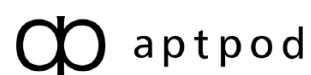


intdash Edge Agent 用 デバイスコネクター デベロッパーガイド

Device Connector Framework v2.1.0

第 2 版 (2023 年 7 月)



目次

01 はじめに	4
02 デバイスコネクタ開発の概要	5
2.1 デバイスコネクタとは.....	5
2.2 Device Connector Framework とは	5
2.3 Device Connector Framework で使用される用語	6
03 デバイスコネクタをビルドし、動かしてみる	9
3.1 開発環境を準備する	9
3.2 基本エレメントのみのデバイスコネクタをビルドする	9
3.3 パイプライン設定ファイルを作成する	9
3.4 デバイスコネクタを実行する	10
04 パイプライン設定ファイル	11
4.1 パイプライン設定ファイルの基本	11
4.2 runner の設定	11
4.3 bg_processes の設定	12
4.4 before_task の設定	12
4.5 after_task の設定	12
4.6 plugin の設定	12
4.7 tasks の設定	12
05 Device Connector Framework に含まれる基本エレメント	14
5.1 FileSinkElement (file-sink).....	14
5.2 FileSrcElement (file-src).....	14
5.3 NullSinkElement (null-sink)	15
5.4 ProcessSrcElement (process-src)	15
5.5 StatFilterElement (stat-filter).....	16
5.6 StdoutSinkElement (stdout-sink)	16
5.7 TextSrcElement (text-src).....	17
06 独自エレメントの開発の概要	18
6.1 プラグインにしてデバイスコネクタ実行時にロードする (Rust または C)	18
6.2 デバイスコネクタの実行ファイルに含める (Rust)	19
07 Rust による独自エレメントの開発	20
7.1 テンプレートを利用した準備.....	20
7.2 ElementBuildable トrait	20
7.3 HelloSrcElement を作る	21
7.4 エレメントをプラグインにする／実行ファイルに含める	24

7.5	ビルド	25
7.6	(補足) sink エLEMENTの例	26
08	C による独自ELEMENTの開発	28
8.1	開発用のファイルの準備	28
8.2	ELEMENTの実装	28
8.3	コンパイル	32
09	Device Connector Framework の既知の制約	33
10	device-connector-intdash とは	34
10.1	インストール	34
10.2	パッケージ情報	35
11	device-connector-intdash の機能	37
11.1	パイプライン設定ファイルの環境変数	37
11.2	パイプライン設定ファイルのサンプル	37
12	device-connector-intdash に含まれるELEMENT	38
12.1	apt-analogtrx-src	39
12.2	apt-cantrx-src	40
12.3	nmea-packet-src	42
12.4	ubx-src	42
12.5	v4l2-src	44
12.6	h264-split-filter	45
12.7	iscp-v2-compatible-filter	46
12.8	jpeg-split-filter	54
12.9	pcm-split-filter	54
12.10	print-log-filter	57
12.11	ubx-iscpv2-filter	58
12.12	apt-cantrx-sink	59

01 はじめに

重要:

- このドキュメントに記載されている仕様は予告なく変更される場合があります。このドキュメントは情報提供を目的としたものであり、仕様を保証するものではありません。
- 説明で使用している画面は一例です。ご使用の環境やアプリケーションのバージョンによって、表示や手順が一部異なる場合があります。

注釈: このドキュメントに記載されている会社名、サービス名、製品名等は、一般に、各社の登録商標または商標です。本文および図表中には、「™」、「®」は明記していません。

本ドキュメントは、intdash Edge Agent 用デバイスコネクタを使用・開発する方のために、以下を説明するものです。

- Device Connector Framework を使って開発されたデバイスコネクタの設定方法（エレメントを使ってパイプラインを組む方法など）
- Device Connector Framework を使って独自デバイスコネクタを開発する方法
- アプトポッド製のデバイスコネクタ (device-connector-intdash) の使用方法

02 デバイスコネクタ開発の概要

2.1 デバイスコネクタとは

デバイスコネクタは、intdash エッジにおいて、intdash Edge Agent と外部デバイスとの間でデータを仲介するソフトウェアコンポーネントです。

デバイスコネクタは intdash Edge Agent と同じエッジデバイスにインストールされ、以下のような機能を担います。

- センサーなどの外部デバイスからデータを受け取り、intdash Edge Agent に渡す。
- 逆に、intdash Edge Agent からデータを受け取り、ロボットなどの外部デバイスに渡す。

デバイスコネクタと intdash Edge Agent の間のデータの受け渡しには、FIFO（名前付きパイプ）を使います。

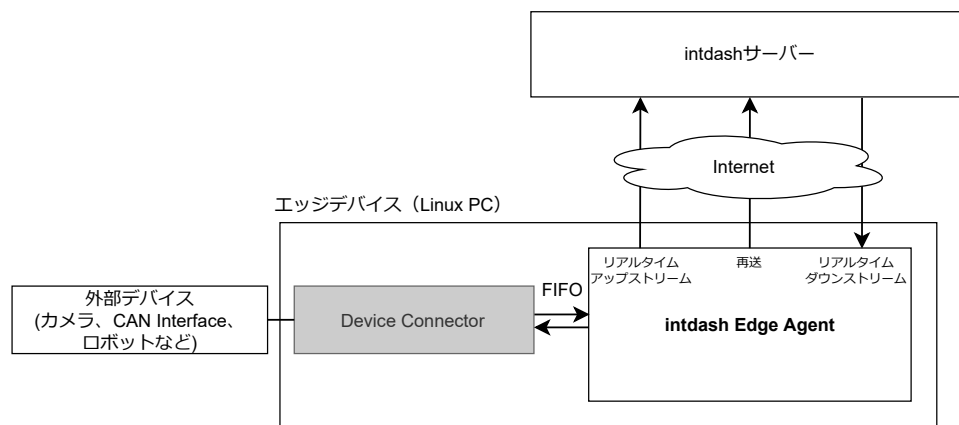


図1 デバイスコネクタ

2.2 Device Connector Framework とは

Device Connector Framework は、デバイスコネクタを開発する方のための開発用フレームワークです。

- Device Connector Framework により作成されたデバイスコネクタでは、デバイスへの接続やデータ収集、収集したデータに対する処理は、それぞれ「エレメント (Element)」として定義されます。ユーザーは、エレメントを組み合わせ、データが流れるパイプラインを設定することができます。設定はパイプライン設定ファイルで行います。
- デバイスコネクタを実行すると、パイプライン設定ファイルで指定された各エレメントは、同期または非同期に動作する「タスク (Task)」として実行されます。また、タスク間でのメッセージパッシングはデバイスコネクタにより管理されます。

これにより、柔軟にエレメントを組み合わせ、データ（メッセージ）が流れるパイプラインを作成することができます。

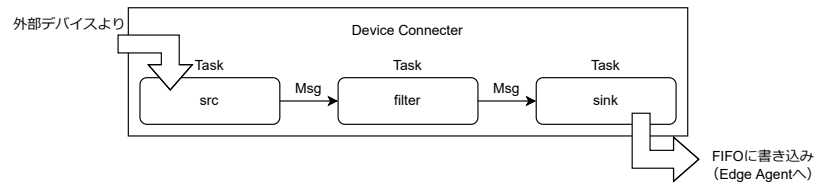


図 2 デバイスから intdash Edge Agent へ入力するパイプラインの例

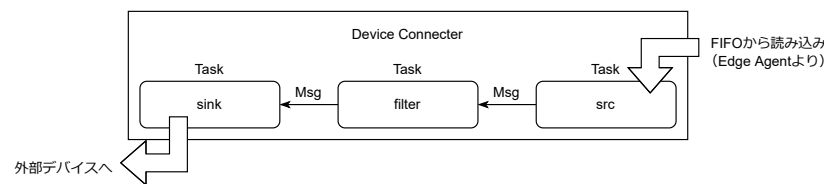


図 3 intdash Edge Agent からデバイスへ出力するパイプラインの例

なお、ファイルからの読み込み、ファイルへの書き出しなどの基本的な処理を行うエレメントは Device connector Framework に含まれています（基本エレメント）。

2.3 Device Connector Framework で使用される用語

Device Connector Framework で使用される用語について説明します。

2.3.1 メッセージ

タスク間で送受信されるデータです。

タスク間で送受信されるメッセージは、何らかの型を持ちます。型情報は、MIME タイプ（例：MIME タイプ "image/jpeg"）またはカスタムタイプ（例：カスタムタイプ "iscp-v2-compatible-msg"）です。

各ポートがどの型のメッセージを受け取れるかは、エレメントにおいて定義されます。

送信元のタスクは、メッセージを最初に送信するとき、そのメッセージの型をデバイスコネクタに通知します。デバイスコネクタは、メッセージを受信するポートがその型を受け入れ可能かどうかを判定し、受け入れができない場合はエラーとします。

データを、MIME タイプ "image/jpeg"、MIME タイプ "text/plain"、カスタムタイプ "iscp-v2-compatible-msg" 等で送出するエレメントの場合は、1 枚の画像、1 行のテキスト、1 つのデータポイントのように、意味を持つ単位を 1 つのメッセージとして送出します。

それに対し、MIME タイプ "application/octet-stream" で送出するエレメントの場合は、送出するデータは区切りのないバイナリストリームであるため、意味的な区切りとは関係なく送信上の都合でセグメントに分割して送信します。この場合のセグメントもここではメッセージと呼びます。

2.3.2 エレメント

メッセージの生成やメッセージの処理を定義したものです。デバイスコネクタ内にメッセージを生成する「src エレメント」、受け取るだけの「sink エレメント」、送受信を共に行う「filter エレメント」があります。

エレメントは、実行時には「タスク」として実行されます。

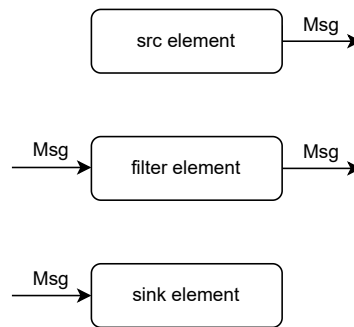


図 4 エレメントの種類 (src、filter、sink)

デバイスコネクタ内のパイプラインでは、以下の順序で処理が行われます。

- src エレメントにおいて、デバイスコネクタの外部（例えばセンサー）からデータを取得して、メッセージを生成
- filter エレメントにおいて、メッセージを意味のある単位に区切る、変換する、などの処理を行う
- sink エレメントにおいて、デバイスコネクタの外部（例えば intdash Edge Agent）にデータを出力する

