、TRy、TRz はニュートン時間メートル単位で計測されたこれらの軸の周りのトルクです。

戻り値

レンチ（pose）

get_tcp_offset()

アクティブな tcp オフセット、すなわち出力フランジ座標系から TCP への変換をポーズとして取得します。

戻り値

tcp オフセット ポーズ

get_tool_accelerometer_reading()

ツール加速度計の現在の読み取り値を 3 次元ベクトルとして返します。

加速度計軸はツール座標系に沿っており、軸の上方向を指すと正の読み取り値が得られます。

戻り値

X、Y、Z は SI 単位系で計測された加速度（m/s^2）の構成要素です。

get_tool_current()

ツール電流を返します。

アンペア単位で計測されたツール消費電流。

戻り値

ツール電流（アンペア単位）。

is_steady()

ロボットが完全に静止しているかどうかを確認します。

ロボットが完全に静止しており、工業用ドライバーからなど、外部からの大きな力およびトルクを受け入れる準備ができている場合、true が返されます。これは、ロボットの位置に影響を与えるドライバーやその他のアクチュエーターを起動させる前に、GUI の待機ノードと組み合わせると便利です。

注：標準位置モード以外のモードでは、この関数は常に false を返します。力モードや教育モードなどでは false です。

戻り値

ロボットが完全に静止している場合は True。そうでない場合は False を返します（bool）。

is_within_safety_limits(*pose*)

ロボットの現在の安全制限内で指定されたポーズに到達できるかどうかをチェックします。

このチェックでは、ジョイント制限、安全平面制限、ロボットの TCP 方向偏差制限および範囲が考慮されます。指定された目標 TCP ポーズに逆運動学を適用したときに解が見つかると、このポーズは到達可能とみなされます。

パラメータ

`pose`：目標ポーズ（ジョイント位置として指定することもできます）

戻り値

制限内にある場合は true、そうでない場合は false です（bool）

コマンドの例：`is_within_safety_limits(p[.1,.2,.2,0,3.14,0])`

●パラメータの例：

- pose＝p[.1,.2,.2,0,3.14,0] →位置が x＝100mm、y＝200mm、z＝200mm、回転ベクトルが rx＝0 度、ry＝180 度、rz＝0 度の目標ポーズ。

popup(*s, title='*`Popup`*', warning=*`False`*, error=*`False`*, blocking=*`False`)

GUI にポップアップを表示します。

GUI のポップアップ ウィンドウにメッセージを表示します。

パラメータ

`s`： メッセージ文字列

`title`： タイトル文字列

`warning`： 警告メッセージかどうか

`error`： エラー メッセージかどうか

`blocking`： True の場合、「continue」が渡されるまでプログラムが中断されます。

コマンドの例：`popup("here I am", title="Popup #1",blocking=True)`

●パラメータの例：

- s ポップアップ テキストは「here I am」です。
- title ポップアップ タイトルは「Popup #1」です。
- blocking = true →他のアクションを実行する前に、ポップアップをクリアする必要があります。

powerdown()

ロボットをシャットダウンし、ロボットとコントローラーの電源をオフにします。

set_gravity(*d*)

ロボットに発生している加速度の方向を設定します。ロボットの位置が固定されている場合、これは地球の中心からかかる重力加速度に対応します。

```
>>> set_gravity([0, 9.82*sin(theta), 9.82*cos(theta)])
```

これは、ロボット ベース座標系の x 軸の周りを「theta」ラジアン回転したロボットの加速度を設定します。

パラメータ

`d`： ロボットのベースを基準として重力の方向を示す 3 次元ベクトル。

コマンドの例：`set_gravity(0,9.82,0)`

●パラメータの例：

- d は方向 y（ロボット ケーブルの方向）、マグニチュード 9.82m/s^2（1g）のベクトルです。

set_payload(*m, cog*)

ペイロードの質量と重心を設定します。

ペイロードの質量と重心（略：CoG）を設定します。

この関数は、ペイロードの重量または重量分布が変化したとき、つまりロボットが重いワークピースを持ち上げたまたは置いたときに呼び出す必要があります。

パラメータ

`m`： 質量（キログラム単位）

`cog`： 重心、ツールのマウント位置からの変位を示すベクトル[CoGx, CoGy, CoGz]（メートル単位）。

警告：cog パラメータの省略は**非推奨**です。後で `set_tcp` を呼び出すと重心も新しいツール中心点（TCP）に変更されるという副作用によって、cog パラメータが存在しない場合、TCP が使用されます。`set_payload_mass` 関数を使用して質量のみを変更するか、第 2 引数として `get_target_payload_cog` を使用して重心が変更されないようにします。

コマンドの例：

●`set_payload(3., [0,0,.3])`

- パラメータの例：
  - ＊m＝3 →質量が 3kg のペイロードに設定されます。
  - ＊cog＝[0,0,.3] →ツール座標におけるツール マウントの中心から x＝0mm、y＝0mm、z＝300mm に重心が設定されます。

●`set_payload(2.5, get_target_payload_cog())`

- パラメータの例：
  - ＊m＝2.5 →質量が 2.5kg のペイロードに設定されます。
  - ＊cog＝現在の重心設定を使用します。

set_payload_cog(*CoG*)

重心（CoG）を設定します。

廃止予定：この関数は廃止予定です。**常に**質量を使用して重心を設定することをお勧めします（`set payload` 参照）。

set_payload_mass(*m*)

ペイロードの質量を設定します。

`set_payload` も参照してください。

ペイロードの質量を指定し、重心（CoG）はそのまま変更しません。

パラメータ

`m`： 質量（キログラム単位）

コマンドの例：`set_payload_mass(3.)`

●パラメータの例：

- m＝3 →質量が 3kg のペイロードに設定されます。

set_tcp(*pose*)

アクティブな tcp オフセット、すなわち出力フランジ座標系から TCP への変換をポーズとして設定します。

パラメータ

`pose`： 変換を示すポーズ。

コマンドの例：`set_tcp(p[0.,.2,.3,0.,3.14,0.])`

●パラメータの例：

- pose＝p[0.,.2,.3,0.,3.14,0.] →ツール中心点はツール座標における x＝0mm、y＝200mm、z＝300mm、回転ベクトル rx＝0 度、ry＝180 度、rz＝0 度に設定されます。

sleep(*t*)

一定時間スリープします。

パラメータ

`t`： 時間［s］

コマンドの例：`sleep(3.)`

●パラメータの例：

- t＝3.→スリープする時間

str_at(*src, index*)

文字列のバイトに対する直接アクセスを提供します。

このスクリプトでは、指定された `index` に対応する位置のソース文字列のバイトを含む文字列を返します。特殊エンコード文字を含む文字列（マルチバイト エンコーディングや可変長エンコーディング）の場合、実際の文字に対応しない場合があります。

文字列はゼロからインデックスが作成されます。

パラメータ

`src`： ソース文字列。

`index`： ソース文字列内の位置を指定する整数。

戻り値

ソース文字列の位置インデックスのバイトを含む文字列。`index` が有効ではない場合、例外が発生します。

コマンドの例：

●`str_at("Hello", 0)`
- 「H」を返します

●`str_at("Hello", 1)`
- 「e」を返します

●`str_at("Hello", 10)`
- エラー（インデックス範囲外）

●`str_at("", 0)`
- エラー（ソース文字列が空）

str_cat(*op1, op2*)

文字列の結合

このスクリプトでは、入力として指定した 2 つのオペランドを結合した文字列が返されます。2 つのオペランドは以下の型のいずれかとなります。String、Boolean、Integer、Float、Pose、Boolean/Integer/Float/Pose のリスト他の型の場合は例外が発生します。

得られる文字列は 1023 文字を超えることはできず、超えてしまうと例外がスローされます。

Float の数値は 6 桁の小数にフォーマットされ、末尾のゼロは削除されます。

関数をネストして複雑な文字列を作成することができます（最後の例を参照）。

パラメータ

`op1`： 第 1 オペランド

`op2`： 第 2 オペランド

戻り値

op1 と op2 の結合文字列

コマンドの例：

●`str_cat("Hello", " World!")`
- 「Hello World!」が返されます。

●`str_cat("Integer ", 1)`
- 「Integer 1」が返されます。

●`str_cat("", p[1.0, 2.0, 3.0, 4.0, 5.0, 6.0])`
- 「p[1, 2, 3, 4, 5, 6]」が返されます。

●`str_cat([True, False, True], [1, 0, 1])`
- 「[True, False, True][1, 0, 1]」が返されます。

●
```
str_cat(str_cat("", str_cat("One",
"Two")),str_cat(3, 4))
```
- 「OneTwo34」が返されます。

str_empty(*str*)

`str` が空の場合 true が返され、そうでない場合は false が返されます。

パラメータ

`str`： ソース文字列。

戻り値

文字列が空の場合 true が返され、そうでない場合は false が返されます。

コマンドの例：

●`str_empty("")`

- True が返されます。

●`str_empty("Hello")`

- False が返されます。

str_find(*src, target, start_from*=`0`)

`src` 内で部分文字列 target が初めて出現する場所が返されます。

このスクリプトは、指定された（任意の）位置から、`str` 内で部分文字列 `target` が初めて出現するインデックス（すなわちバイト）を返します。

`src` にマルチバイトまたは可変長エンコード文字が含まれる場合、target の最初の文字の実際の位置に対応しない場合があります。

文字列はゼロからインデックスが作成されます。

パラメータ

`src`： ソース文字列。

`target`： 検索する部分文字列。

`Start_from`： 任意の開始位置（デフォルトは 0）。

戻り値

`src` 内で `target` が初めて出現する場所のインデックス。`src` 内で `target` が見つからない場合-1。

コマンドの例：

●`str_find("Hello World!", "o")`

- 4 が返されます。

●`str_find("Hello World!", "lo")`

- 3 が返されます。

●`str_find("Hello World!", "o", 5)`

- 7 が返されます。

●`str_find("abc", "z")`

- -1 が返されます。

str_len(*str*)

文字列のバイト数を返します。

返される値は、マルチバイトまたは可変長エンコード文字のシーケンスの実際の文字数と対応しない場合があることに注意してください。

文字列はゼロからインデックスが作成されます。

パラメータ

`str`： ソース文字列。

戻り値

入力文字のバイト数。

コマンドの例：

●`str_len("Hello")`

- 5 が返されます。

●`str_len("")`

- 0 が返されます。

str_sub(*src, index, len*)