注釈: アプトポッドにより提供されるエレメントは以下のような名前になっています。

- src エレメント: *-src
- filter エレメント: *-filter
- sink エレメント: *-sink

例えば、入力を jpeg のフレームごとのメッセージにするのは filter エレメントの役目で、jpeg-split-filter と名付けています。

2.3.3 タスク

エレメントにおいて定義された処理が実際に実行されるとき、その処理はタスクと呼ばれます。タスクは、メッセージの送受信が必要になったときに起動されます。

2.3.4 ポート

エレメントには、0 個以上 255 個以下の受信ポートと、0 個以上 255 個以下の送信ポートを定義することができます。複数のポートを使うことによって、ポートごとに別の型のメッセージを送受信させることができます。

- 複数の送信ポートからのメッセージを、1 つの受信ポートで受信することができます。
- 1 つの送信ポートから、複数の受信ポートに向けてメッセージを送信することはできません。

注釈: Device Connector Framework v1.0 現在、送受信ともに 0 番ポート 1 つのみを使用することができます。今後のリリースで、順次複数のポートに対応する予定です。

03 デバイスコネクタをビルドし、動かしてみる

この章では、Device Connector Framework を使用してデバイスコネクタをビルドし、動作確認します。ここでビルドされるデバイスコネクタには、基本エレメントが含まれます。基本エレメントについては、[Device Connector Framework に含まれる基本エレメント](#) (p. 14) を参照してください。

3.1 開発環境を準備する

Device Connector Framework を使ってデバイスコネクタをビルドするには、Rust の開発環境が必要です。Rust の開発環境をインストールしてください。

3.2 基本エレメントのみのデバイスコネクタをビルドする

Device Connector Framework には基本エレメントのソースコードが付属しています。

以下のコマンドを実行することにより、Device Connector Framework を取得し、基本エレメントを内部に含んだデバイスコネクタをビルドすることができます。

```
git clone https://github.com/aptpod/device-connector-framework.git
cd device-connector-framework
cargo build --release
```

生成された、./target/release/device-connector-run という実行ファイルが、基本エレメントを含むデバイスコネクタです。

3.3 パイプライン設定ファイルを作成する

Device Connector Framework により作成されたデバイスコネクタを使用するには、行いたい処理をパイプラインとして定義する必要があります。

例として、以下のパイプライン設定ファイル `conf.yml` をカレントディレクトリに用意します。ここでは、文字列を 100ms ごとに送出するエレメント (text-src) に、受信内容を標準出力に書き出すエレメント (stdout-sink) をつないで、パイプラインを作成しています。

```
tasks:
- id: 1
  element: text-src
  conf:
    text: "Hello, World!"
    interval_ms: 100

- id: 2
  element: stdout-sink
  from: [[ 1 ]]
  conf:
    separator: "\n"
```

3.4 デバイスコネクタを実行する

以下のコマンドでデバイスコネクタを実行します。

```
./target/release/device-connector-run --config conf.yml
```

すると、パイプライン設定ファイルで指定した文字列が 100ms ごとに出力されます。

```
Hello, World!  
Hello, World!  
Hello, World!  
Hello, World!  
Hello, World!
```

終了するには Ctrl-C を押してください。

以上で、デバイスコネクタの動作確認は終了です。

この例では文字列を標準出力に出力しましたが、代わりに、intdash Edge Agent 用のバイナリフォーマットで FIFO に出力し、intdash Edge Agent がその FIFO から読み取るようにすることでデータを渡すことができます。

04 パイプライン設定ファイル

4.1 パイプライン設定ファイルの基本

パイプライン設定ファイルは、デバイスコネクタで行いたい処理をエレメントの組み合わせで定義したものです。

以下の設定例では、text-src エレメントにより定義されたタスクから 100ms ごとに "Hello, World!" という文字列を送出し、stdout-sink エレメントにより定義されたタスクがそれを受け取って標準出力に出力します。

また、プラグインファイル libdc_my_plugin0.so をロードするように設定しています。

```
runner:
  channel_capacity: 16

plugin:
  plugin_files:
    - /path/to/my/plugin0/libdc_my_plugin0.so

tasks:
  - id: 1
    element: text-src
    conf:
      text: "Hello, World!"
      interval_ms: 100

  - id: 2
    element: stdout-sink
    from: [[ 1 ]]
    conf:
      separator: "\n"
```

設定項目の詳細は以下のとおりです。

4.2 runner の設定

パイプライン設定ファイルの runner: には、デバイスコネクタの実行パラメータを記述します。

項目	設定値	説明
channel_capacity	整数値	スレッド間のメッセージ送信に用いるチャンネルの容量（メッセージの個数）です。これを超える容量のメッセージを送信しようとすると、受信側でメッセージが処理されるまで送信側のスレッドは処理がブロックされます。省略した場合デフォルトの値が使用されます。

4.3 bg_processes の設定

```
bg_processes:  
- command: /background/process/path args
```

バックグラウンドで実行するコマンドを指定します。タスクの開始時に before_task の前に起動されます。

4.4 before_task の設定

```
before_task:  
- command1  
- command2
```

タスクの開始前に実行されるコマンドを指定します。bg_processes と異なり、起動されたプロセスが終了するまでタスクの開始はブロックされます。

4.5 after_task の設定

```
after_task:  
- command1  
- command2
```

デバイスコネクタの終了時、タスクの終了後に実行されるコマンドを指定します。

4.6 plugin の設定

パイプライン設定ファイルの plugin: には、実行時にロードするプラグインについて設定します。

独自エレメントをプラグイン形式で作成した場合、ここでそのファイルを指定します。独自エレメントについては、[独自エレメントの開発の概要](#) (p. 18) を参照してください。

項目	設定値	説明
plugin_files	文字列の配列	ロードしたいプラグインファイル (例:libdc_XXX.so) へのパスのリストです。

4.7 tasks の設定

タスクの設定は、tasks: 内に並列に記述します。

項目	設定値	説明
id	整数値	タスクに割り当てる ID です。これはこのパイプライン上で一意でなくてはなりません。
element	文字列	このタスクに割り当てるエレメントを指定します。
from	文字列の二次元配列	<p>このタスクが受け取るメッセージの送信元を指定します。</p> <p>[["<送信元タスク ID>:<送信元ポート番号>", ...]</p> <p>メッセージは非同期に送られます。</p> <p>例えば、[["1:0"], ["2:1"]] は、以下を意味します。</p> <ul style="list-style-type: none">• 0 番ポートが「タスク ID1 の 0 番ポート」からメッセージを受け取る• 1 番ポートが「タスク ID2 の 1 番ポート」からメッセージを受け取る <p>ポート番号が 0 の場合、ポート番号を省略して "1:0" を "1" のように記述することができます。したがって、[["1:0"], ["2:1"]] は [["1"], ["2:1"]] と同義です。</p>

重要: from に設定する送信元のタスク ID は、自身のタスク ID より小さい数字である必要があります。

エレメントによっては、独自の設定項目も存在します。エレメント独自の設定項目は、conf: に記述してください。

05 Device Connector Framework に含まれる基本エレメント

Device Connector Framework には、パイプラインを作成するための基本的なエレメントが付属しています。

5.1 FileSinkElement (file-sink)

ドキュメント

指定されたパスのファイルに、受け取ったメッセージを書き出します。

エレメントの種類

sink

前のエレメントから受信するポートの数

1

受信ポートでの受信形式

任意（受け取ったデータをそのままファイルに書き出します）

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
path	文字列	出力先のファイルパス
create	true/false	true の場合、出力先のファイルがない場合は作成する。デフォルトは false
separator	文字列	メッセージ出力ごとに挿入するテキスト。デフォルトは空
flush_size	整数	入力をフラッシュするまでにバッファにためるデータ量 (バイト)。デフォルトは 0

5.2 FileSrcElement (file-src)

ドキュメント

指定されたパスのファイルを読み込み、内容をメッセージとして送出します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
path	文字列	読み取るファイルのパス

5.3 NullSinkElement (null-sink)

ドキュメント

メッセージを受け取りますが、それを何にも使わずに破棄します。

エレメントの種類

sink

前のエレメントから受信するポートの数

1

受信ポートでの受信形式

任意

エレメント独自の設定項目はありません。

5.4 ProcessSrcElement (process-src)

ドキュメント

プロセスを起動し、起動したプロセスの標準出力をバッファサイズに収まるように区切って送出します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
program	文字列	実行ファイル
args	文字列の配列	実行ファイルに渡す引数
command	文字列	実行ファイルと引数を 1 文字列で指定したもの。program と args が指定されなかった場合に使用。
buffer_size	整数	バッファサイズ (バイト)。デフォルトは 255。

5.5 StatFilterElement (stat-filter)

[ドキュメント](#)

データを受け取った回数やサイズを確認するためのエレメントです。メッセージを受け取り、それをそのまま送ります。受け取ったメッセージの回数とサイズを記録し、その統計情報を標準エラー出力に書き出します。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

任意

送信ポートでの送信形式

受信ポートに入力されたデータをそのまま出力

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
interval_ms	整数	出力頻度をミリ秒にて指定

5.6 StdoutSinkElement (stdout-sink)

[ドキュメント](#)

受け取ったメッセージを標準出力に書き出します。

エレメントの種類

sink

前のエレメントから受信するポートの数

1

受信ポートでの受信形式

任意

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
separator	文字列	メッセージ出力ごとに挿入するテキスト。デフォルトは空

5.7 TextSrcElement (text-src)

ドキュメント

指定されたテキストを送出します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
text	文字列	メッセージのテキスト
interval_ms	整数値	メッセージ生成の間隔をミリ秒にて指定
repeat	整数値	「メッセージ送出のタイミングごとに、メッセージを何回送出するか」を指定します。デフォルトは 1 です例えば interval_ms: 100 かつ repeat: 5 の場合、100 ミリ秒経過ごとに 5 回メッセージを送出します。

06 独自エレメントの開発の概要

Device Connector Framework では、独自エレメントを開発し、パイプラインに組み込むことができます。ここでは、新たにエレメントを開発する方法を解説します。

独自エレメントを使用する方法には、以下の 2 つがあります。

- [プラグインにしてデバイスコネクタ実行時にロードする \(Rust または C\) \(p. 18\)](#)
- [デバイスコネクタの実行ファイルに含める \(Rust\) \(p. 19\)](#)

独自エレメントを Rust 言語で開発する場合は、どちらの方法も使用可能です。

独自エレメントを C 言語で開発する場合は、プラグインにする方法のみ使用可能です。

6.1 プラグインにしてデバイスコネクタ実行時にロードする (Rust または C)

Device Connector Framework で作成されたエレメントは、プラグインの形式 (共有ライブラリファイル、.so 形式) にビルドすることができます。1 つのプラグインには、複数のエレメントを定義することが可能です。

ビルドされたプラグインを使用するには、パイプライン設定ファイルの `plugin.plugin_files` に、プラグインファイルのパスの配列を指定します。

```
runner:
  ..

plugin:
  plugin_files:
    - /path/to/my/plugin0/libdc_my_plugin0.so
    - /path/to/my/plugin1/libdc_my_plugin1.so

tasks:
  - id: 1
    element: my-plugin0-foo-element # defined in loaded plugin
  ..
```