`src` の部分文字列を返します。

結果は、最大 `len` バイトの長さの、インデックスで指定されたバイトから始まる、`src` の部分文字列です。リクエストされた部分文字列が元の文字列の末尾以降まで拡張されている場合（例：`index + len > src` の長さ）、得られる部分文字列の長さは `src` のサイズに制限されます。

`index` および／または `len` が範囲外の場合、例外がスローされます。文字列はゼロからインデックスが作成されます。

パラメータ

`src`： ソース文字列。

`index`： 範囲[0, src の長さ]の最初のバイトを指定する整数値。

`len`： （任意）範囲[0, MAX_INT]の部分文字列の長さ。len が指定されていない場合。範囲[index, src の長さ]の文字列。

戻り値

バイト index から始まる長さ len 文字の src の部分文字列。

コマンドの例：

●`str_sub("0123456789abcdefghij", 5, 3)`
- 「567」が返されます。

●`str_sub("0123456789abcdefghij", 10)`
- 「abcdefghij」が返されます。

●`str_sub("0123456789abcdefghij", 2, 0)`
- 「」が返されます（len が 0）。

●`str_sub("abcde", 2, 50)`
- 「cde」が返されます。

●`str_sub("abcde", -5, 50)`
- error：index が範囲外

sync()

現在のフレームでスレッドが所有している残りの「物理」時間を使い果たします。

textmsg(*s1, s2=”*)

ログにテキスト メッセージを送信します。

GUI ログ タブに表示する、s1 と s2 が結合されたメッセージを送信します。

パラメータ

`s1`： メッセージ文字列。他の型（int、bool、pose など）の変数も送信できます。

`s2`： メッセージ文字列。他の型（int、bool、pose など）の変数も送信できます。

コマンドの例：`textmsg("value=", 3)`

●パラメータの例：

- s1 はメッセージの最初の部分を「value=」に設定します。
- s2 はメッセージの 2 つ目の部分を 3 に設定します。

＊ログのメッセージは「value=3」です。

to_num(*str*)

文字列を数値に変換します。

to_num は、入力文字列に小数点があるかどうかによって、整数または浮動小数点数を返します。ロケール設定にかかわらず、「.」のみが小数点として認識されます。

有効な文字列には、任意の先頭のホワイト スペースに続く任意のプラス（+）またはマイナス（-）記号と、以下のいずれかを含めることができます。

(i) 一連の 10 進数（10、-5 など）、浮動小数点数（1.5234、-2.0、.36 など）を示す任意の「.」、および 10 の累乗による乗数を示す任意の 10 進指数（10e3、2.5E-5、-5e-4 など）からなる 10 進数値。

(ii) 「0x」または「0X」の後に空ではない一連の 16 進が続く 16 進数値（「0X3A」、「0xb5」など）。

(iii) 無限（「INF」または「INFINITY」、大文字と小文字は区別されます）

(iv) 非数（「NAN」、大文字と小文字は区別されます）

ソース文字列に有効な数値が含まれていない場合や結果が結果の型の範囲外の場合、実行時例外が発生します。

パラメータ

str： 変換する文字列

戻り値

入力文字列に従った整数値または浮動小数点数値。

コマンドの例：

●to_num("10")
- 整数値の 10 が返されます。

●to_num("3.14")
- 浮動小数点数の 3.14 が返されます。

●to_num("-3.0e5")
- 入力文字列の「.」によって、浮動小数点数の-3.0e5 が返されます。

●to_num("+5.")
- 入力文字列の「.」によって、浮動小数点数の 5.0 が返されます。

●to_num("123abc")
- エラー文字列には有効な数値が含まれていません。

to_str(*val*)

値の文字列表現を取得します。

このスクリプトは、Boolean、Integer、Float、Pose の値（またはこれらの型のリスト）を文字列に変換します。

得られる文字列は 1023 文字を超えることはできません。

Float の数値は 6 桁の小数にフォーマットされ、末尾のゼロは削除されます。

パラメータ

`val`： 変換する値

戻り値

指定された値の文字列表現。

コマンドの例：

●`to_str(10)`

- 「10」が返されます。

●`to_str(2.123456123456)`

- 「2.123456」が返されます。

●`to_str(p[1.0, 2.0, 3.0, 4.0, 5.0, 6.0])`

- 「p[1, 2, 3, 4, 5, 6]」が返されます。

●`to_str([True, False, True])`

- 「[True, False, True]」が返されます。

3.2 変数

名前	説明
__package__	**値**：`None`

4 urmath モジュール

4.1 関数

acos(*f*)

f のアーク コサインが返されます。

f のアーク コサインの主値をラジアン表現で返します。f が範囲[-1, 1]の範囲外の場合、実行時エラーが発生します。

パラメータ

`f`： 浮動小数点数値

戻り値

f のアーク コサイン。

コマンドの例：`acos(0.707)`

●パラメータの例：

- f は 45 度（.785 ラジアン）のコサインです。
  - ＊.785 が返されます。

asin(*f*)

f のアーク サインが返されます。

f のアーク サインの主値をラジアン表現で返します。f が範囲[-1, 1]の範囲外の場合、実行時エラーが発生します。

パラメータ

`f`： 浮動小数点数値

戻り値

f のアーク サイン。

コマンドの例：`asin(0.707)`

●パラメータの例：

- f は 45 度（.785 ラジアン）のサインです。
  - ＊.785 が返されます。

atan(*f*)

f のアーク タンジェントが返されます。

f のアーク タンジェントの主値をラジアン表現で返します。

パラメータ

`f`： 浮動小数点数値

戻り値

f のアーク タンジェント。

コマンドの例：`atan(1.)`

●パラメータの例：

- f は 45 度（.785 ラジアン）のタンジェントです。
  - ＊.785 が返されます。

atan2(*x, y*)

x/y のアーク タンジェントが返されます。

x/y のアーク タンジェントの主値をラジアン表現で返します。値を計算するため、関数は両方の引数の符号を使用して象限を判断します。

パラメータ

`x`： 浮動小数点数値

`y`： 浮動小数点数値

戻り値

x/y のアーク タンジェント。

コマンドの例：`atan2(.5,.5)`

●パラメータの例：

- x は三角形の一辺です。
- y は三角形の 2 つ目の一辺です。
  - ＊atan(.5/.5)＝.785 が返されます。

binary_list_to_integer(*l*)

リスト l の内容によって表される値が返されます。

符号付きバイナリ値として評価される場合、リスト l に含まれるブール値によって表される整数値が返されます。

パラメータ

`l`： 整数に変換されるブール値のリスト。インデックス 0 のブール値は最下位ビットとして評価されます。False は 0 を表し、True は 1 を表します。リストが空の場合、関数は 0 を返します。リストに 32 個以上の bool 値が含まれる場合、関数はリストの最初の 32 個のブール値の符号付き整数値を返します。

戻り値

バイナリ リストの内容の整数値。

コマンドの例：

```
binary_list_to_integer([True,False,False,True])
```

●パラメータの例：

- l はバイナリ値 1001 を示します。

＊9 が返されます。

ceil(*f*)

f 以上の最小の整数値が返されます。

浮動小数点数を f 以上の最小の整数値に丸めます。

パラメータ

`f`： 浮動小数点数値

戻り値

丸められた整数

コマンドの例：`ceil(1.43)`

●パラメータの例：

- 2 が返されます。

cos(*f*)

f のコサインが返されます。

f ラジアンの角度のコサインが返されます。

パラメータ

`f`： 浮動小数点数値

戻り値

f のコサイン。

コマンドの例：`cos(1.57)`

●パラメータの例：

- f は 1.57 ラジアン（90 度）の角度です。
  ＊0.0 が返されます。

d2r(*d*)

d を度数単位からラジアン単位に変換したものが返されます。

「d」度のラジアン値が返されます。実際の計算式：(d/180)*MATH_PI

パラメータ

`d`： 度数単位の角度

戻り値

ラジアン単位の角度

コマンドの例：`d2r(90)`

●パラメータの例：

- d：度数単位の角度
  ＊ラジアン単位の角度 1.57 が返されます。

floor(*f*)

f 以下の最大の整数値が返されます。

浮動小数点数を f 以下の最大の整数値に丸めます。

パラメータ

`f`： 浮動小数点数値

戻り値

丸められた整数

コマンドの例：`floor(1.53)`

●パラメータの例：

- 1 が返されます。

get_list_length(*v*)

リスト変数の長さを返します。

リストの長さはリストを構成するエントリの数です。

パラメータ

`v`： リスト変数

戻り値

指定されたリストの長さを示す整数値。

コマンドの例：`get_list_length([1,3,3,6,2])`

●パラメータの例：

- v はリスト 1,3,3,6,2 です。
  ＊5 が返されます。

integer_to_binary_list(*x*)

x のバイナリ表現が返されます。

符号付き整数値 x のバイナリ表現として、ブール値のリストを返します。

パラメータ

`x`： バイナリ リストに変換される整数値。

戻り値

32 個のブール値のリスト、False は 0 を表し、True は 1 を表します。
インデックス 0 のブール値は最下位ビットです。

コマンドの例：`integer_to_binary_list(57)`

●パラメータの例：

- x：整数値 57
  ＊バイナリ リストが返されます。

interpolate_pose(*p_from, p_to, alpha*)

ツール位置および方向の線形補間。

alpha が 0 の場合、p_from が返されます。alpha が 1 の場合、p_to が返されます。alpha が 0 から 1 になると、p_from から p_to へ直線で向かう（および測地方向を変化させる）ポーズが返されます。alpha が 0 未満の場合、直線上の p_from より前の点が返されます。alpha が 1 を超えるの場合、直線上の p_to より後のポーズが返されます。

パラメータ

`p_from`： ツール ポーズ（pose）

`p_to`： ツール ポーズ（pose）

`alpha`： 浮動小数点数値

戻り値

線形補間されたポーズ（pose）

コマンドの例：`interpolate_pose(p[.2,.2,.4,0,0,0], p[.2,.2,.6,0,0,0], .5)`

●パラメータの例：

- p_from＝p[.2,.2,.4,0,0,0]
- p_to＝p[.2,.2,.6,0,0,0]
- alpha＝.5

＊p[.2,.2,.5,0,0,0]が返されます。

length(*v*)

リスト変数または文字列の長さを返します。

リストまたは文字列の長さは、それを構成するエントリまたは文字数です。

パラメータ

`v`： リストまたは文字列変数

戻り値

指定されたリストまたは文字列の長さを示す整数値。

コマンドの例：`length("here I am")`

●パラメータの例：

- v は文字列「here I am」と等しくなります

＊9 が返されます。

log(*b, f*)