独自エレメントのプラグインは、Rust または C で開発します。詳細は以下を参照してください。

- [Rust による独自エレメントの開発 \(p. 20\)](#)
- [C による独自エレメントの開発 \(p. 28\)](#)

6.2 デバイスコネクタの実行ファイルに含める (Rust)

Device Connector Framework で作成されたエレメントは、デバイスコネクタの実行ファイルに含めてビルドすることも可能です。

必要なエレメントがデバイスコネクタの実行ファイルに含まれていればプラグインは必要ないため、パイプライン設定ファイルに `plugin` の記述は不要になります。

ただしこの場合、独自エレメントを追加・修正するたびにデバイスコネクタの実行ファイルを再ビルドする必要があります。

独自のエレメントをデバイスコネクタ実行ファイルに含めるには、エレメントを Rust で開発する必要があります。エレメントの開発方法はプラグインの場合と同様です。詳細は以下を参照してください。

- [Rust による独自エレメントの開発](#) (p. 20)

07 Rust による独自エレメントの開発

Device Connector Framework では、Rust で独自エレメントを開発することができます。

注釈: エレメントをプラグインにする場合も、デバイスコネクター実行ファイルに含める場合も、この後の [エレメントをプラグインにする／実行ファイルに含める](#) (p. 24) の前までは同じ手順です。

7.1 テンプレートを利用した準備

エレメントの開発のために、cargo-generate 用のテンプレートが用意されています。

注釈: [cargo-generate](#) は、既存の Rust プロジェクトをテンプレートにして新しいプロジェクトを作成するためのツールです。インストールしていない場合、次のコマンドでインストールしてください。

```
cargo install cargo-generate
```

次のコマンドでデバイスコネクターの新しいプロジェクトを作成します。新しいプロジェクトの名前の入力を求められるので、任意の名前を入力してください。

```
cargo generate --git https://github.com/aptpod/device-connector-template.git
```

これで新しいプロジェクトが作成されます。

このプロジェクトには、以下の 2 つのエレメントのソースコードがあらかじめ含まれています。

- src/hello.rs (hello-src)
- src/hexdump.rs (hexdump-sink)

7.2 ElementBuildable トレイト

Rust では、型に何らかの属性や実装を付与するためにトレイトと呼ばれる機能が提供されています。独自エレメントを定義するには、そのエレメントにおいて ElementBuildable トレイトを実装する必要があります。

ElementBuildable トレイトの定義は以下のとおりです。

```
pub trait ElementBuildable: Sized + Send + 'static {
    type Config: DeserializeOwned;
    const NAME: &'static str;
    const RECV_PORTS: Port = 0;
    const SEND_PORTS: Port = 0;
    fn acceptable_msg_types() -> Vec<Vec<MsgType>> {
        Vec::new()
    }
    fn new(conf: Self::Config) -> Result<Self, Error>;
    fn next(&mut self, pipeline: &mut Pipeline, receiver: &mut MsgReceiver) -> ElementResult;
    fn finalizer(&mut self) -> Result<Option<ElementFinalizer>, crate::error::Error>;
}
```

(次のページに続く)

(前のページからの続き)

}

Config

このエレメントを構築するのに必要な設定項目を指定します。デシリアライズ可能なデータ型が指定できますが、一般的には `#[derive(Debug, Deserialize)]` を指定した構造体にします。設定を受け取らない場合、`device_connector::EmptyElementConf` を指定します。

NAME エレメントの名前を指定します。他のエレメントとの重複は許されません。他のエレメントの名前と重複していると、実行時にエラーになります。

RECV_PORTS

メッセージを受け取るポートの数です。src エレメントの場合は 0 を指定します。filter または sink エレメントの場合は 1 以上を指定します。デフォルトは 0 です。

SEND_PORTS

メッセージを送出するポートの数です。sink エレメントの場合は 0 を指定します。src または filter エレメントの場合は 1 以上を指定します。デフォルトは 0 です。

acceptable_msg_types()

受け取ることのできるメッセージの型情報の配列を返す関数です。メッセージを受け取らない src エレメントの場合は実装する必要はありません。

new()

Config で指定した設定情報を受け取り、エレメントを実際に構築して返すメソッドです。

next()

データを受け取るための pipeline を受け取り、エレメントの実行結果を返します。next() は、MsgBuf に有効なデータを書き出すか終了するまで、処理を返さないように実装します。

finalizer()

プロセス終了時に実行されるクロージャを返すメソッドです。このクロージャではエレメント固有の終了処理を定義します。このメソッドを実装しない場合、終了処理を行わない挙動（デフォルト）となります。

7.3 HelloSrcElement を作る

実例として、device-connector-template に含まれているサンプルのエレメント HelloSrcElement を見ていきます。

このエレメントは、一定時間ごとにテキストを送信します。

```
use device_connector::{error::Error, ElementBuildable, ElementResult, MsgType, Pipeline, Port};
use serde_derive::Deserialize;
use std::io::Write;
use std::thread::sleep;
use std::time::Duration;

// ElementBuildable を実装するための対象となる型
pub struct HelloSrcElement {
    conf: HelloSrcElementConf,
```

(次のページに続く)

(前のページからの続き)

```
}

// HelloSrcElement が受け取る設定の定義
#[derive(Debug, Deserialize)]
#[serde(deny_unknown_fields)]
pub struct HelloSrcElementConf {
    text: String,
}

// ElementBuildable の実装
impl ElementBuildable for HelloSrcElement {
    type Config = HelloSrcElementConf;

    const NAME: &'static str = "hello-src";

    const SEND_PORTS: Port = 1;

    fn new(conf: Self::Config) -> Result<Self, Error> {
        Ok(HelloSrcElement { conf })
    }

    fn next(&mut self, pipeline: &mut Pipeline, _receiver: &mut MsgReceiver) -> ElementResult {
        pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap());

        sleep(Duration::from_millis(100));

        let mut buf = pipeline.msg_buf(0);
        buf.write_all(self.conf.text.as_bytes());

        Ok(ElementValue::MsgBuf)
    }
}
```

まず、本体である HelloSrcElement を定義します。

```
pub struct HelloSrcElement {
    conf: HelloSrcElementConf,
}
```

エレメントを実装する場合、実行に必要なデータを構造体のメンバとして持ちます。ここでは、送りたいテキストを含む HelloSrcElementConf をメンバとして持ちます。

HelloSrcElementConf は、エレメントを構築・実行するための設定を持ちます。

```
#[derive(Debug, Deserialize)]
#[serde(deny_unknown_fields)]
pub struct HelloSrcElementConf {
    text: String,
}
```

構造体の宣言に付随する各属性は、パイプライン設定ファイルからのデシリアライズを可能にするためのものです。基本的に、この例の通りの属性を付与すれば問題ありません。

エレメントを構築可能にするために、ElementBuildable を実装します。

```
impl ElementBuildable for HelloSrcElement {
    type Config = HelloSrcElementConf;

    const NAME: &'static str = "hello-src";

    const SEND_PORTS: Port = 1;

    fn new(conf: Self::Config) -> Result<Self, Error> {
        Ok(HelloSrcElement { conf })
    }

    ..
}
```

Config にはさきほど定義した HelloSrcElementConf を、NAME にはエレメントの名前である "hello-src" を指定します。この名前はパイプライン設定ファイルにおいてエレメントを指定する時に使われるもので、他のエレメントと重複しないものを選ぶ必要があります。

このエレメントは 1 つの送信ポートを持つので、SEND_PORTS に 1 を指定します。

new() には、受け取った設定 (HelloSrcElementConf) を元に、HelloSrcElement を生成するためのコードを記述します。

new() は HelloSrcElementConf 型の値 conf (パイプライン設定ファイルからパースされた設定) を受け取り、その設定に基づいて HelloSrcElement を作成して返却します。

filter エレメントまたは sink エレメントを実装する場合、どの型のメッセージを受け取れるかを示すために acceptable_msg_types() を実装しなければなりません。実装しない場合、どのようなデータも受け取れないエレメントとなります。

最後に、HelloSrcElement がメッセージを生成するためのメソッド next を定義します。

```
impl ElementBuildable for HelloSrcElement {
    ..

    fn next(&mut self, pipeline: &mut Pipeline, _receiver: &mut MsgReceiver) -> ElementResult {
        pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap());

        sleep(Duration::from_millis(100));

        let mut buf = pipeline.msg_buf(0);
        buf.write_all(self.conf.text.as_bytes());

        Ok(ElementValue::MsgBuf)
    }
}
```

src エレメントとして実装するために、next() メソッドの内部を記述していきます。

```
pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap());
```

送出メッセージの型情報を pipeline.check_send_msg_type に渡し、受信側で受け取りができるのかを判定します。このエレメントはテキストを送出するので、“text/plain” という MIME 情報を渡します。

```
sleep(Duration::from_millis(100));
```

データ量を適当な量に制限するため、ここでは 100ms の間 sleep します。

```
let mut buf = pipeline.msg_buf(0);
buf.write_all(self.conf.text.as_bytes());

Ok(ElementValue::MsgBuf)
```

設定で与えられたテキストをメッセージにするために、まずは Pipeline::msg_buf() を呼び出して、MsgBuf 型のバッファを用意します。

Pipeline::msg_buf() では、メッセージの送信に用いるポート番号を引数にします（大抵の場合は 0 を指定します）。

MsgBuf は std::io::Write を実装しているため、write_all() を用いてバッファにテキストを書き込みます。

関数の最後の Ok(ElementValue::MsgBuf) は、MsgBuf に書き込んだデータが送信先のタスクに送信されるよう指定するためのものです。

返却されたメッセージは、エレメントが独立したスレッドで動作している場合、関連付けられたタスクへチャンネルを用いて送信されます。エレメントが同期的に呼び出されている場合、呼び出し元のタスクの実行へ戻ります。

7.4 エレメントをプラグインにする／実行ファイルに含める

Rust で実装したエレメントを動作可能にする方法は、プラグインにする場合と、デバイスコネクタの実行ファイルに含める場合とで異なります。

7.4.1 プラグインにする場合

実装したエレメントをプラグインにする場合、define_dc_load!() マクロを使用します。例として、テンプレートの src/lib.rs を見てみます。

```
mod hello;

// Implement plugin interface.
device_connector::define_dc_load!(hello::HelloSrcElement);
```

ここでは、さきほど実装した HelloSrcElement をプラグインに登録しています。define_dc_load!() マクロには、複数のエレメントを渡すことも可能です。プラグインを開発したい場合、このような記述を lib.rs に記述します。

7.4.2 デバイスコネクタの実行ファイルに含める場合

Rust で実装したエレメントをデバイスコネクタの実行ファイルに含めるには、ElementBank に登録します。

ElementBuildable を実装した型を append_from_buildable() により登録し、その後に Runner を構築することで、エレメントが利用できるようになります。