底 b の f の対数が返されます。

底 b の f の対数が返されます。b または f が負の場合、または b が 1 の場合、実行時エラーが発生します。

パラメータ

`b`： 浮動小数点数値

`f`： 浮動小数点数値

戻り値

底 b の f の対数

コマンドの例：`log(10, 4.)`

●パラメータの例：

- b は底 10 です。
- f は 4 の対数です。
  ＊0.60206 が返されます。

norm(*a*)

引数のノルムが返されます。

引数は 4 つの型のいずれかとなります。

Pose：この場合、ポーズのユークリッド ノルムが返されます。

Float：この場合、fabs(a)が返されます。

Int：この場合、abs(a)が返されます。

List：この場合、リストのユークリッド ノルムが返されます。リストの要素は数値である必要があります。

パラメータ

`a`： Pose、float、int または List

戻り値

a のノルム

コマンドの例：

●`norm(-5.3)`→5.3 が返されます。

●`norm(-8)`→8 が返されます。

●`norm(p[-.2,.2,-.2,-1.57,0,3.14])`→3.52768 が返されます。

point_dist(*p_from, p_to*)

点の距離

パラメータ

`p_from`： ツール ポーズ（pose）

`p_to`： ツール ポーズ（pose）

戻り値

2 つのツール位置間の距離（回転を考慮しない）

コマンドの例：`point_dist(p[.2,.5,.1,1.57,0,3.14], p[.2,.5,.6,0,1.57,3.14])`

●パラメータの例：

- p_from＝p[.2,.5,.1,1.57,0,3.14] →1 つ目の点
- p_to＝p[.2,.5,.6,0,1.57,3.14] →2 つ目の点

＊回転に関係なく、2 点間の距離が返されます。

pose_add(*p_1, p_2*)

ポーズの加算

両方の引数に、P という 3 つの位置パラメータ(x, y, z)と、R という 3 つの回転パラメータ(R_x, R_y, R_z)が含まれます。この関数は、指定されたポーズの加算として、以下のように結果の x_3 を計算します。

p_3.P = p_1.P + p_2.P

p_3.R = p_1.R * p_2.R

パラメータ

`p_1`： ツール ポーズ 1（pose）

`p_2`： ツール ポーズ 2（pose）

戻り値

位置部分と回転部分の積の合計（pose）

コマンドの例：`pose_add(p[.2,.5,.1,1.57,0,0], p[.2,.5,.6,1.57,0,0])`

●パラメータの例：

- p_1＝p[.2,.5,.1,1.57,0,0] →1 つ目の点
- p_2＝p[.2,.5,.6,1.57,0,0] →2 つ目の点

＊p[0.4,1.0,0.7,3.14,0,0]が返されます。

pose_dist(*p_from, p_to*)

ポーズの距離

パラメータ

`p_from`： ツール ポーズ（pose）

`p_to`： ツール ポーズ（pose）

戻り値

距離

コマンドの例：`pose_dist(p[.2,.5,.1,1.57,0,3.14], p[.2,.5,.6,0,1.57,3.14])`

●パラメータの例：

- p_from＝p[.2,.5,.1,1.57,0,3.14] →1 つ目の点
- p_to＝p[.2,.5,.6,0,1.57,3.14] →2 つ目の点

＊回転を含む 2 つのポーズ間の距離が返されます。

pose_inv(*p_from*)

ポーズの逆数を取得します。

パラメータ

`p_from`： ツール ポーズ（空間ベクトル）

戻り値

逆ツール ポーズ変換（空間ベクトル）

コマンドの例：`pose_inv(p[.2,.5,.1,1.57,0,3.14])`

●パラメータの例：

- p_from＝p[.2,.5,.1,1.57,0,3.14] →点

＊p[0.19324,0.41794,-0.29662,1.23993,0.0,2.47985]が返されます。

pose_sub(*p_to, p_from*)

ポーズの減算

パラメータ

`p_to`： ツール ポーズ（空間ベクトル）

`p_from`： ツール ポーズ（空間ベクトル）

戻り値

ツール ポーズ変換（空間ベクトル）

コマンドの例：`pose_sub(p[.2,.5,.1,1.57,0,0], p[.2,.5,.6,1.57,0,0])`

●パラメータの例：

- p_1＝p[.2,.5,.1,1.57,0,0] →1 つ目の点
- p_2＝p[.2,.5,.6,1.57,0,0] →2 つ目の点

＊p[0.0,0.0,-0.5,0.0,.0.,0.0]が返されます。

pose_trans(*p_from, p_from_to*)

ポーズ変換

第 1 引数 p_from は、第 2 引数 p_from_to を変換するために使用され、その後結果が返されます。つまり、結果は、p_from の座標系から始まり、その座標系で p_from_to を移動したときに得られるポーズです。

この関数は、2 種類の表示で見ることができます。この関数は、p_from のパラメータによって、p_from_to を変換、つまり平行移動および回転させます。または、この関数は、最初に p_from の移動を実行し、そこから p_from_to の移動を実行した場合に得られるポーズを取得するために使用されます。

ポーズが変換行列とみなされる場合、以下のようになります。

T_world->to = T_world->from * T_from->to

T_x->to = T_x->from * T_from->to

パラメータ

`p_from`： 開始ポーズ（空間ベクトル）

`p_from_to`：開始ポーズを基準としたポーズの変化（空間ベクトル）

戻り値

得られるポーズ（空間ベクトル）

コマンドの例：`pose_trans(p[.2,.5,.1,1.57,0,0], p[.2,.5,.6,1.57,0,0])`

●パラメータの例：

- p_1＝p[.2,.5,.1,1.57,0,0] →1 つ目の点
- p_2＝p[.2,.5,.6,1.57,0,0] →2 つ目の点

＊p[0.4,-0.0996,0.60048,3.14,0.0,0.0]が返されます。

pow(*base, exponent*)

基数に指数をべき乗した数を返します。

基数に指数をべき乗した結果を返します。基数が負であり指数が整数値ではない場合、または基数がゼロで指数が負の場合、実行時エラーが発生します。

パラメータ

`base`： 浮動小数点数値

`exponent`： 浮動小数点数値

戻り値

基数に指数をべき乗した値

コマンドの例：`pow(5.,3)`

●パラメータの例：

- Base＝5
- Exponent＝3

＊125 が返されます。

r2d(*r*)

r をラジアン単位から度数単位に変換したものが返されます。

「r」ラジアンの度数値を返します。

パラメータ

`r`： ラジアン単位の角度

戻り値

度数単位の角度

コマンドの例：`r2d(1.57)`

●パラメータの例：

- r：1.5707 ラジアン

＊90 度が返されます。

random()

ランダムな数値

戻り値

0 から 1 までの疑似乱数値（float）

rotvec2rpy(*rotation_vector*)

rotation_vector に対応する RPY ベクトルが返されます。

「rotation_vector」に対応する RPY ベクトルが返されます。回転ベクトルはラジアン単位の回転角度に対応する長さの回転の軸です。

パラメータ

`rotation_vector`： ラジアン単位の回転ベクトル（Vector3d）。これは軸角度ベクトルとも呼ばれます（回転の単位軸にラジアン単位の回転角度を乗算したもの）。

戻り値

ラジアン単位の RPY ベクトル（Vector3d）。X-Y-Z 軸を中心としたロール - ピッチ - ヨーの一連の外部回転を表します（Z-Y'-X"軸を中心とした内部回転に対応します）。行列形式では、RPY ベクトルは Rrpy＝Rz(yaw)Ry(pitch)Rx(roll)と定義されます。

コマンドの例：`rotvec2rpy([3.14,1.57,0])`

●パラメータの例：

- rotation_vector＝[3.14,1.57,0] →rx＝3.14、ry＝1.57、rz＝0
  - ＊[2.80856, .16202, 0.9] →ロール＝2.80856、ピッチ＝.16202、ヨー＝0.9 が返されます。

rpy2rotvec(*rpy_vector*)

rpy_vector に対応する回転ベクトルが返されます。

「rpy_vector」に対応する回転ベクトルが返されます。RPY（ロール - ピッチ - ヨー）回転は X-Y-Z 軸を中心とした外部回転です（Z-Y'-X"軸を中心とした内部回転に対応します）。

パラメータ

`rpy_vector`： ラジアン単位の RPY ベクトル（Vector3d）。X-Y-Z 軸を中心としたロール - ピッチ - ヨーの一連の外部回転を表します（Z-Y'-X"軸を中心とした内部回転に対応します）。行列形式では、RPY ベクトルは Rrpy＝Rz(yaw)Ry(pitch)Rx(roll)と定義されます。

戻り値

ラジアン単位の回転ベクトル（Vector3d）。これは軸角度ベクトルとも呼ばれます（回転の単位軸にラジアン単位の回転角度を乗算したもの）。

コマンドの例：`rpy2rotvec([3.14,1.57,0])`

●パラメータの例：

- rpy_vector＝[3.14,1.57,0] →ロール＝3.14、ピッチ＝1.57、ヨー＝0
  - ＊[2.22153, 0.00177, -2.21976] →rx＝2.22153、ry＝0.00177、rz＝-2.21976 が返されます。

sin(*f*)

f のサインが返されます。

f ラジアンの角度のサインが返されます。

パラメータ

`f`： 浮動小数点数値

戻り値

f のサイン。

コマンドの例：`sin(1.57)`

●パラメータの例：

- f は 1.57 ラジアン（90 度）の角度です。
  - ＊1.0 が返されます。

sqrt(*f*)

f の平方根が返されます。

f の平方根が返されます。f が負の場合、実行時エラーが発生します。

パラメータ

`f`： 浮動小数点数値

戻り値

f の平方根。

コマンドの例：`sqrt(9)`

●パラメータの例：

- f＝9

＊3 が返されます。

tan(*f*)

f のタンジェントが返されます。

f ラジアンの角度のタンジェントが返されます。

パラメータ

`f`： 浮動小数点数値

戻り値

f のタンジェント。

コマンドの例：`tan(.7854)`

●パラメータの例：

- f は.7854 ラジアン（45 度）の角度です。

＊1.0 が返されます。

wrench_trans(*T_from_to, w_from*)

レンチ変換

レンチの視点を移動します。

注：トルクは平行移動の長さでスケーリングされるため、レンチの変換はポーズの変換ほど簡単ではありません。

w_to = T_from->to * w_from

パラメータ

`T_from_to`：新しい視点への変換（pose）

`w_from`：　リスト形式の変換するレンチ[F_x, F_y, F_z, M_x, M_y, M_z]

戻り値

リスト形式の得られるレンチ w_to　[F_x, F_y, F_z, M_x, M_y, M_z]

4.2　変数

名前	説明
__package__	**値**：`None`

5 インターフェース モジュール

5.1 関数

get_analog_in(*n*)

*廃止予定：*アナログ入力信号レベルを取得します。

パラメータ

n： 入力の番号（id）、整数：[0:3]

戻り値

float、アンペアまたはボルト単位の信号レベル

廃止予定：この関数は `get_standard_analog_in` および `get_tool_analog_in` に置き換えられます。後者の関数では、ポート 2～3 を 0～1 に変更する必要があります。この関数は次のメジャー リリースで削除される場合があります。

注：n：2～3 の後方互換性については、ツール アナログ入力に移動します。

コマンドの例：`get_analog_in(1)`

●パラメータの例：

- n はアナログ入力 1 です。
  - アナログ出力#1 の値が返されます。

get_analog_out(*n*)

*廃止予定：*アナログ出力信号レベルを取得します。

パラメータ

n： 出力の番号（id）、整数：[0:1]

戻り値

float、アンペアまたはボルト単位の信号レベル

廃止予定：この関数は `get_standard_analog_out` に置き換えられます。この関数は次のメジャー リリースで削除される場合があります。

コマンドの例：`get_analog_out(1)`

●パラメータの例：

- n はアナログ出力 1 です。
  - アナログ出力#1 の値が返されます。

get_configurable_digital_in(*n*)

設定可能デジタル入力信号レベルを取得します。

`get_standard_digital_in` および `get_tool_digital_in` も参照してください。

パラメータ

`n`： 入力の番号（id）、整数：[0:7]

戻り値

ブール値、信号レベル。

コマンドの例： `get_configurable_digital_in(1)`

●パラメータの例：

- n は設定可能デジタル入力 1 です。
  - ＊True または False が返されます。

get_configurable_digital_out(*n*)

設定可能デジタル出力信号レベルを取得します。

`get_standard_digital_out` および `get_tool_digital_out` も参照してください。

パラメータ

`n`： 出力の番号（id）、整数：[0:7]

戻り値

ブール値、信号レベル。

コマンドの例： `get_configurable_digital_out(1)`

●パラメータの例：

- n は設定可能デジタル出力 1 です。
  - ＊True または False が返されます。

get_digital_in(*n*)

*廃止予定：*デジタル入力信号レベルを取得します。

パラメータ

n： 入力の番号（id）、整数：[0:9]

戻り値

ブール値、信号レベル。

廃止予定：この関数は `get_standard_digital_in` および `get_tool_digital_in` に置き換えられます。後者の関数では、ポート 8～9 を 0～1 に変更する必要があります。この関数は次のメジャー リリースで削除される場合があります。

注：n：8～9 の後方互換性については、ツール デジタル入力に移動します。

コマンドの例：`get_digital_in(1)`

●**パラメータの例**：

- n はデジタル入力 1 です。

＊True または False が返されます。

get_digital_out(*n*)

*廃止予定：*デジタル出力信号レベルを取得します。

パラメータ

n： 出力の番号（id）、整数：[0:9]

戻り値

ブール値、信号レベル。

廃止予定：この関数は `get_standard_digital_out` および `get_tool_digital_out` に置き換えられます。後者の関数では、ポート 8～9 を 0～1 に変更する必要があります。この関数は次のメジャー リリースで削除される場合があります。

注：n：8～9 の後方互換性については、ツール デジタル出力に移動します。

コマンドの例：`get_digital_out(1)`

●パラメータの例：

- n はデジタル出力 1 です。

＊True または False が返されます。

get_flag(*n*)

フラグは内部デジタル出力のように動作します。情報はプログラム実行間で維持されます。

パラメータ

n： フラグの番号（id）、整数：[0:32]

戻り値

ブール値、格納済みビット。

コマンドの例：`get_flag(1)`

●パラメータの例：

- n はフラグ番号 1 です。
  - ＊True または False が返されます。

get_standard_analog_in(*n*)

標準アナログ入力信号レベルを取得します。

get_tool_analog_in も参照してください。

パラメータ

n： 入力の番号（id）、整数：[0:1]

戻り値

float、アンペアまたはボルト単位の信号レベル

コマンドの例：`get_standard_analog_in(1)`

●パラメータの例：

- n は標準アナログ入力 1 です。
  - ＊標準アナログ入力#1 の値が返されます。

get_standard_analog_out(*n*)

標準アナログ出力信号レベルを取得します。

パラメータ

n： 出力の番号（id）、整数：[0:1]

戻り値

float、アンペアまたはボルト単位の信号レベル

コマンドの例：`get_standard_analog_out(1)`

●パラメータの例：

- n は標準アナログ出力 1 です。
  - ＊標準アナログ出力#1 の値が返されます。

get_standard_digital_in(*n*)

標準デジタル入力信号レベルを取得します。

`get_configurable_digital_in` および `get_tool_digital_in` も参照してください。

パラメータ

`n`： 入力の番号（id）、整数：[0:7]

戻り値

ブール値、信号レベル。

コマンドの例：`get_standard_digital_in(1)`

●パラメータの例：

- n は標準デジタル入力 1 です。
  - ＊True または False が返されます。

get_standard_digital_out(*n*)

標準デジタル出力信号レベルを取得します。

`get_configurable_digital_out` および `get_tool_digital_out` も参照してください。

パラメータ

`n`： 出力の番号（id）、整数：[0:7]

戻り値

ブール値、信号レベル。

コマンドの例：`get_standard_digital_out(1)`

●パラメータの例：

- n は標準デジタル出力 1 です。
  - ＊True または False が返されます。

get_tool_analog_in(*n*)

ツール アナログ入力信号レベルを取得します。

`get_standard_analog_in` も参照してください。

パラメータ

`n`： 入力の番号（id）、整数：[0:1]

戻り値

float、アンペアまたはボルト単位の信号レベル

コマンドの例：`get_tool_analog_in(1)`

●パラメータの例：

- n はツール アナログ入力 1 です。
  - ツール アナログ入力#1 の値が返されます。

get_tool_digital_in(*n*)

ツール デジタル入力信号レベルを取得します。

`get_configurable_digital_in` および `get_standard_digital_in` も参照してください。

パラメータ

`n`： 入力の番号（id）、整数：[0:1]

戻り値

ブール値、信号レベル。

コマンドの例：`get_tool_digital_in(1)`

●パラメータの例：

- n はツール デジタル入力 1 です。
  - True または False が返されます。

get_tool_digital_out(*n*)

ツール デジタル出力信号レベルを取得します。

`get_standard_digital_out` および
`get_configurable_digital_out` も参照してください。

パラメータ

`n`： 出力の番号（id）、整数：[0:1]

戻り値

ブール値、信号レベル。

コマンドの例：`get_tool_digital_out(1)`

●パラメータの例：

- n はツール デジタル出力 1 です。
  - ＊True または False が返されます。

get_tool_digital_output_mode(*n*)

ツール出力ピンの出力モードを取得します。

パラメータ

`n`： 出力の番号（id）、整数：[0:1]

戻り値

整数[1:3]、ピン出力モード。1＝シンク／NPN、2＝ソース／PNP、3＝プッシュ - プル

コマンドの例：get_tool_digital_output_mode(1)

●n はツール デジタル出力 1 です。

- 整数[1:3]が返されます。

get_tool_output_mode()

ツール デジタル出力モードを取得します。

戻り値

整数[0:1]：0＝デジタル出力モード、1＝電源（デュアル ピン）モード

modbus_add_signal(*IP, slave_number, signal_address, signal_type, signal_name, sequential_mode*=`False`)

コントローラーが監視する新しい modbus 信号を追加します。期待されるレスポンスはありません。

```
>>> modbus_add_signal("172.140.17.11", 255, 5, 1, "output1")
```

パラメータ

`IP`：	modbus 信号が接続される modbus ユニットの IP アドレスを指定する文字列。
`slave_number`：	通常は使用されず 255 に設定された整数ですが、0 から 255 の間で自由に選択できます。
`signal_address`：	この新しい信号が反映すべきコイルまたはレジスタのアドレスを指定する整数。この情報については、modbus ユニットの構成を確認してください。
`signal_type`：	追加する信号の種類を指定する整数。0＝デジタル入力、1＝デジタル出力、2＝レジスタ入力、3＝レジスタ出力。
`signal_name`：	信号を一意に識別する文字列。すでに追加されている信号と等しい文字列が指定された場合、新しい信号によって古い信号が置き換えられます。
`sequential_mode`：	True に設定すると、modbus クライアントが次のリクエストの送信までレスポンスを待機するよう強制されます。このモードは、一部のフィールドバス ユニットで必要です（任意）。

コマンドの例：`modbus_add_signal("172.140.17.11", 255, 5, 1, "output1")`

●パラメータの例：

- IP アドレス＝172.140.17.11
- スレーブ番号＝255
- 信号アドレス＝5
- 信号タイプ＝1 デジタル出力
- 信号名＝output1

modbus_delete_signal(*signal_name*)

指定された信号名で識別される信号を削除します。

```
>>> modbus_delete_signal("output1")
```

パラメータ

`signal_name`： 削除されるべき信号の名前と等しい文字列。

コマンドの例：`modbus_delete_signal("output1")`

●パラメータの例：

- 信号名＝output1

modbus_get_signal_status(*signal_name, is_secondary_program*)

特定の信号の現在の値を読み取ります。

```
>>> modbus_get_signal_status("output1",False)
```

パラメータ

`signal_name`： 値を取得する信号の名前と等しい文字列。

`is_secondary_program`： 内部使用専用のブール値。False に設定する必要があります。

戻り値

整数またはブール値。デジタル信号の場合：True または False。レジスタ信号の場合：符号なし整数として表現されるレジスタ値。

コマンドの例：`modbus_get_signal_status("output1",False)`

●パラメータの例：

- 信号名＝output1
- Is_secondary_program＝False（注：False に設定する必要があります）

modbus_send_custom_command(*IP, slave_number, function_code, data*)

ユーザーによって指定されたコマンドを、指定された IP アドレスにある modbus ユニットに送信します。レスポンスが受信されないため、データをリクエストするのには使用できません。ユーザーは指定された関数コードに対して意味のあるデータを指定します。組み込み関数が modbus フレームの構築を処理するため、ユーザーがコマンドの長さに留意する必要はありません。

```
>>> modbus_send_custom_command("172.140.17.11",103,6,
>>>                [17,32,2,88])
```