例として、テンプレートの src/main.rs は以下のようになっています。

```
fn main() -> Result<(), ()> {  
    // ログシステムの初期化  
    env_logger::init();  
  
    // パイプライン設定ファイルの読み込み  
    let opts: Opts = Opts::parse();  
    let conf = Conf::read_from_file(&opts.config)?;  
  
    // ElementBank の作成  
    let mut bank = ElementBank::new();  
  
    // プラグインのロード  
    let loaded_plugin = LoadedPlugin::from_conf(&conf.plugin)?;  
  
    // 実装した HelloSrcElement の登録  
    bank.append_from_buildable::(<hello::HelloSrcElement>());  
  
    // runner の作成  
    let mut runner_builder = RunnerBuilder::new(&bank, &loaded_plugin, &conf.runner);  
    runner_builder.append_from_conf(&conf.tasks)?;  
    let runner = runner_builder.build()?;  
  
    // 起動  
    runner.run()?;  
  
    Ok(())  
}
```

これは通常の Rust の main 関数であるため、ここに追記することでデバイスコネクタが動作する前後の挙動をカスタマイズすることも可能です。

7.5 ビルド

テンプレートを使用した場合、実装した拡張は以下のコマンドでビルドできます。

```
cargo build --release
```

これにより、target/release に、以下の 2 種類の成果物が生成されます。いずれかを使用してください。

- libdc_XXXXXX.so : HelloSrcElement を含むプラグインファイル（共有ライブラリ）。デバイスコネクタ実行時にこのプラグインをロードするように指定すれば、HelloSrcElement を使用できます。プラグイン

のロード方法については [plugin の設定](#) (p. 12) を参照してください。

- xxxxxx-run : HelloSrcElement を含むデバイスコネクタの実行ファイル。このデバイスコネクタを実行すれば、プラグインなしで HelloSrcElement を使用できます。

注釈: テンプレートで作成したプロジェクトにある Dockerfile を使用してクロスビルドすることも可能です。詳細については、プロジェクトにある README.md を参照してください。

注意: アプトポッド製のデバイスコネクタ ([device-connector-intdash](#)) (p. 34) を利用してプラグイン開発を行う場合、[Device Connector Framework の既知の制約](#) (p. 33) により、Rust のバージョンを本体とプラグインで合わせる必要があります。

プラグインを開発する場合、[パッケージ情報](#) (p. 35) を参照し、device-connector-intdash パッケージに合った Rust バージョンでプラグインをビルドしてください。

Dockerfile でクロスビルドする場合、Rust バージョンを RUST_TOOLCHAIN で設定することが可能です。RUST_TOOLCHAIN のデフォルト設定値は、最新の device-connector-intdash のバージョンに合わせて更新されるため、最新バージョンを利用する場合はデフォルト設定のままで構いません。

7.6 (補足) sink エレメントの例

上の説明では例として src エレメントを作成しましたが、sink エレメントを作成する場合、例えば以下のようになります。

このエレメント (hexdump-sink) では、受け取った各メッセージを標準エラーに出力します。

注釈: このエレメント (hexdump-sink) は [device-connector-template](#) に含まれます。

```
use device_connector::EmptyElementConf;
use device_connector::{
    ElementBuildable, ElementResult, Error, MsgReceiver, MsgType, Pipeline, Port,
};

pub struct HexdumpSinkElement {}

impl ElementBuildable for HexdumpSinkElement {
    type Config = EmptyElementConf;

    const NAME: &'static str = "hexdump-sink";
    const RECV_PORTS: Port = 1;
    const SEND_PORTS: Port = 0;

    fn acceptable_msg_types() -> Vec<Vec<MsgType>> {
        vec![vec![MsgType::any()]]
    }

    fn new(_conf: Self::Config) -> Result<Self, Error> {
```

(次のページに続く)

(前のページからの続き)

```
Ok(Self {})  
}  
  
fn next(&mut self, _pipeline: &mut Pipeline, receiver: &mut MsgReceiver) -> ElementResult {  
    loop {  
        let msg = receiver.recv(0)?;  
        let bytes = msg.as_bytes();  
        eprintln!(  
            "msg ={}",  
            bytes  
                .iter()  
                .map(|x| format!("{:02X}", x))  
                .collect::<String>()  
        );  
    }  
}  
}
```

08 C による独自エレメントの開発

Device Connector Framework では、C インターフェイスを使って独自エレメントのプラグインを開発することができます。

8.1 開発用のファイルの準備

C による開発を行うためには、以下の 2 つのファイルが必要です。

- Device Connector Framework のリポジトリに含まれるインクルードファイル `common/include/device_connector.h`
- 静的リンク用のライブラリファイル `libdevice_connector_common.a`。以下のコマンドによりビルドしてください。

```
git clone https://github.com/aptpod/device-connector-framework.git && cd device-connector-framework
cargo build -p device-connector-common --release
```

このコマンドにより、`target/release` 以下に `libdevice_connector_common.a` が生成されます。これを任意のディレクトリにコピーして使用してください。

8.2 エレメントの実装

"example-plugin" という src エレメントを実装する場合、以下のようになります。これを `example_plugin.c` という名前のファイルとして保存します。

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <device_connector.h>

#define N_ELEMENT 1
#define PLUGIN_NAME "example-plugin"

typedef struct {
    const char* text;
} ExamplePlugin;

void *example_plugin_new(const char *config);
DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver);
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer);
void example_plugin_free(void* element);

bool dc_load(DcPlugin *plugin) {
    dc_init(PLUGIN_NAME);
```

(次のページに続く)

(前のページからの続き)

```
plugin->version = "0.1.0";
plugin->n_element = N_ELEMENT;

DcElement *elements = (DcElement *)malloc(sizeof(DcElement) * N_ELEMENT);

// Element settings
elements[0].name = PLUGIN_NAME;
elements[0].recv_ports = 0;
elements[0].send_ports = 1;
elements[0].acceptable_msg_types = NULL;
elements[0].config_format = "json";
elements[0].new_ = example_plugin_new;
elements[0].next = example_plugin_next;
elements[0].finalizer = example_plugin_finalizer;
elements[0].free = example_plugin_free;

plugin->elements = elements;
return true;
}

void *example_plugin_new(const char *config) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)malloc(sizeof(ExamplePlugin));
    example_plugin->text = "hello, world from plugin";
    return example_plugin;
}

DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;

    if (!dc_pipeline_send_msg_type_checked(pipeline)) {
        DcMsgType msg_type;
        if (dc_msg_type_new("mime:text/plain", &msg_type)) {
            dc_pipeline_check_send_msg_type(pipeline, 0, msg_type);
        }
    }

    sleep(1);

    DcMsgBuf *msg_buf = dc_pipeline_msg_buf(pipeline);

    const uint8_t *data = (const uint8_t *)example_plugin->text;
    const size_t len = strlen(example_plugin->text);
    dc_msg_buf_write(msg_buf, data, len);

    return DcElementResult_MsgBuf;
}
```

(次のページに続く)

(前のページからの続き)

```
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer) {
    return true;
}

void example_plugin_free(void* element) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
    free(example_plugin);
}
```

以下、このソースコードについて解説していきます。

```
#include <device_connector.h>
```

プラグイン開発のために device_connector.h をインクルードします。

```
typedef struct {
    const char* text;
} ExamplePlugin;
```

エレメントの本体を定義します。

```
bool dc_load(DcPlugin *plugin) {
    dc_init(PLUGIN_NAME);
}
```

プラグインをロードする時に呼び出される dc_load 関数を定義します。最初に、dc_init にプラグインの名称を渡します。

```
plugin->version = "0.1.0";
plugin->n_element = N_ELEMENT;
```

このプラグインがターゲットとするデバイスコネクタのバージョンと、読み込ませたいエレメントの数を指定します。

```
DcElement *elements = (DcElement *)malloc(sizeof(DcElement) * N_ELEMENT);
```

DcElement の配列を malloc で用意します。この長さは読み込ませたいエレメントの数 (N_ELEMENT) と同じです。

```
elements[0].name = PLUGIN_NAME;           // エレメントの名前（実行時に他のエレメントと重複しているとエラーになる）
elements[0].recv_ports = 0;                // 受信ポートの数
elements[0].send_ports = 1;               // 送信ポートの数
elements[0].acceptable_msg_types = NULL;  // 受け取ることのできるデータ型。何も受信しない場合は NULL
elements[0].config_format = "json";       // 設定フォーマット (json or yaml)。このフォーマットが new_ に指定した関数に渡される
elements[0].new_ = example_plugin_new;    // エレメントを生成する関数
elements[0].next = example_plugin_next;   // エレメントを実行する関数
elements[0].finalizer = example_plugin_finalizer; // プロセス終了時呼び出されるファイナライザを設定する関数
elements[0].free = example_plugin_free;   // エレメントを終了・解放する関数
```

エレメントの詳細を定義します。

```
plugin->elements = elements;
return true;
}
```

plugin->elements に DcElement の配列を設定し、dc_load が成功したら true を返します。

```
void *example_plugin_new(const char *config) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)malloc(sizeof(ExamplePlugin));
    example_plugin->text = "hello, world from plugin";
    return example_plugin;
}
```

エレメントを生成する関数です。config には、パイプライン設定ファイルに記述されたエレメントの設定（conf フィールドの値）が、config_format で指定したフォーマットに変換され文字列として渡されます。（この例では config の値は使用していません。）

ExamplePlugin のための領域を malloc で確保し、初期化後に void ポインタとして返します。失敗時には NULL を返却します。

```
DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
```

example_plugin_next はエレメントを実行するための関数です。受け取った element を ExamplePlugin * にキャストします。

```
if (!dc_pipeline_send_msg_type_checked(pipeline)) {
    DcMsgType msg_type;
    if (dc_msg_type_new("mime:text/plain", &msg_type)) {
        dc_pipeline_check_send_msg_type(pipeline, 0, msg_type);
    }
}
```

dc_pipeline_send_msg_type_checked() で送信するメッセージの型チェックが行われているか調べ、行われていなければ、DcMsgType を作成して dc_pipeline_check_send_msg_type() に渡します。

```
sleep(1);

DcMsgBuf *msg_buf = dc_pipeline_msg_buf(pipeline);

const uint8_t *data = (const uint8_t *)example_plugin->text;
const size_t len = strlen(example_plugin->text);
dc_msg_buf_write(msg_buf, data, len);
```

1 秒間スリープした後、msg_buf を取得し、送信したいデータを dc_msg_buf_write() で書き込みます。ここで書き込むのは example_plugin_new で設定したテキストです。

```
return DcElementResult_MsgBuf;
}
```

DcElementResult_MsgBuf を返し、msg_buf に書き込んだデータを送信することを示します。