上記の例は、Beckhoff BK9050 のウォッチドック タイムアウトを 600ms に設定しています。modbus 関数コード 6（プレセット信号レジスタ）を使用し、データ配列の最初の 2 バイトにレジスタ アドレス（[17,32] = [0x1120]）を、最後の 2 バイトに目的のレジスタの内容（[2,88] = [0x0258] = dec 600）を指定することで、これを実行します。

パラメータ

`IP`：	カスタム コマンドを送信すべき modbus ユニットを特定する IP アドレスを指定する文字列。
`slave_number`：	カスタム コマンドに使用するスレーブ番号を指定する整数。
`function_code`：	カスタム コマンドの関数コードを指定する整数。
`data`：	各エントリが有効なバイト（0～255）値である必要がある整数の配列。

コマンドの例：`modbus_send_custom_command("172.140.17.11", 103, 6, [17,32,2,88])`

●パラメータの例：

- IP アドレス＝172.140.17.11
- スレーブ番号＝103
- 関数コード＝6
- データ＝[17,32,2,88]
  - ＊関数コードとデータは、UR コントローラーに接続されているスレーブ Modbus デバイスの製造元によって指定されています。

modbus_set_digital_input_action(*signal_name, action*)

選択したデジタル入力信号を「default」または「freedrive」アクションに設定します。

```
>>> modbus_set_digital_input_action("input1", "freedrive")
```

パラメータ

`signal_name`： 以前追加したデジタル入力信号を識別する文字列。

`action`： アクションのタイプ。アクションは「default」または「freedrive」のいずれかとなります。（string）

コマンドの例：

```
modbus_set_digital_input_action("input1", "freedrive")
```

●パラメータの例：

- 信号名＝「input1」
- アクション＝「freedrive」

modbus_set_output_register(*signal_name, register_value, is_secondary_program*)

指定された名前で識別される出力レジスタ信号を指定された値に設定します。

```
>>> modbus_set_output_register("output1",300,False)
```

パラメータ

`signal_name`： 事前に追加した出力レジスタ信号を識別する文字列。

`register_value`： 有効なワード（0～65535）値である必要がある整数。

`is_secondary_program`： 内部使用専用のブール値。False に設定する必要があります。

コマンドの例：

```
modbus_set_output_register("output1", 300, False)
```

●パラメータの例：

- 信号名＝output1
- レジスタ値＝300
- Is_secondary_program＝False（注：False に設定する必要があります）

modbus_set_output_signal(*signal_name, digital_value, is_secondary_program*)

指定された名前で識別される出力デジタル信号を指定された値に設定します。

```
>>> modbus_set_output_signal("output2",True,False)
```

パラメータ

signal_name：	事前に追加した出力デジタル信号を識別する文字列。
digital_value：	信号を設定する値のブール値。
is_secondary_program：	内部使用専用のブール値。False に設定する必要があります。

コマンドの例：`modbus_set_output_signal("output1", True, False)`

●パラメータの例：

- 信号名＝output1
- デジタル値＝True
- Is_secondary_program＝False（注：False に設定する必要があります）

modbus_set_signal_update_frequency(*signal_name, update_frequency*)

ロボットが Modbus コントローラーに信号値を読み取るまたは書き込むためのリクエストを送信する頻度を設定します。

```
>>> modbus_set_signal_update_frequency("output2",20)
```

パラメータ

signal_name：	事前に追加した出力デジタル信号を識別する文字列。
update_frequency：	Hz 単位で更新頻度を指定する範囲 0～500 の整数。

コマンドの例：

```
modbus_set_signal_update_frequency("output2", 20)
```

●パラメータの例：

- 信号名＝output2
- 信号更新頻度＝20Hz

read_input_boolean_register(*address*)

フィールド バスもアクセスできる入力レジスタのいずれかからブール値を読み取ります。独自のメモリ空間が使用されることに注意してください。

```
>>> bool_val = read_input_boolean_register(3)
```

パラメータ

address： レジスタのアドレス（0:63）

戻り値

レジスタによって保持されるブール値（True、False）

コマンドの例：`read_input_boolean_register(3)`

●パラメータの例：

- アドレス＝入力ブール レジスタ 3

read_input_float_register(*address*)

フィールド バスもアクセスできる入力レジスタのいずれかから浮動小数点数を読み取ります。独自のメモリ空間が使用されることに注意してください。

```
>>> float_val = read_input_float_register(3)
```

パラメータ

address： レジスタのアドレス（0:23）

戻り値

レジスタによって保持される値（float）

コマンドの例：`read_input_float_register(3)`

●パラメータの例：

- アドレス＝入力浮動小数点数レジスタ 3

read_input_integer_register(*address*)

フィールド バスもアクセスできる入力レジスタのいずれかから整数を読み取ります。独自のメモリ空間が使用されることに注意してください。

```
>>> int_val = read_input_integer_register(3)
```

パラメータ

address： レジスタのアドレス（0:23）

戻り値

レジスタによって保持される値[-2,147,483,648 : 2,147,483,647]

コマンドの例：`read_input_integer_register(3)`

●パラメータの例：

- アドレス＝入力整数レジスタ 3

read_output_boolean_register(*address*)

フィールド バスもアクセスできる出力レジスタのいずれかからブール値を読み取ります。独自のメモリ空間が使用されることに注意してください。

```
>>> bool_val = read_output_boolean_register(3)
```

パラメータ

address： レジスタのアドレス（0:63）

戻り値

レジスタによって保持されるブール値（True、False）

コマンドの例：`read_output_boolean_register(3)`

●パラメータの例：

- アドレス＝出力ブール レジスタ 3

read_output_float_register(*address*)

フィールド バスもアクセスできる出力レジスタのいずれかから浮動小数点数を読み取ります。独自のメモリ空間が使用されることに注意してください。

```
>>> float_val = read_output_float_register(3)
```

パラメータ

address： レジスタのアドレス（0:23）

戻り値

レジスタによって保持される値（float）

コマンドの例：`read_output_float_register(3)`

●パラメータの例：

- アドレス＝出力浮動小数点数レジスタ 3

read_output_integer_register(*address*)

フィールド バスもアクセスできる出力レジスタのいずれかから整数を読み取ります。独自のメモリ空間が使用されることに注意してください。

```
>>> int_val = read_output_integer_register(3)
```

パラメータ

address： レジスタのアドレス（0:23）

戻り値

レジスタによって保持される整数値[-2,147,483,648 : 2,147,483,647]

コマンドの例：`read_output_integer_register(3)`

●パラメータの例：

- アドレス＝出力整数レジスタ 3

read_port_bit(*address*)

Modbus クライアントもアクセスできるいずれかのポートを読み取ります。

```
>>> boolval = read_port_bit(3)
```

パラメータ

`address`：ポートのアドレス（サポート サイトのページ「UsingModbusServer」のポートマップを参照してください。）

戻り値

ポートによって保持される値（True、False）

コマンドの例：`read_port_bit(3)`

●パラメータの例：

- アドレス＝ポート ビット 3

read_port_register(*address*)

Modbus クライアントもアクセスできるいずれかのポートを読み取ります。

```
>>> intval = read_port_register(3)
```

パラメータ

`address`：ポートのアドレス（サポート サイトのページ「UsingModbusServer」のポートマップを参照してください。）

戻り値

ポートによって保持される符号付き整数値（-32768 : 32767）

コマンドの例：`read_port_register(3)`

●パラメータの例：

- アドレス＝ポート レジスタ 3

rpc_factory(*type, url*)

新しいリモート プロシージャー コール（RPC）ハンドルを作成します。RPC の詳細な説明については、サブセクション 1.6 {リモート プロシージャー コール（RPC）}を参照してください。

```
>>> proxy = rpc_factory("xmlrpc",
>>> "http://127.0.0.1:8080/RPC2")
```

パラメータ

`type`：使用するように指示された RPC のタイプ。現在は「xmlrpc」プロトコルのみを使用できます。

`url`：RPC サーバーの URL。現在、2 つのプロトコルがサポートされています： pstream および http。pstream URL は「<ip-address>:<port>」のような形式です。たとえば、ポート 8080 でローカル接続を確立するには、「127.0.0.1:8080」となります。http URL は一般的に「http://<ip-address>:<port>/<path>」のような形式です。<path> は http サーバーのセットアップによって異なります。上記の例では、ポート 8080 でのローカルの Python Web サーバーへの接続が確立されています。これは、XMLRPC 呼び出しをパス「RPC2」で受信することを想定しています。

戻り値

指定された RPC バックエンドを使用して指定されたサーバーに接続している RPC ハンドル。サーバーが利用可能ではない場合、関数とプログラムは失敗します。このインスタンスを使用すると、サーバー上で利用可能な関数を呼び出すことができます。たとえば、「bool isTargetAvailable(int number, ...)」は「proxy.isTargetAvailable(var_1, ...)」となります。任意の数の引数がサポートされます（...で示されています）。

注：RPC インスタンスに適切な名前を指定すると、プログラムの可読性が向上します（「proxy」はあまり良い名前ではありません）。

コマンドの例：`rpc_factory("xmlrpc", "http://127.0.0.1:8080/RPC2")`

●パラメータの例：

- type＝xmlrpc
- url＝http://127.0.0.1:8080/RPC2

rtde_set_watchdog(*variable_name, min_frequency, action='pause'*)

この関数は、RTDE に対する特定の入力変数のウォッチドッグをアクティブ化します。ウォッチドッグが min_frequency（Hz）で指定された期間内に指定された変数の入力更新を受信しなかった場合、対応するアクションが実行されます。プログラムの停止時にすべてのウォッチドッグが削除されます。

```
>>> rtde_set_watchdog("input_int_register_0", 10, "stop")
```

パラメータ

`variable_name`：入力変数名（string）。RTDE インターフェースによって指定されます。

`min_frequency`：入力の更新の発生が期待される最低頻度（float）

`action`：任意：最低頻度の違反時に、プログラムを「ignore（無視）」、「pause（一時停止）」、「stop（停止）」します。デフォルトのアクションは「pause」です。

戻り値

なし

注：更新がない場合に指定されたアクションを保証するには、RTDE 入力パッケージあたりに必要なウォッチドッグは 1 つだけです。

コマンドの例：`rtde set watchdog( "input int register 0" , 10, "stop" )`

●パラメータの例：

- 変数名＝input int register 0
- 最低頻度＝10Hz
- アクション＝プログラムを停止（stop）

set_analog_inputrange(*port, range*)

*廃止予定：*アナログ入力の範囲を設定します。

ポート 0 および 1 はコントローラー ボックス内にあり、2 および 3 はツール コネクター内にあります。

パラメータ

`port`： アナログ入力ポート番号、0、1＝コントローラー、2、3＝ツール

`range`： *コントローラー* アナログ入力範囲 0：*0～5V*（範囲 2 に自動的にマッピング）、範囲 2：*0～10V*。

`range`： *ツール* アナログ入力範囲 0：*0～5V*（範囲 1 に自動的にマッピング）、1：*0～10V* および 2：*4～20mA*。

廃止予定：この関数は `set_standard_analog_input_domain` および `set_tool_analog_input_domain` に置き換えられます。後者の関数では、ポート 2～3 を 0～1 に変更する必要があります。この関数は次のメジャー リリースで削除される場合があります。

注：*コントローラー*入力範囲 1 の場合：-5～5V、3：-10～10V はサポートされなくなり、GUI では例外が表示されます。

set_analog_out(*n, f*)

*廃止予定：*アナログ出力信号レベルを設定します。

パラメータ

`n`： 出力の番号（id）、整数：[0:1]

`f`： 相対信号レベル[0;1]（float）

廃止予定：この関数は `set_standard_analog_out` に置き換えられます。この関数は次のメジャー リリースで削除される場合があります。

コマンドの例：`set_analog_out(1,0.5)`

●パラメータの例：

- n はアナログ出力ポート 1 です（コントローラー上）。
- f＝0.5、出力ポート上の 5V に対応（またはドメイン設定によっては 12mA）

set_configurable_digital_out(*n, b*)

設定可能デジタル出力信号レベルを設定します。

`set_standard_digital_out` および `set_tool_digital_out` も参照してください。

パラメータ

`n`： 出力の番号（id）、整数：[0:7]

`b`： 信号レベル。（boolean）

コマンドの例： `set_configurable_digital_out(1,True)`

●パラメータの例：

- n は設定可能デジタル出力 1 です。
- b＝True

set_digital_out(*n, b*)

*廃止予定：*デジタル出力信号レベルを設定します。

パラメータ

`n`： 出力の番号（id）、整数：[0:9]

`b`： 信号レベル。（boolean）

廃止予定：この関数は `set_standard_digital_out` および `set_tool_digital_out` に置き換えられます。後者の関数では、ポート 8～9 を 0～1 に変更する必要があります。この関数は次のメジャー リリースで削除される場合があります。

コマンドの例： `set_digital_out(1,True)`

●パラメータの例：

- n はデジタル出力 1 です。
- b＝True

set_flag(*n, b*)

フラグは内部デジタル出力のように動作します。情報はプログラム実行間で維持されます。

パラメータ

`n`： フラグの番号（id）、整数：[0:32]

`b`： 格納済みビット。（boolean）

コマンドの例： `set_flag(1,True)`

●パラメータの例：

- n はフラグ番号 1 です。
- b＝True はビットを True に設定します。

set_standard_analog_out(*n, f*)

標準アナログ出力信号レベルを設定します。

パラメータ

`n`： 出力の番号（id）、整数：[0:1]

`f`： 相対信号レベル[0;1]（float）

コマンドの例：`set_standard_analog_out(1,1.0)`

●パラメータの例：

- n は標準アナログ出力ポート 1 です。
- f＝1.0、出力ポート上の 10V に対応（またはドメイン設定によっては 20mA）

set_standard_digital_out(*n, b*)

標準デジタル出力信号レベルを設定します。

`set_configurable_digital_out` および `set_tool_digital_out` も参照してください。

パラメータ

`n`： 出力の番号（id）、整数：[0:7]

`b`： 信号レベル。（boolean）

コマンドの例：`set_standard_digital_out(1,True)`

●パラメータの例：

- n は標準デジタル出力 1 です。
- f＝True

set_tool_communication(*enabled, baud_rate, parity, stop_bits, rx_idle_chars, tx_idle_chars*)

この関数は、「ツール通信インターフェース」（TCI）をアクティブ化または非アクティブ化します。TCI は、ロボット アナログ入力を介して外部ツールとの通信を有効化することで、外部配線を回避します。

```
>>> set_tool_communication(True, 115200, 1, 2, 1.0, 3.5)
```

パラメータ

`enabled`：	TCI を有効化または無効化するブール値（string） 有効な値：True（有効化）または False（無効化）
`baud_rate`：	使用されるボーレート（int）。有効な値：9600、19200、38400、57600、115200、1000000、2000000、5000000。
`parity`：	使用されるパリティ（int）。有効な値：0（なし）、1（奇数）、2（偶数）。
`stop_bits`：	停止ビットの数（int）。有効な値：1、2。
`rx_idle_chars`：	メッセージが終了とマークされるまで／PC にメッセージを送信するまで、ツールの RX ユニットが待機する必要がある文字数（float）。有効な値：最小＝1.0、最大＝40.0。
`tx_idle_chars`：	バス上の前回のアクティビティ以降新しい転送が開始されるまで、ツールの TX ユニットが待機する必要がある文字数（float）。有効な値：最小＝0.0、最大＝40.0。

戻り値

なし

注：この機能を有効化すると、ロボット ツール アナログ入力が無効化されます。

コマンドの例：`set_tool_communication(True, 115200, 1, 2, 1.0, 3.5)`

●パラメータの例：

- 有効化＝True
- ボーレート＝115200
- パリティ＝ODD
- 停止ビット＝2
- rx アイドル時間＝1.0
- tx アイドル時間＝3.5

set_tool_digital_out(*n, b*)

ツール デジタル出力信号レベルを設定します。

`set_configurable_digital_out` および
`set_standard_digital_out` も参照してください。

アクティブ時の動作モードの説明については、マニュアルの「ツール デジタル出力」を参照してください。

パラメータ

`n`： 出力の番号（id）、整数：[0:1]

`b`： 信号のアクティブ化。（boolean）

コマンドの例：`set_tool_digital_out(1,True)`

●パラメータの例：

- n はツール デジタル出力 1 です。
- b＝True

set_tool_digital_output_mode(*n, mode*)

ツール出力ピンの出力モードを設定します。

パラメータ

`n`： 出力の番号（id）、整数：[0:1]

`mode`： ピン モード。整数：[1:3]。1＝シンク／NPN、2＝ソース／PNP、3＝プッシュ - プル

コマンドの例：`set_tool_digital_output_mode(0, 2)`

●パラメータの例：

- 0 はデジタル出力ピン 0 です。
- 2 はソース／PNP のピン モードです。1 に設定するとピンはソース 電流となり、0 に設定するとピンはハイ インピーダンスになります。

set_tool_output_mode(*mode*)

ツール デジタル出力モードを設定します。

パラメータ

`mode`： 設定する出力モード、整数：[0:1]。0＝デジタル出力モード、1＝電源（デュアル ピン）モード

コマンドの例：`set_tool_output_mode(1)`

●パラメータの例：

- 1 は電源（デュアル ピン）モードです。デジタル出力は追加電源として使用されます。

set_tool_voltage(*voltage*)

ロボットのツール フランジのコネクタ プラグに電力を供給する電源の電圧レベルを設定します。電圧は 0、12 または 24 ボルトとなります。

パラメータ

`voltage`： ツール コネクタでの電圧（整数）、整数：0、12 または 24。

コマンドの例：`set_tool_voltage(24)`

●パラメータの例：

- 電圧＝24 ボルト

socket_close(*socket_name='socket_0'*)

TCP/IP ソケット通信をクローズします。

サーバーとのソケット接続をクローズします。

```
>>> socket_comm_close()
```

パラメータ

`socket_name`： ソケットの名前（string）

コマンドの例：`socket_close(socket_name="socket_0")`

●パラメータの例：

- socket_name＝socket_0

socket_get_var(*name, socket_name='socket_0'*)

サーバーから整数を読み取ります。

ソケットを介してメッセージ「GET <name>\n」を送信します。2 秒以内に「<name> <int>\n」というレスポンスがあることが想定されます。タイムアウトすると 0 が返されます。

パラメータ

`name`： 変数名（string）

`socket_name`： ソケットの名前（string）

戻り値

サーバーからの整数（int）、0 はタイムアウトした場合の値です。

コマンドの例：`x_pos = socket_get_var("POS_X")`

socket_0 に GET POS_X\n を送信し、レスポンスが 2 秒以内にあることが想定されます。

●パラメータの例：

- name＝POS_X→変数の名前
- socket_name＝デフォルト：socket_0

socket_open(*address, port, socket_name=*'socket_0')

TCP/IP イーサネット通信をオープンします。

ソケット通信のオープンを試行します。2 秒後タイムアウトします。

パラメータ

address： サーバー アドレス（string）

port： ポート番号（int）

socket_name： ソケットの名前（string）

戻り値

失敗した場合は False、接続が正常に確立された場合は True。

注：使用されるネットワーク セットアップは、クライアント／サーバー通信のパフォーマンスに影響を与えます。たとえば、TCP/IP 通信は、基礎となるネットワーク インターフェースによってバッファされます。

コマンドの例：
```
socket_open("192.168.5.1", 50000, "socket_10")
```

●パラメータの例：

- address＝192.168.5.1
- port＝50000
- socket_name＝socket_10

socket_read_ascii_float(*number, socket_name=*'socket_0', *timeout=*2)

ソケットから複数の ascii 形式の浮動小数点数を読み取ります。1 コマンドで最大 30 件の値を読み取ることができます。

数値はかっこで囲まれ、「,」で区切られている形式です。4 つの数値のリストの例は「(1.414 , 3.14159, 1.616, 0.0)」のようになります。

返されるリストには、読み取る数値の合計数が含まれ、それに続いて各数値が連続して含まれます。たとえば、上記の例の read_ascii_float では、[4, 1.414, 3.14159, 1.616, 0.0]が返されます。

読み取りに失敗したり、タイムアウトしたりすると、最初の要素が 0、続く要素が「非数（nan）」となるリストが返されます（例：3 つの数値を読み取る場合は[0, nan, nan, nan]）。

パラメータ

number：	読み取る変数の数（int）。
socket_name：	ソケットの名前（string）
timeout：	読み取りアクションがタイムアウトするまでの秒数（float）timeout が 0 または負数の場合、関数が読み取りを完了するまでリターンしてはいけないことを示します。