```
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer) {  
    return true;  
}
```

プロセス終了時に呼び出されるファイナライザを登録するための関数です。ここでは特に何も行いませんが、プロセス終了時にエレメントが占有するリソースを解放する必要がある場合、ファイナライザに記述します。

```
void example_plugin_free(void* element) {  
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;  
    free(example_plugin);  
}
```

終了処理を記述します。ここでは malloc() で確保した領域を free() に渡すだけです。

8.3 コンパイル

上記の example_plugin.c を、GCC でコンパイルするには以下のコマンドを実行します。

```
gcc -I/include_dir -Wall -O2 -fPIC -shared -L/library_dir \  
-o libdc_example_plugin.so example_plugin.c -lddevice_connector_common
```

-I オプションで、device_connector.h ファイルが含まれるディレクトリを、-L オプションで、libdevice_connector_common.a ファイルが含まれるディレクトリを指定してください。

これにより、プラグインファイル libdc_example_plugin.so が生成されます。プラグインのロード方法については [plugin の設定](#) (p. 12) を参照してください。

09 Device Connector Framework の既知の制約

Device Connector Framework には、以下のような既知の問題・制約があります。

エレメント利用者向け

- タスクの設定において、メッセージの送信元のタスク `from` を設定するときは、送信元のタスク ID は自身のタスク ID より小さい数字である必要があります。
- 番号が 1 以上の送信ポートについては未実装です。

独自エレメントの開発者向け

- 本体とプラグインで、利用するメモリアロケータが異なる場合の動作は未定義です。この問題は、Rust の将来のバージョンでメモリアロケータを固定化できれば回避できます。メモリアロケータの設定を故意に変えなければ問題は起こりません。
- Rust の `Vec` と `String` を、プラグインの FFI (Foreign Function Interface) 間で受け渡しているため、もしこの ABI (Application Binary Interface) に変更があれば、コンパイラのバージョンを本体とプラグインで合わせておかないと、プラグインが壊れる可能性があります。

10 device-connector-intdash とは

アプトポッド製のデバイスコネクタ (device-connector-intdash) は、intdash 用のデータを扱うのに便利な機能を集めたデバイスコネクタです。device-connector-intdash を使用すると、外部デバイスからのデータを、intdash Edge Agent 2 に渡すための FIFO 用データフォーマット (iscp-v2-compat) に簡単に交換することができます。また、EDGEPLANT CAN-USB Interface や、EDGEPLANT ANALOG-USB Interface からのデータを簡単に扱うことができます。

注釈: device-connector-intdash は、Device Connector Framework を使用して開発されています。

10.1 インストール

注意: device-connector-intdash は、intdash Edge Agent 2 をインストールした際に依存パッケージとしてインストールされています。インストールされていることを確認するには、以下のコマンドを実行してください。

```
device-connector-intdash --version
```

device-connector-intdash がインストール済みの場合は、本インストール手順は不要です。

device-connector-intdash を正しくインストールするには以下の要件を満たす必要があります。

ディストリビューションとアーキテクチャー

ディストリビューション	バージョン	アーキテクチャー
Ubuntu	22.04(LTS), 20.04(LTS), 18.04(LTS)	x86_64 (or amd64), armhf, arm64
Debian	11, 10, 9	x86_64 (or amd64), armhf, arm64

device-connector-intdash は、アプトポッドの公開リポジトリで「device-connector-intdash」パッケージとして提供されています。device-connector-intdash をインストールするには、インストール先エッジデバイスのターミナルで以下のコマンドを実行します。

1. アプトポッドの公開リポジトリの設定を行います。

コマンド内の \${DISTRIBUTION} には、ご使用の環境に応じて、ubuntu または debian を指定してください。

```
$ sudo apt-get update
$ sudo apt-get install -y \
  apt-transport-https \
  ca-certificates \
  curl \
  gnupg-agent \
  lsb-release
```

(次のページに続く)

(前のページからの続き)

```
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://repository.aptod.jp/intdash-edge/linux/${DISTRIBUTION}/gpg | \
    sudo gpg --dearmor -o /etc/apt/keyrings/intdash-edge.gpg
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/intdash-edge.gpg] \
    https://repository.aptod.jp/intdash-edge/linux/${DISTRIBUTION} \
    $(lsb_release -cs) \
    stable" \
    | sudo tee /etc/apt/sources.list.d/intdash-edge.list
$ sudo apt-get update
```

2. device-connector-intdash をインストールします。

```
$ sudo apt-get install -y device-connector-intdash
```

注釈: 上記のインストール手順では、推奨される依存パッケージも含めてインストールされます。プラグイン開発を目的に、必要最低限の機能のみインストールしたい場合は、`apt-get install` コマンドに `--no-install-recommends` オプションを付けて実行してください。

```
$ sudo apt-get install -y --no-install-recommends device-connector-intdash
```

推奨パッケージがインストールされていない場合、以下のエレメントを利用できません。これらのエレメントをパイプラインで利用したい場合は、`--no-install-recommends` を指定せずに `device-connector-intdash` をインストールしてください。

- apt-analogtrx-src
- apt-cantrx-src
- h264-split-filter
- pcm-split-filter
- apt-cantrx-sink

10.2 パッケージ情報

利用可能な `device-connector-intdash` のバージョンは、アプトポッドの公開リポジトリの設定後に以下のコマンドで確認が可能です。

表示例：

```
$ apt policy device-connector-intdash
device-connector-intdash:
Installed: 2.0.0
Candidate: 2.0.0
Version table:
*** 2.0.0 500
      500 https://repository.aptod.jp/intdash-edge/linux/ubuntu focal/stable amd64 Packages
      100 /var/lib/dpkg/status
```

プラグインを開発する場合、以下の表を参照し、`device-connector-intdash` パッケージに合った Rust バージョンでプラグインをビルドしてください。

device-connector-intdash パッケージのバージョン	Rust バージョン
2.0.0 以降	1.65.0

重要: device-connector-intdash パッケージを利用してプラグイン開発を行う場合、[Device Connector Framework の既知の制約](#) (p. 33) により、Rust のバージョンを本体とプラグインで合わせる必要があります。

11 device-connector-intdash の機能

11.1 パイプライン設定ファイルの環境変数

device-connector-intdash では、パイプライン設定ファイルの設定値において、環境変数を展開して使用することができます。環境変数を展開するには、パイプライン設定ファイル内で `$(VARIABLE_NAME)` の形式で記述します。

例として、以下のような設定ファイル (conf.yaml) を考えます。path に `$(MY_FIFO_NAME)` が含まれています。

```
...  
  
- id: 3  
  element: file-sink  
  from: [[2]]  
  conf:  
    flush_size: 10  
    path: /var/run/intdash/$(MY_FIFO_NAME).fifo
```

その上で、この設定ファイルを使ってデバイスコネクタを実行する際に環境変数を与えます。