戻り値

読み取った数値のリスト（浮動小数点数のリスト、長さ＝number＋1）

コマンドの例：
```
list_of_four_floats = socket_read_ascii_float(4,"socket_10")
```

●パラメータの例：

- number＝4 →読み取る浮動小数点数の数
- socket_name＝socket_10

＊リストを返します。

socket_read_binary_integer(*number, socket_name=*'socket_0', *timeout=*2)

ソケットから複数の 32 ビット整数を読み取ります。バイトはネットワーク バイト オーダーです。1 コマンドで最大 30 件の値を読み取ることができます。

たとえば、[3,100,2000,30000]を返します。タイムアウトが設定されているか応答が無効な場合、読み取った整数が 0 件であることを示す[0,-1,-1,-1]が返されます。

パラメータ

number：	読み取る変数の数（int）。
socket_name：	ソケットの名前（string）
timeout：	読み取りアクションがタイムアウトするまでの秒数（float）timeout が 0 または負数の場合、関数が読み取りを完了するまでリターンしてはいけないことを示します。

戻り値

読み取った数値のリスト（整数のリスト、長さ=number+1）

コマンドの例：
```
list_of_ints = socket_read_binary_integer(4,"socket_10")
```

●パラメータの例：

- number＝4 →読み取る整数の数
- socket_name＝socket_10

socket_read_byte_list(*number, socket_name=*'socket_0', *timeout*=2)

ソケットから複数のバイトを読み取ります。1 コマンドで最大 30 件の値を読み取ることができます。

たとえば、[3,100,200,44]を返します。タイムアウトが設定されているか応答が無効な場合、読み取ったバイトが 0 件であることを示す[0,-1,-1,-1]が返されます。

パラメータ

number：	読み取るバイトの数（int）。
socket_name：	ソケットの名前（string）
timeout：	読み取りアクションがタイムアウトするまでの秒数（float）timeout が 0 または負数の場合、関数が読み取りを完了するまでリターンしてはいけないことを示します。

戻り値

読み取った数値のリスト（整数のリスト、長さ=number+1）

コマンドの例：
```
list_of_bytes =
socket_read_byte_list(4,"socket_10")
```

●パラメータの例：

- number＝4 →読み取るバイト変数の数
- socket_name＝socket_10

socket_read_line(*socket_name*='socket_0', *timeout*=2)

*廃止予定：*最初の「\r\n」（キャリッジ リターンおよび改行）文字または「\n」（改行）文字までソケット バッファを読み取り、データを文字列として返します。返される文字列には「\n」文字も「\r\n」文字も含まれません。

たとえば、「サーバーからの応答」が返されます。タイムアウトするか応答が無効な場合、空の行が反されます（「」）。if ステートメントを使用して行が空かどうかをテストできます。

```
>>> if(line_from_server) :
>>>   popup("the line is not empty")
>>> end
```

パラメータ

socket_name：	ソケットの名前（string）
timeout：	読み取りアクションがタイムアウトするまでの秒数（float）timeout が 0 または負数の場合、関数が読み取りを完了するまでリターンしてはいけないことを示します。

戻り値

1 行の文字列

廃止予定：この関数は socket_read_string に置き換えられます。フラグ「interpret_escape」を「True」に設定すると、エスケープ シーケンス「\n」、「\r」、および「\t」のプレフィックスまたはサフィックスとしての使用が有効化されます。

コマンドの例：

```
line_from_server =
socket_read_line("socket_10")
```

●パラメータの例：

- socket_name＝socket_10

socket_read_string(*socket_name='socket_0', prefix='', suffix='', interpret_escape='False', timeout=2*)

ソケットからすべてのデータを読み取り、データを文字列として返します。

たとえば、「サーバーからの応答：\n Hello World」が返されます。タイムアウトするか応答が無効な場合、空の文字列が反されます（「」）。if ステートメントを使用して文字列が空かどうかをテストできます。

```
>>> if(string_from_server) :
>>>   popup("the string is not empty")
>>> end
```

任意のパラメータ「prefix」および「suffix」は、ソケットから抽出されるものを表現するのに使用できます。「prefix」は、ソケットから抽出された部分文字列（メッセージ）の先頭を示します。「prefix」の終わりまでのデータは無視され、ソケットから削除されます。「suffix」は、ソケットから抽出された部分文字列（メッセージ）の末尾を示します。「suffix」より後のソケットの残りのデータは保持されます。

「prefix」および「suffix」を使用すると、サフィックスによってメッセージの終わりが定義されるため、一度に複数の文字列をコントローラーに送信することもできます。例：「>hello<>world<」を送信し、このスクリプト関数を prefix=「>」および suffix=「<」として.呼び出します。

現在のソフトウェアではプレフィックスおよびサフィックス文字列の先頭のスペースは無視されますが、今後のリリースでは通信エラーを引き起こす可能性があります。

任意のパラメータ「interpret_escape」は、プレフィックスまたはサフィックスの一部としてエスケープ シーケンス「\n」、「\t」、「\r」の使用を許可するために使用できます。

パラメータ

socket_name：	ソケットの名前（string）
prefix：	プレフィックスを定義します（string）
suffix：	サフィックスを定義します（string）
interpret_escape：	エスケープ シーケンスの解釈を有効化します（bool）。
timeout：	読み取りアクションがタイムアウトするまでの秒数（float）timeout が 0 または負数の場合、関数が読み取りを完了するまでリターンしてはいけないことを示します。

戻り値

文字列

コマンドの例：

```
string_from_server = socket_read_string("socket_10",prefix=">",suffix="<")
```

socket_send_byte(*value, socket_name='*`socket_0`*'*)

サーバーにバイトを送信します。

ソケットを介してバイト<value>を送信します。期待されるレスポンスはありません。特殊 ASCII 文字を送信するのに使用できます。10 は改行、2 はテキストの先頭、3 はテキストの末尾です。

パラメータ

`value`： 送信する数値（byte）。

`socket_name`： ソケットの名前（string）

戻り値

送信操作が成功したかどうかを示すブール値

コマンドの例：`socket_send_byte(2,"socket_10")`

●パラメータの例：

- value＝2
- socket_name＝socket_10
  - ＊True または False（送信されたか送信されてないか）が返されます。

socket_send_int(*value, socket_name='*`socket_0`*'*)

サーバーに int（int32_t）を送信します。

ソケットを介して int<value>を送信します。ネットワーク バイト オーダーで送信します。期待されるレスポンスはありません。

パラメータ

`value`： 送信する数値（int）。

`socket_name`： ソケットの名前（string）

戻り値

送信操作が成功したかどうかを示すブール値

コマンドの例：`socket_send_int(2,"socket_10")`

●パラメータの例：

- value＝2
- socket_name＝socket_10
  - ＊True または False（送信されたか送信されてないか）が返されます。

socket_send_line(*str, socket_name='socket_0'*)

サーバーに改行文字を含む文字列を送信します—UR ダッシュボード サーバーとの通信に役立ちます。

ソケットを介して ASCII コードで文字列<str>を送信します。期待されるレスポンスはありません。

パラメータ

`str`： 送信する文字列（ascii）。

`socket_name`： ソケットの名前（string）

戻り値

送信操作が成功したかどうかを示すブール値

コマンドの例：`socket_send_line("hello","socket_10")`

socket_10 に hello\n を送信します。

●パラメータの例：

- str＝hello
- socket_name＝socket_10
  - ＊True または False（送信されたか送信されてないか）が返されます。

socket_send_string(*str, socket_name='socket_0'*)

サーバーに文字列を送信します。

ソケットを介して ASCII コードで文字列<str>を送信します。期待されるレスポンスはありません。

パラメータ

`str`： 送信する文字列（ascii）。

`socket_name`： ソケットの名前（string）

戻り値

送信操作が成功したかどうかを示すブール値

コマンドの例：`socket_send_string("hello","socket_10")`

socket_10 に hello を送信します。

●パラメータの例：

- str＝hello
- socket_name＝socket_10
  - ＊True または False（送信されたか送信されてないか）が返されます。

socket_set_var(*name, value, socket_name='socket_0'*)

サーバーに整数を送信します。

ソケットを介してメッセージ「SET <name> <value>\n」を送信します。期待されるレスポンスはありません。

パラメータ

`name`： 変数名（string）

`value`： 送信する数値（int）。

`socket_name`： ソケットの名前（string）

コマンドの例：`socket_set_var("POS_Y",2200,"socket_10")`

文字列を送信します：SET POS_Y 2200\n to socket_10

●パラメータの例：

- name＝POS_Y →変数の名前
- value＝2200
- socket_name＝socket_10

write_output_boolean_register(*address, value*)

フィールド バスもアクセスできる出力レジスタのいずれかにブール値を書き込みます。独自のメモリ空間が使用されることに注意してください。

```
>>> write_output_boolean_register(3, True)
```

パラメータ

`address`： レジスタのアドレス（0:63）

`value`： レジスタに設定する値（True、False）

コマンドの例：`write_output_boolean_register(3,True)`

●パラメータの例：

- address＝3
- value＝True

write_output_float_register(*address, value*)

フィールド バスもアクセスできる出力レジスタのいずれかに浮動小数点数値を書き込みます。独自のメモリ空間が使用されることに注意してください。

```
>>> write_output_float_register(3, 37.68)
```

パラメータ

`address`： レジスタのアドレス（0:23）

`value`： レジスタに設定する値（float）

コマンドの例：`write_output_float_register(3,37.68)`

●パラメータの例：

- address＝3
- value＝37.68

write_output_integer_register(*address, value*)

フィールド バスもアクセスできる出力レジスタのいずれかに整数値を書き込みます。独自のメモリ空間が使用されることに注意してください。

```
>>> write_output_integer_register(3, 12)
```

パラメータ

`address`： レジスタのアドレス（0:23）

`value`： レジスタに設定する値[-2,147,483,648 : 2,147,483,647]

コマンドの例：`write_output_integer_register(3,12)`

●パラメータの例：

- address＝3
- value＝12

write_port_bit(*address, value*)

Modbus クライアントもアクセスできるいずれかのポートを書き込みます。

```
>>> write_port_bit(3,True)
```

パラメータ

`address`： ポートのアドレス（サポート サイトのページ「UsingModbusServer」のポートマップを参照してください。）

`value`： レジスタに設定する値（True、False）

コマンドの例：`write_port_bit(3,True)`

●パラメータの例：

- address＝3
- value＝True

write_port_register(*address, value*)