```
$ MY_FIFO_NAME=device123 device-connector-intdash --config conf.yaml
```

そうすると、この場合、path の値が `/var/run/intdash/device123.fifo` となります。

11.2 パイプライン設定ファイルのサンプル

付属デバイスコネクタにはパイプライン設定ファイルのサンプルが付属しています。サンプルは `/etc/dc_conf` ディレクトリにインストールされます。

12 device-connector-intdash に含まれるエレメント

device-connector-intdash には以下のエレメントが含まれています ([Device Connector Framework の基本エレメント](#) (p. 14) を含みます)。

src エレメント

- [apt-analogtrx-src](#) (p. 39)
- [apt-cantrx-src](#) (p. 40)
- [file-src](#) (p. 14) (基本エレメント)
- [nmea-packet-src](#) (p. 42)
- [process-src](#) (p. 15) (基本エレメント)
- [text-src](#) (p. 17) (基本エレメント)
- [ubx-src](#) (p. 42)
- [v4l2-src](#) (p. 44)

filter エレメント

- [h264-split-filter](#) (p. 45)
- [iscp-v2-compat-filter](#) (p. 46)
- [jpeg-split-filter](#) (p. 54)
- [pcm-split-filter](#) (p. 54)
- [print-log-filter](#) (p. 57)
- [stat-filter](#) (p. 16) (基本エレメント)
- [ubx-iscpv2-filter](#) (p. 58)

sink エレメント

- [apt-cantrx-sink](#) (p. 59)
- [file-sink](#) (p. 14) (基本エレメント)
- [null-sink](#) (p. 15) (基本エレメント)
- [stdout-sink](#) (p. 16) (基本エレメント)

注釈: device-connector-intdash には以下のエレメントも付属していますが、intdash Terminal System 専用または実験的なものであるため、直接は使用しないでください。

- [h265-split-filter](#)

独自エレメントを開発する場合は、名前が付属エレメントと重複しないよう注意してください。

12.1 apt-analogtrx-src

エッジデバイスに接続された EDGEPLANT ANALOG-USB Interface からデータを取得するときに使用する src エレメントです。

このエレメントは、デバイスパスにより指定された EDGEPLANT ANALOG-USB Interface から専用カーネルモジュール経由でデータを取得します。そして iscp-v2-compatible フォーマット (bytes 型) で送じます。

注釈: EDGEPLANT ANALOG-USB Interface 用のカーネルモジュールについては、[アプトポッドのウェブサイト](#) の周辺機器についてのページを参照してください。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

カスタムタイプ "iscp-v2-compatible-msgs" (iscp-v2-compatible フォーマット (bytes 型) を複数個連結したメッセージ)

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの conf: 以下に記述します。

項目	設定値	説明
path	文字列	ANALOG-USB Interface のデバイスパス (例: /dev/apb-usb/by-id/usb-xxx)
input_send_rate	整数 (10 / 100 / 1000 / 10000 / 156250 / 312500 / 625000 / 1250000 / 2500000 / 5000000 / 10000000)	ANALOG-USB Interface のサンプリング周波数 [mHz](全ポート共通)
input_enabled	boolean(true / false) 8 個の配列	ANALOG-USB Interface のアナログ入力ポートごと (全 8 ポート) の有効/無効
input_voltage_min	整数 8 個の配列 (0 / -5000 / -10000)	ANALOG-USB Interface のアナログ入力ポートごと (全 8 ポート) の入力電圧 [mV] の最小値
input_voltage_max	整数 8 個の配列 (5000 / 10000)	ANALOG-USB Interface のアナログ入力ポートごと (全 8 ポート) の入力電圧 [mV] の最大値 ただし、同一ポートの最小値と最大値の組み合わせは以下のいずれかである必要があります。 <ul style="list-style-type: none">• min:-10000, max:10000• min:-5000, max:5000• min:0, max:5000 上記に一致しない場合は実行時にエラーになります。
output_enabled	true / false	ANALOG-USB Interface のアナログ出力ポートの有効/無効

次のページに続く

表 1 – 前のページからの続き

項目	設定値	説明
output_voltage	整数 (20～5000 (分解能 20))	ANALOG-USB Interface のアナログ出力ポートから出力する信号の電圧 [mV]
output_wave-form_type	整数 (0 / 1 / 2 / 3 / 16)	ANALOG-USB Interface のアナログ出力ポートから出力する信号の波形 (0:擬似ランダム信号、1:正弦波、2:三角波、3:矩形波、16:固定)
output_frequency	整数 (1000～100000 (分解能 1000))	ANALOG-USB Interface のアナログ出力ポートから出力する信号波形の周波数 [mHz]
clock_id	文字列 (CLOCK_MONOTONIC / CLOCK_MONOTONIC_RAW)	<p>タイムスタンプを算出するときにシステムからどのように時刻を取得するか</p> <ul style="list-style-type: none"> • CLOCK_MONOTONIC: NTP が行う段階的な調整の影響を受ける • CLOCK_MONOTONIC_RAW: NTP が行う段階的な調整の影響を受けない <p>intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。</p>
timestamp_mode	文字列 (device / host) デフォルト: device	<p>受信したデータに対して、タイムスタンプをどのように付与するか</p> <ul style="list-style-type: none"> • device: ANALOG-USB Interface 上で、ANALOG-USB Interface のクロックを元にタイムスタンプを付与する • host: エッジデバイスが ANALOG-USB Interface からのデータを USB 経由で受信したタイミングで、エッジデバイスのクロックを元にタイムスタンプを付与する

12.2 apt-cantrx-src

エッジデバイスに接続された EDGEPLANT CAN-USB Interface からデータを取得するときに使用する src エレメントです。

このエレメントは、デバイスパスにより指定された EDGEPLANT CAN-Interface から、専用カーネルモジュール経由でデータを取得します。そして、iscp-v2-compatible フォーマット (can_frame 型) で送出します。

注釈: EDGEPLANT CAN-USB Interface 用のカーネルモジュールについては、[アプトボットのウェブサイト](#)の周辺機器についてのページを参照してください。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

カスタムタイプ "iscp-v2-compatible-msgs" (iscp-v2-compatible フォーマット (can_frame 型) を複数個連結したメッセージ)

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの conf: 以下に記述します。

重要: EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して apt-cantrx-src と apt-cantrx-sink を同時に使用する場合は、apt-cantrx-src と apt-cantrx-sink の設定をすべて同じにしてください。

項目	設定値	説明
path	文字列	CAN-USB Interface のデバイスパス (例: /dev/ apt-usb/by-id/usb-xxx)
baudrate	整数 (125 / 250 / 500 / 10000)	CAN 通信のボーレート [kbps]
silent	true / false	CAN-USB Interface から CAN バスへの ACK の送信 <ul style="list-style-type: none">• true: ACK を送信しない• false: ACK を送信する
clock_id	文字列 (CLOCK_MONOTONIC / CLOCK_MONOTONIC_RAW)	タイムスタンプを算出するときにシステムからどのように時刻を取得するか <ul style="list-style-type: none">• CLOCK_MONOTONIC: NTP が行う段階的な調整の影響を受ける• CLOCK_MONOTONIC_RAW: NTP が行う段階的な調整の影響を受けない intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。
timestamp_mode	文字列 (device / host) デフォルト: device	受信したデータに対して、タイムスタンプをどのように付与するか <ul style="list-style-type: none">• device: CAN-USB Interface 上で、ANALOG-USB Interface のクロックを元にタイムスタンプを付与する• host: エッジデバイスが CAN-USB Interface からのデータを USB 経由で受信したタイミングで、エッジデバイスのクロックを元にタイムスタンプを付与する

12.3 nmea-packet-src

このエレメントでは、デバイスパスにより指定された TTY デバイスから TTY ドライバー経由で NMEA データを取得します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "text/plain" (1 つの NMEA センテンスが 1 つのメッセージ)

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
path	文字列	GPS デバイスのデバイスパス (例: /dev/ttyUSB9)
baudrate	整数 (0, 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400)	ボーレート (例:57600) termios で定義された左記いずれかの値である必要があります。

注釈:

- nmea-packet-src エレメントは、GPS デバイスの初期化は行いません。あらかじめ GPS デバイス (tty デバイス) が所定のボーレートで NMEA を出力する状態にしてください。
- EDGEPLANT T1 で L4T を使用する場合は、以下のようにすることで GPS デバイスを初期化してデータを取得することができます。
 1. edgeplant-l4t-tools サービスを有効にします。方法については、[EDGEPLANT T1 デベロッパーガイド](#) を参照してください。
 2. パイプライン設定ファイルに以下を追加します。

before_task:

```
- while true; do if test $(stty -F /dev/ttyTHS1 speed) = 57600; then break; fi; sleep 1; done  
- sleep 3
```

12.4 ubx-src

u-blox GNSS モジュールから UBX プロトコルのメッセージを取得し、そのままバイナリデータとして送ります。

取得対象の UBX プロトコルのメッセージは以下のとおりです。

- UBX-ESF-STATUS
- UBX-HNR-STATUS
- UBX-HNR-ATT

- UBX-HNR-INS
- UBX-HNR-PVT

重要: u-blox GNSS モジュールとして NEO-M8U のみをサポートしています。NEO-M8U を搭載していないエッジデバイス（VTC1910-S など）はサポートしていません。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
path	文字列	u-blox GNSS モジュールのデバイスパス (例: /dev/ttyTHS1)
baud_rate	整数 (9600, 19200, 38400, 57600, 115200, 230400, 460800)	ボーレート (例:57600)
meas_rate_ms	小数	GNSS 測定周期 (ミリ秒) (例: 200) 50 ミリ秒以上の値である必要があります。 以下のメッセージの送信頻度に影響します。 <ul style="list-style-type: none">• UBX-ESF-STATUS• UBX-HNR-STATUS
nav_rate	整数	ナビゲーションソリューションに対する測定頻度 (例: 1) 127 以下の値である必要があります。n を指定した場合、各ナビゲーションソリューションに対して n 回の測定を行います (n が大きくなると、送信頻度が下がります)。 以下のメッセージの送信頻度に影響します。 <ul style="list-style-type: none">• UBX-ESF-STATUS• UBX-HNR-STATUS
esf_status_rate	整数	UBX-ESF-STATUS メッセージ送信頻度 (例: 1) n を指定した場合は、 $n * \text{nav_rate} * \text{meas_rate_ms}$ の周期で UBX-ESF-STATUS メッセージが送信されます (n が大きくなると、送信頻度が下がります)。 0 を指定した場合は、送信されません。

次のページに続く

表 3 – 前のページからの続き

項目	設定値	説明
nav_status_rate	整数	UBX-NAV-STATUS メッセージ送信頻度 (例: 1) n を指定した場合は、 $n * \text{nav_rate} * \text{meas_rate_ms}$ の周期で UBX-NAV-STATUS メッセージが送信されます (n が大きくなると、送信頻度が下がります)。 0 を指定した場合は、送信されません。
high_nav_rate_hz	整数	UBX-HNR メッセージの送信間隔 (周波数) (例: 5) 20 以下の値である必要があります。 以下のメッセージの送信頻度に影響します。 <ul style="list-style-type: none">• UBX-HNR-ATT• UBX-HNR-INS• UBX-HNR-PVT
hnr_att_rate	整数	UBX-HNR-ATT メッセージの送信頻度 (例: 1) n を指定した場合は、 $\text{high_nav_rate_hz} / n$ の周波数で UBX-HNR-ATT メッセージが送信されます (n が大きくなると、送信頻度が下がります)。 0 を指定した場合は、送信されません。
hnr_ins_rate	整数	UBX-HNR-INS メッセージの送信頻度 (例: 1) n を指定した場合は、 $\text{high_nav_rate_hz} / n$ の周波数で UBX-HNR-INS メッセージが送信されます (n が大きくなると、送信頻度が下がります)。 0 を指定した場合は、送信されません。
hnr_pvt_rate	整数	UBX-HNR-PVT メッセージの送信頻度 (例: 1) n を指定した場合は、 $\text{high_nav_rate_hz} / n$ の周波数で UBX-HNR-PVT メッセージが送信されます (n が大きくなると、送信頻度が下がります)。 0 を指定した場合は、送信されません。

注釈: 各設定項目の詳細については、u-blox GNSS モジュールのドキュメント "u-blox 8 / u-blox M8 Receiver description" を参照してください。

12.5 v4l2-src

指定されたカメラから V4L2 (Video for Linux 2) 経由で H.264 形式の動画データを受け取り、h264_annex_b 型の iscp-v2-compatible フォーマットで送出します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

カスタムタイプ "iscp-v2-compatible-msg" (h264_annex_b 型)

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの conf: 以下に記述します。

項目	設定値	説明
path	文字列	デバイスパス (例: /dev/video0)
type	文字列 (h264 / jpeg)	ドライバーから取得するデータの種類。 <ul style="list-style-type: none">h264: H.264 のデータを取得します。jpeg: JPEG データを取得します。
width	整数	画像幅。カメラが対応している画像幅を指定してください。
height	整数	画像高さ。カメラが対応している画像高さを指定してください。
fps	整数	FPS。カメラが対応している FPS を指定してください。

12.6 h264-split-filter

メッセージを受け取り、H.264 形式の動画データとしてパースし、iscp-v2-compatible フォーマット (h264_annex_b 型) で送出します。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

H.264 Annex B 形式のバイナリストリームを任意の箇所で区切ったバイト列

送信ポートでの送信形式

カスタムタイプ "iscp-v2-compatible-msg" (h264_annex_b 型)

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
clock_id	文字列 (CLOCK_MONOTONIC / CLOCK_MONOTONIC_RAW)	<p>タイムスタンプを算出するときにシステムからどのように時刻を取得するか</p> <ul style="list-style-type: none"> CLOCK_MONOTONIC: NTP が行う段階的な調整の影響を受ける CLOCK_MONOTONIC_RAW: NTP が行う段階的な調整の影響を受けない <p>intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。</p>
delay_ms	小数	<p>タイムスタンプを付与する際に差し引くオフセット (カメラ処理時間)</p> <p>例えば、カメラ内での処理に 100 ミリ秒かかる場合は、「100」を指定します。これにより、タイムスタンプ付与時に 100 ミリ秒前の時刻が使用されます。</p> <p>設定例:</p> <pre>conf: clock_id: CLOCK_MONOTONIC delay_ms: 100</pre>

12.7 iscp-v2-compatible-filter

データを指定されたフォーマットでパースし、指定に応じてタイムスタンプを付与し、can_frame / jpeg / string/nmea / bytes / float64 / int64 / string いずれかの型の iscp-v2-compatible フォーマットで送出します。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

任意 (timestamp および convert_rule の指定に適合していること)

送信ポートでの送信形式

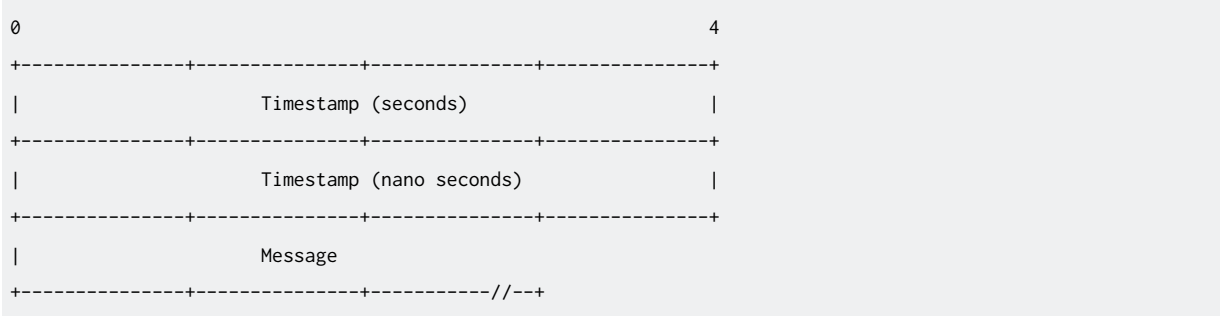
カスタムタイプ "iscp-v2-compatible-msg" (can_frame 型 / jpeg 型 / string/nmea 型 / bytes 型 / float64 型 / int64 型 / string 型)

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
timestamp	文字列 (stamped)、または、タイムスタンプ付与方法を定義したオブジェクト	<p>受信するメッセージにタイムスタンプ（受信データのタイムスタンプ (p. 48)）が含まれているか</p> <ul style="list-style-type: none"> stamped: タイムスタンプが含まれている（このタイムスタンプを出力時にも付与します。） <pre>conf: timestamp: stamped</pre> <ul style="list-style-type: none"> stamp: タイムスタンプが含まれていない（出力時には clock_id の指定に従ってタイムスタンプを付与します。） <pre>conf: timestamp: stamp: clock_id: CLOCK_MONOTONIC</pre> <p>clock_id の指定:</p> <ul style="list-style-type: none"> CLOCK_MONOTONIC: NTP が行う段階的な調整の影響を受ける時計を使用する CLOCK_MONOTONIC_RAW: NTP が行う段階的な調整の影響を受けない時計を使用する <p>clock_id は、intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。</p>
convert_rule	変換ルールを定義したオブジェクト	<p>iscp-v2-compatible フォーマットに変換するためのルールを指定します。詳細は以下を参照してください。</p> <ul style="list-style-type: none"> can (p. 48) jpeg (p. 49) nmea (p. 49) bytes (p. 50) float64 (p. 51) int64 (p. 52) string (p. 53)

12.7.1 受信データのタイムスタンプ

受信するメッセージの先頭にタイムスタンプが追加されている場合は、設定項目 `timestamp` の値を `stamped` とします。この場合、受信したメッセージの先頭 8 バイトがタイムスタンプとしてパースされます。期待するフォーマットは以下のとおりです。



この `Timestamp (seconds)` フィールド、`Timestamp (nano seconds)` フィールドの値は、それぞれ `iscp-v2-compat` での出力時に同名のフィールドに出力されます。

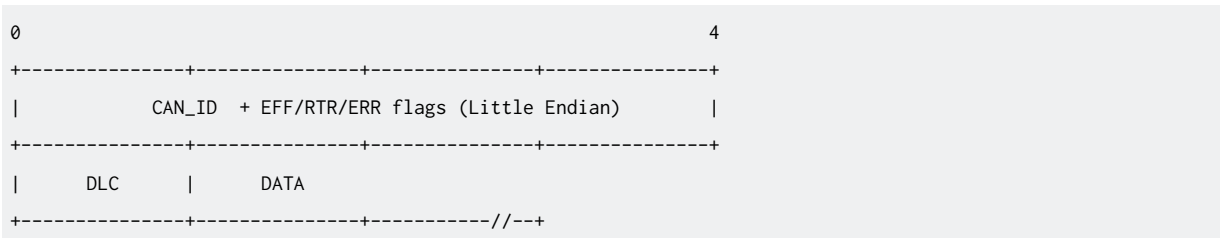
`Message` は、設定項目 `convert_rule` で指定された任意のフォーマットのデータです。

12.7.2 CAN データ用変換ルール (can)

設定例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule:
    can:
      lower_hex: true
```

CAN データ用の変換では、入力されたメッセージを、[Linux SocketCan Driver の `can_frame` 構造体](#) のフォーマットとしてパースします。期待する入力データのフォーマットは以下のとおりです。



この変換ルールでは以下の項目を設定可能です。

項目	設定値	説明
lower_hex	boolean(true / false)	出力する iscp-v2-compatible フォーマットの Data Name フィールドに入れる 16 進数数値表現を小文字で行うか

出力される iscp-v2-compatible の各フィールドは以下のようになります。

- Data Type: can_frame
- Data Name: 上記入力データ内の CAN_ID (16 進 8 桁の文字列表現) (ただし EFF フラグはオフになります)
- Payload: 上記入力データ内の DATA フィールドの内容

12.7.3 JPEG 用変換ルール (jpeg)

設定例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule: jpeg
```

JPEG 用の変換では、入力されたメッセージを、JPEG (ITU-T Rec. T.81 / ISO/IEC 10918-1, 10918-2) としてパースします。期待する入力データのフォーマットは以下のとおりです。

```
0                                     4
+-----+-----+-----+-----+
|               JPEG               |
+-----+-----+-----+-----+//---+
```

出力される iscp-v2-compatible の各フィールドは以下のようになります。

- Data Type: jpeg
- Data Name: 空文字
- Payload: 上記入力データ内の JPEG フィールドの内容

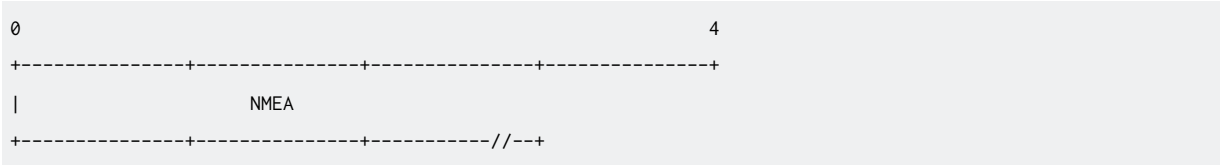
この変換ルールには設定項目はありません。

12.7.4 NMEA データ用変換ルール (nmea)

設定例:

```
conf:
  timestamp:
    stamp:
      clock_id: CLOCK_MONOTONIC
  convert_rule: nmea
```

NMEA データ用の変換では、入力されたメッセージを NMEA 0183 としてパースします。期待する入力データのフォーマットは以下のとおりです。



出力される iscp-v2-compatible の各フィールドは以下のようになります。

- Data Type: string/nmea
- Data Name: <NMEA のトーカ>/<NMEA のメッセージ> (上記入力データ内の NMEA フィールドから抽出されたもの)
- Payload: 上記入力データ内の NMEA フィールドの内容

この変換ルールには設定項目はありません。

12.7.5 Bytes データ用変換ルール (bytes)

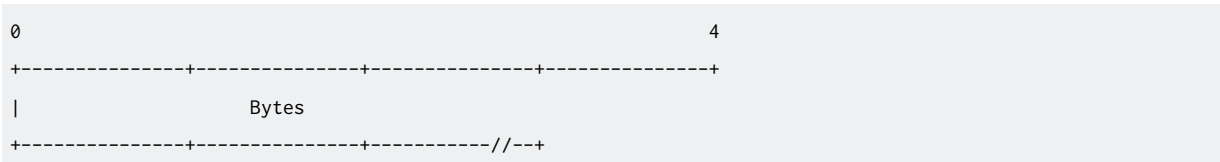
設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    bytes:
      name: null
```

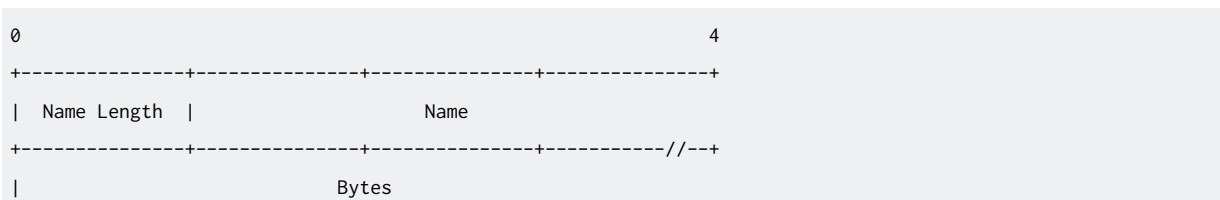
この変換ルールでは以下の項目が設定可能です。

項目	設定値	説明
name	string または null	出力する iscp-v2-compatible フォーマットの Data Name フィールドに入れる文字列

設定項目の name に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。



設定項目の name が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマットは以下のとおりです。



(次のページに続く)

(前のページからの続き)

$$+ \cdots + \cdots + \cdots // - +$$

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compatible の各フィールドは以下のようになります。

- Data Type: bytes
- Data Name: 設定項目 `name` に指定された文字列。または、`name` が `null` の場合、上記入力データ内の `Name` フィールドの内容。
- Payload: 上記入力データ内の `Bytes` フィールドの内容

12.7.6 Float64 データ用変換ルール (float64)

設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    float64:
      name: null
```

この変換ルールでは以下の項目が設定可能です。

項目	設定値	説明
name	string または null	出力する iscp-v2-compatible フォーマットの Data Name フィールドに入れる文字列

設定項目の name に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。

Diagram illustrating the memory layout of a float (Little Endian) in memory. The float is 4 bytes long, with the least significant byte at the lowest address. The diagram shows a horizontal line with tick marks at 0, 4, 8, 12, 16, and 20. The text "Float (Little Endian)" is centered between the 4 and 12 marks.

設定項目の name が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマットは以下のとおりです。

```

0
+-----+-----+-----+-----+
| Name Length |                               Name                               |
+-----+-----+-----+-----+-----+
|
+                               Float (Little Endian)                               +
|

```

(次のページに続く)

(前のページからの続き)

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compatible の各フィールドは以下のようになります。

- Data Type: float64
- Data Name: 設定項目 name に指定された文字列。または、name が null の場合、上記入力データ内の Name フィールドの内容。
- Payload: 上記入力データ内の Float フィールドの内容

12.7.7 int64 データ用変換ルール (int64)

設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    int64:
      name: null
```

この変換ルールでは以下の項目が設定可能です。

項目	設定値	説明
name	string または null	出力する iscp-v2-compatible フォーマットの Data Name フィールドに入れる文字列