Modbus クライアントもアクセスできるいずれかのポートを書き込みます。

```
>>> write_port_register(3,100)
```

パラメータ

`address`： ポートのアドレス（サポート サイトのページ「UsingModbusServer」のポートマップを参照してください。）

`value`： ポートに設定する値（0 : 65536）または（-32768 : 32767）

コマンドの例：`write_port_bit(3,100)`

●パラメータの例：

- address＝3
- value＝100

zero_ftsensor()

後続の測定値から現在の測定値を減算することで、組み込みの力／トルクセンサーからの TCP 力／トルク計測値をゼロにします。

注：この関数は G5 にのみ適用されます。

5.2 変数

名前	説明
__package__	**値**：`None`

6 IO 設定モジュール

6.1 関数

modbus_set_runstate_dependent_choice(*signal_name, runstate_choice*)

プログラムの状態（実行中または停止）に応じて、出力信号レベルを設定します。

```
>>> modbus_set_runstate_dependent_choice("output2",1)
```

パラメータ

signal_name：	事前に追加した出力デジタル信号を識別する文字列。
runstate_choice：	整数：0＝プログラム状態を保持、1＝プログラムが実行中ではない場合にローを設定、2＝プログラムが実行中ではない場合にハイを設定、3＝プログラムが実行中の場合はハイ、停止している場合はロー

コマンドの例：

```
modbus_set_runstate_dependent_choice("output2", 3)
```

●パラメータの例：

- 信号名＝output2
- 実行状態依存選択＝3 →プログラムが停止している場合はロー、プログラムが実行中の場合はハイを設定

set_analog_outputdomain(*port, domain*)

アナログ出力のドメインを設定します。

パラメータ

port：	アナログ出力ポート番号
domain：	アナログ出力ドメイン：0：*4～20mA*、1：*0～10V*

コマンドの例： `set_analog_outputdomain(1,1)`

●パラメータの例：

- port はアナログ出力ポート 1 です（コントローラー上）。
- domain＝1（0～10 ボルト）

set_configurable_digital_input_action(*port, action*)

このメソッドを使用して、選択した設定可能デジタル入力レジスタを「default」または「freedrive」アクションに設定します。

以下も参照してください：

- `set_input_actions_to_default`
- `set_standard_digital_input_action`
- `set_tool_digital_input_action`
- `set_gp_boolean_input_action`

パラメータ

`port`： 設定可能デジタル入力ポート番号。（integer）

`action`： アクションのタイプ。アクションは「default」または「freedrive」のいずれかとなります。（string）

コマンドの例：
```
set_configurable_digital_input_action(0, "freedrive")
```

●パラメータの例：

- n は設定可能デジタル入力レジスタ 0 です。
- f は「freedrive」アクションに設定されます。

set_gp_boolean_input_action(*port, action*)

このメソッドを使用して、選択した gp ブール入力レジスタを「default」または「freedrive」アクションに設定します。

以下も参照してください：

- `set_input_actions_to_default`
- `set_standard_digital_input_action`
- `set_configurable_digital_input_action`
- `set_tool_digital_input_action`

パラメータ

`port`： gp ブール入力ポート番号。（Integer）

`action`： アクションのタイプ。アクションは「default」または「freedrive」のいずれかとなります。（string）

コマンドの例：
```
set_gp_boolean_input_action(0, "freedrive")
```

●パラメータの例：

- n は gp ブール入力レジスタ 0 です。
- f は「freedrive」アクションに設定されます。

set_input_actions_to_default()

このメソッドを使用して、すべての標準、設定可能、ツール、gp_boolean 入力レジスタの入力アクションを「default」アクションに設定します。

以下も参照してください：

- set_standard_digital_input_action
- set_configurable_digital_input_action
- set_tool_digital_input_action
- set_gp_boolean_input_action

コマンドの例：`set_input_actions_to_default()`

set_runstate_configurable_digital_output_to_value(*outputId, state*)

プログラムの状態（実行中または停止）に応じて、出力信号レベルを設定します。

例：プログラムが実行中ではない場合、設定可能デジタル出力 5 をハイに設定します。

```
>>> set_runstate_configurable_digital_output_to_value(5, 2)
```

パラメータ

`outputId`：出力信号番号（id）、整数：[0:7]

`state`：出力の状態、整数：0＝状態保持、1＝プログラムが実行中ではない場合はロー、2＝プログラムが実行中ではない場合はハイ、3＝プログラムが実行中の場合はハイ、停止している場合はロー

コマンドの例：

```
set_runstate_configurable_digital_output_to_value(5, 2)
```

- パラメータの例：
  - outputid＝ポート 5 の設定可能デジタル出力
  - 実行状態選択＝2 →プログラムが実行中ではない場合はハイ

set_runstate_gp_boolean_output_to_value(*outputId, state*)

プログラムの状態（実行中または停止）に応じて、出力値を設定します。

例：プログラムが実行中ではない場合、汎用ビット出力 5 をハイに設定します。

```
>>> set_runstate_gp_bool_output_to_value(5, 2)
```

パラメータ

`outputId`：出力信号番号（id）、整数：[0:63]

`state`：出力の状態、整数：0＝状態保持、1＝プログラムが実行中ではない場合はロー、2＝プログラムが実行中ではない場合はハイ、3＝プログラムが実行中の場合はハイ、停止している場合はロー

コマンドの例：`set_runstate_gp_boolean_output_to_value(5, 2)`

●パラメータの例：

- outputid＝ポート 5 の出力
- 実行状態選択＝2→プログラムが実行中ではない場合はハイ

set_runstate_standard_analog_output_to_value(*outputId, state*)

プログラムの状態（実行中または停止）に応じて、出力信号レベルを設定します。

例：プログラムが実行中ではない場合、標準アナログ出力 1 をハイに設定します。

```
>>> set_runstate_standard_analog_output_to_value(1, 2)
```

パラメータ

`outputId`：出力信号番号（id）、整数：[0:1]

`state`：出力の状態、整数：0＝状態保持、1＝プログラムが実行中ではない場合は最小、2＝プログラムが実行中ではない場合は最大、3＝プログラムが実行中の場合は最大、停止している場合は最小

コマンドの例：

```
set_runstate_standard_analog_output_to_value(1, 2)
```

●パラメータの例：

- outputid＝ポート 1 の標準アナログ出力
- 実行状態選択＝2 →プログラムが実行中ではない場合はハイ

set_runstate_standard_digital_output_to_value(*outputId, state*)

プログラムの状態（実行中または停止）に応じて、出力信号レベルを設定します。

例：プログラムが実行中ではない場合、標準デジタル出力 5 をハイに設定します。

```
>>> set_runstate_standard_digital_output_to_value(5, 2)
```

パラメータ

`outputId`：出力信号番号（id）、整数：[0:7]

`state`：出力の状態、整数：0＝状態保持、1＝プログラムが実行中ではない場合はロー、2＝プログラムが実行中ではない場合はハイ、3＝プログラムが実行中の場合はハイ、停止している場合はロー

コマンドの例：

```
set_runstate_standard_digital_output_to_value(5, 2)
```

●パラメータの例：

- outputid＝ポート 1 の標準デジタル出力
- 実行状態選択＝2→プログラムが実行中ではない場合はハイ

set_runstate_tool_digital_output_to_value(*outputId, state*)

プログラムの状態（実行中または停止）に応じて、出力信号レベルを設定します。

例：プログラムが実行中ではない場合、ツール デジタル出力 1 をハイに設定します。

```
>>> set_runstate_tool_digital_output_to_value(1, 2)
```

パラメータ

`outputId`：出力信号番号（id）、整数：[0:1]

`state`：出力の状態、整数：0＝状態保持、1＝プログラムが実行中ではない場合はロー、2＝プログラムが実行中ではない場合はハイ、3＝プログラムが実行中の場合はハイ、停止している場合はロー

コマンドの例：

```
set_runstate_tool_digital_output_to_value(1, 2)
```

●パラメータの例：

- outputid＝ポート 1 のツール デジタル出力
- 実行状態選択＝2 →プログラムが実行中ではない場合はハイ

set_standard_analog_input_domain(*port, domain*)

コントローラー ボックスの標準アナログ入力のドメインを設定します。

ツール入力については、`set_tool_analog_input_domain` を参照してください。

パラメータ

`port`： アナログ入力ポート番号 0 または 1

`domain`： アナログ入力ドメイン：0：*4～20mA*、1：*0～10V*

コマンドの例：`set_standard_analog_input_domain(1,0)`

●パラメータの例：

- port＝アナログ入力ポート 1
- domain＝0（4～20mA）

set_standard_digital_input_action(*port, action*)

このメソッドを使用して、選択した標準デジタル入力レジスタを「default」または「freedrive」アクションに設定します。

以下も参照してください：

●`set_input_actions_to_default`
●`set_configurable_digital_input_action`
●`set_tool_digital_input_action`
●`set_gp_boolean_input_action`

パラメータ

`port`： 標準デジタル入力ポート番号。（Integer）

`action`： アクションのタイプ。アクションは「default」または「freedrive」のいずれかとなります。（string）

コマンドの例：`set_standard_digital_input_action(0, "freedrive")`

●パラメータの例：

- n は標準デジタル入力レジスタ 0 です。
- f は「freedrive」アクションに設定されます。

set_tool_analog_input_domain(*port, domain*)

ツールのアナログ入力のドメインを設定します。

コントローラー ボックス入力については、
`set_standard_analog_input_domain` を参照してください。

パラメータ

`port`： アナログ入力ポート番号 0 または 1

`domain`： アナログ入力ドメイン：0：*4～20mA*、1：*0～10V*

コマンドの例：`set_tool_analog_input_domain(1,1)`

●パラメータの例：

- port＝ツール アナログ入力 1
- domain＝1（0～10 ボルト）

set_tool_digital_input_action(*port, action*)

このメソッドを使用して、選択したツール デジタル入力レジスタを
「default」または「freedrive」アクションに設定します。

以下も参照してください：

●`set_input_actions_to_default`
●`set_standard_digital_input_action`
●`set_configurable_digital_input_action`
●`set_gp_boolean_input_action`

パラメータ

`port`： ツール デジタル入力ポート番号。（Integer）

`action`： アクションのタイプ。アクションは「default」または
「freedrive」のいずれかとなります。（string）

コマンドの例：`set_tool_digital_input_action(0, "freedrive")`

●パラメータの例：

- n はツール デジタル入力レジスタ 0 です。
- f は「freedrive」アクションに設定されます。

6.2 変数

名前	説明
__package__	**値：**`None`

インデックス