設定項目の name に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。

0	4
+-----+-----+-----+-----+	
+ Int (Little Endian)	+
+-----+-----+-----+-----+	

設定項目の name が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマットは以下のとおりです。

0	4
+-----+-----+-----+-----+	
Name Length Name	
+-----+-----+-----+-----+//---+	
+ Int (Little Endian)	+

(次のページに続く)

(前のページからの続き)

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compatible の各フィールドは以下のようになります。

- Data Type: int64
- Data Name: 設定項目 name に指定された文字列。または、name が null の場合、上記入力データ内の Name フィールドの内容。
- Payload: 上記入力データ内の Int フィールドの内容

12.7.8 String データ用変換ルール (string)

設定例:

```
conf:
  clock_id: CLOCK_MONOTONIC
  convert_rule:
    string:
      name: abc
```

この変換ルールでは以下の項目が設定可能です。

項目	設定値	説明
name	string または null	出力する iscp-v2-compatible フォーマットの Data Name フィールドに入れる文字列

設定項目の name に何らかの文字列を設定した場合は、期待する入力データのフォーマットは以下のとおりです。

```
0                                     4
+-----+-----+-----+-----+
|                                     |
|           String                   |
+-----+-----+-----+-----+//---+
```

設定項目の name が null の場合、データ名称は入力データ内で指定します。期待する入力データのフォーマットは以下のとおりです。

```
0                                     4
+-----+-----+-----+-----+
| Name Length |           Name           |
+-----+-----+-----+-----+//---+
|           String                   |
+-----+-----+-----+-----+//---+
```

- Name Length: Name フィールドの長さ。
- Name: データ名称として使用する文字列。

出力される iscp-v2-compatible の各フィールドは以下のようになります。

- Data Type: string
- Data Name: 設定項目 name に指定された文字列。または、name が null の場合、上記入力データ内の Name フィールドの内容。
- Payload: 上記入力データ内の String フィールドの内容

12.8 jpeg-split-filter

jpeg-split-filter は、受信したバイナリストリームを JPEG が複数並んでいるバイナリとしてパースして、1 枚ごとの JPEG に分けて送信します。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

JPEG フォーマットのバイナリを並べたバイナリストリームを任意の箇所で区切ったバイト列

送信ポートでの送信形式

MIME タイプ "image/jpeg" (1 枚の画像が 1 つのメッセージ)

エレメント独自の設定項目はありません。

12.9 pcm-split-filter

GStreamer から出力された音声データを受け取り、bytes 型の iscp-v2-compatible フォーマット に変換して送ります。

重要: エッジデバイスとして T1 / VTC1910-S 以外を使用する場合は正しく設定を行えない可能性があります。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

任意 (前のタスクで process-src エレメントを使って GStreamer から音声を出させ、その出力を本タスクで受け取るようにします。)

GStreamer からの出力は [Raw Audio Media Types](#) における以下のタイプとしてください。

- format: S16LE / S32LE / F32LE
- rate: 48000
- channels: 1
- channel-mask: 使わない (GStreamer コマンドに設定しないこと)
- layout: 使わない (GStreamer コマンドに設定しないこと)

送信ポートでの送信形式

カスタムタイプ "iscp-v2-compatible-msg" (bytes 型)

エレメント独自の設定項目は以下のとおりです。パイプライン設定ファイルの conf: 以下に記述します。

項目	設定値	説明
clock_id	文字列 (CLOCK_MONOTONIC / CLOCK_MONOTONIC_RAW)	<p>タイムスタンプを算出するときにシステムからどのように時刻を取得するか</p> <ul style="list-style-type: none"> • CLOCK_MONOTONIC: NTP が行う段階的な調整の影響を受ける • CLOCK_MONOTONIC_RAW: NTP が行う段階的な調整の影響を受けない <p>intdash Edge Agent 2 の設定と同じにする必要があります。intdash Edge Agent 2 のデフォルトは CLOCK_MONOTONIC です。</p>
delay_ms	小数	<p>データ打刻のオフセットを設定入力します。音声が発生してからデバイスコネクタにデータが届くまでに 100 ミリ秒かかる場合は、100 を指定します。ほとんどの場合 0 で問題ありません。</p>
audio_element	文字列 (Mic Jack/ y Jack-state)	<p>コンピューターのマイクジャックを使用する場合、マイクジャックの状態 (抜き差し) を示すデバイスエレメント名を入力します。</p> <ul style="list-style-type: none"> • Terminal System のコンピューターとして VTC 1910-S を使用する場合: Mic Jack • Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: y Jack-state
audio_iface	文字列 (card /mixer)	<p>音声を制御するデバイスの種別を入力します。</p> <ul style="list-style-type: none"> • Terminal System のコンピューターとして VTC 1910-S を使用する場合: card • Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: mixer

次のページに続く

表 10 – 前のページからの続き

項目	設定値	説明
audio_format	文字列 (S16LE / S32LE / F32LE)	音声データの形式を選択します <ul style="list-style-type: none"> • S16LE: signed 16 bit little-endian • S32LE: signed 32 bit little-endian • F32LE: floating-point little-endian
audio_rate	整数 (48000) (固定)	サンプリング周波数 [Hz]
audio_channels	整数 (1) (固定)	チャンネルの数。
audio_volume_iface	文字列 (mixer)	音量を設定する場合、音量を制御するデバイスの種別を入力します。 <ul style="list-style-type: none"> • Terminal System のコンピューターとして VTC 1910-S を使用する場合: mixer • Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: mixer
audio_volume_element	文字列 (Capture Volume/ y Mic Capture Volume)	音量を設定する場合、音量を示すデバイスエレメント名を入力します。 <ul style="list-style-type: none"> • Terminal System のコンピューターとして VTC 1910-S を使用する場合: Capture Volume • Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: y Mic Capture Volume
audio_volume_value	整数	音量を設定する場合、音量として設定する値を入力します。 <ul style="list-style-type: none"> • Terminal System のコンピューターとして VTC 1910-S を使用する場合: <ul style="list-style-type: none"> • 範囲: 0 - 46 • デフォルト: 28 • 0 のときに 16.00 dB 設定 • 値を 1 上げるごとに 1.00 dB 上昇 • Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合: <ul style="list-style-type: none"> • 範囲: 0 - 20 • デフォルト: 20 • 0 のときに 0.00 dB (音量上昇なし、等倍) 設定 • 値を 1 上げるごとに 1.00 dB 上昇

次のページに続く

表 10 – 前のページからの続き

項目	設定値	説明
audio_boost_element	文字列 (Mic Boost Volume/ y Mic Boost Capture Volume)	音量を設定する場合、ブーストを示すデバイスエレメント名を入力します。 <ul style="list-style-type: none">Terminal System のコンピューターとして VTC 1910-S を使用する場合: Mic Boost VolumeTerminal System のコンピューターとして EDGEPLANT T1 を使用する場合: y Mic Boost Capture Volume
audio_boost_value	整数	音量を設定する場合、ブーストとして設定する値を入力します。 <ul style="list-style-type: none">Terminal System のコンピューターとして VTC 1910-S を使用する場合:<ul style="list-style-type: none">範囲: 0 - 3デフォルト: 00 のときに 0.00 dB 設定値を 1 上げるごとに 10.00 dB 上昇Terminal System のコンピューターとして EDGEPLANT T1 を使用する場合:<ul style="list-style-type: none">範囲: 0 - 3デフォルト: 10: -20.00 dB (Mute)1: 0.00 dB (gain なし)2: 20.00 dB3: 30.00 dB

12.10 print-log-filter

受け取ったメッセージの数とサイズの情報をログとして出力します。ログの出力先としてデバイスコネクタのログまたは標準エラーを選択できます。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

任意

送信ポートでの送信形式

任意（受け取ったメッセージをそのまま送信ポートに書き出します）

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

項目	設定値	説明
interval_ms	小数	ログを出力するミリ秒単位の周期（例：1000） 例えば「1000」を指定すると、受け取ったメッセージ数と合計サイズを、1000 ミリ秒ごとにログとして出力します。
tag	string	ログを識別できるように各行の先頭に出力されるタグ。 connector1 と指定した場合は、以下のように出力されます。 [connector1] 100 msgs, 1000 bytes
output	string (log-trace / stderr)	ログの出力先を指定します。 log-trace: デバイスコネクタのログの仕組みを使用して出力します。ログレベルは trace となります。 stderr: 標準エラーに出力します。

12.11 ubx-iscpv2-filter

ubx-src から UBX メッセージを受け取り、メッセージをパースして Data Name をバイナリデータの先頭に付与し、iscpv2-compat-filter に送信します。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

MIME タイプ "application/octet-stream" (ubx-src が送信したメッセージ)

送信ポートでの送信形式

MIME タイプ "application/octet-stream" (iscpv2-compat-filter が受信するメッセージ)

エレメント独自の設定項目はありません。

12.12 apt-cantrx-sink

メッセージを受け取り、iscp-v2-compatible フォーマット (can_frame 型) としてパースし、専用カーネルモジュール経由で EDGEPLANT CAN-USB Interface に送出します。

注釈: EDGEPLANT CAN-USB Interface 用のカーネルモジュールについては、[アブトボットのウェブサイト](#)の周辺機器についてのページを参照してください。

エレメントの種類

sink

前のエレメントから受信するポートの数

1

受信ポートでの受信形式

任意 (iscp-v2-compatible フォーマット (can_frame 型) を複数個連結したバイナリストリーム)

エレメント独自の設定項目は以下のとおりです。conf: 以下に記述します。

重要: EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して apt-cantrx-src と apt-cantrx-sink を同時に使用する場合は、apt-cantrx-src と apt-cantrx-sink の設定をすべて同じにしてください。

項目	設定値	説明
path	文字列	CAN-USB Interface のデバイスパス (例: /dev/apb-usb/by-id/usb-xxx)
baudrate	整数 (125 / 250 / 500 / 10000)	CAN 通信のボーレート [kbps]
silent	true / false	CAN-USB Interface から CAN バスに ACK を送信しない設定の有効/無効 (true:ACK なし、false:ACK あり)
clock_id	文字列 (CLOCK_MONOTONIC / CLOCK_MONOTONIC_RAW)	タイムスタンプを算出するときにシステムからどのように時刻を取得するか 本エレメントはタイムスタンプを付与しないため、任意の値で構いません。 ただし、EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して、apt-cantrx-src と本エレメントを同時に使用する場合は、apt-cantrx-src と同じ設定にしてください。

次のページに続く

表 11 – 前のページからの続き

項目	設定値	説明
timestamp_mode	文字列 (device / host) デフォルト: device	受信したデータに対して、タイムスタンプをどのように付与するか 本エレメントはタイムスタンプを付与しないため、任意の値で構いません。 ただし、EDGEPLANT CAN-USB Interface の 1 つのデバイスパスに対して、apt-cantrx-src と本エレメントを同時に使用する場合は、apt-cantrx-src と同じ設定にしてください。