UNIX 機における IoT 機器制御のためのタイミング管理

松浦 智之
(有)USP 研究所
richmikan@richlab.org

柳戸 新一
(有)USP 研究所
s.yanagido@gmail.com

鈴木 裕信
(有)USP 研究所
h-suzuki@usp-lab.com

大野 浩之
金沢大学
hohno@staff.kanazawa-u.ac.jp

要旨

さまざまな電子デバイスを *IoT* 機器として活用するためには，*UNIX* 系 *OS* 搭載コンピュータ（以降，*UNIX* 機）と接続・連携できると都合が良い．なぜなら，*UNIX* 機は *TCP/IP* スタックを既に持っていて，かつ広く普及してるなど，開発や保守にかかるコストを抑えられるという期待が持てるからである．しかし，*UNIX* 機からそれら電子デバイスを直接制御する場合には，解決しなければならない課題が多い．例えば，プリエンプティブなタスクスケジューラや，パイプライン上のバッファリング機能などにより，制御にとって重要な「タイミング」を損なう要因が *UNIX* にはいくつかある．また，*POSIX*（*POSIX.1-2017*）の範囲では精密なタイミング管理のためのコマンドも十分に揃っていない．

そこで著者らは，タイミングの変動を低減あるいは管理可能にするためのアイデアと，それに基づくコマンドを考案した．ただし，*POSIX* 仕様を逸脱しない範囲の *UNIX* 機（*POSIX* 機と称す）で実現するという制約を課した．逸脱するほど *UNIX* というプラットフォームから外れていき，*UNIX* が本来持っている高い汎用性や持続性という恩恵が得られなくなっていくからである．そして，新たに作成したタイミング管理コマンドを用い，*POSIX* 機での *PID* 制御による倒立振子を実現するなど，いくつかの研究で実用性が確認できたため，*POSIX* 機でのタイミング管理実現のための考え方，および作成したコマンド実装について本論文で報告する．

1. はじめに

2021 年現在，産業用機器から家電に至るまで，UNIX 系 OS 搭載コンピュータ（以降，UNIX 機[*1]）は組み込み機器の分野においても広く活用されている．これは，UNIX の系譜がさまざまに分派した後に POSIX という共通インターフェース仕様が提唱され，POSIX は無償かつ改訂も少なく準拠しやすかったことから，普及率の高いプラットフォームになれたことが大きな要因である．さらに，今や UNIX 機は TCP/IP スタックを標準的に実装しているため，UNIX 機を活用することで IoT 機器を開発することも容易になった．しかし，UNIX 機を活用したことで UNIX 機が持つ高い互換性や持続性がシステム全体として発揮されるようになったとは言いきれない．

文献 [1] には工場のネットワーク構成図の一例が示されているが，同様にして UNIX を活用した IoT 機器について考えると典型的に図 1 の (a) のような構成になっている．UNIX 機はインターネットクラウド上に存在するサーバから受けた操作指示をローカルデバイス機器に伝達したり，あるいはローカルデバイス機器から得られた動作状況をサーバに報告する役割を担ったりという，サーバとローカル機器の橋渡し役を担うのみでデバイス機器の制御自体は担当しない．UNIX 機と末端のセンサやアクチュエータの間には別途コントローラが介在し，計測と制御を担当する．そのコントローラ自体もまたプロセッサを搭載したコンピュータであり，別の OS ある

[*1] 仮想マシンも本論文における UNIX 機の範疇とする．

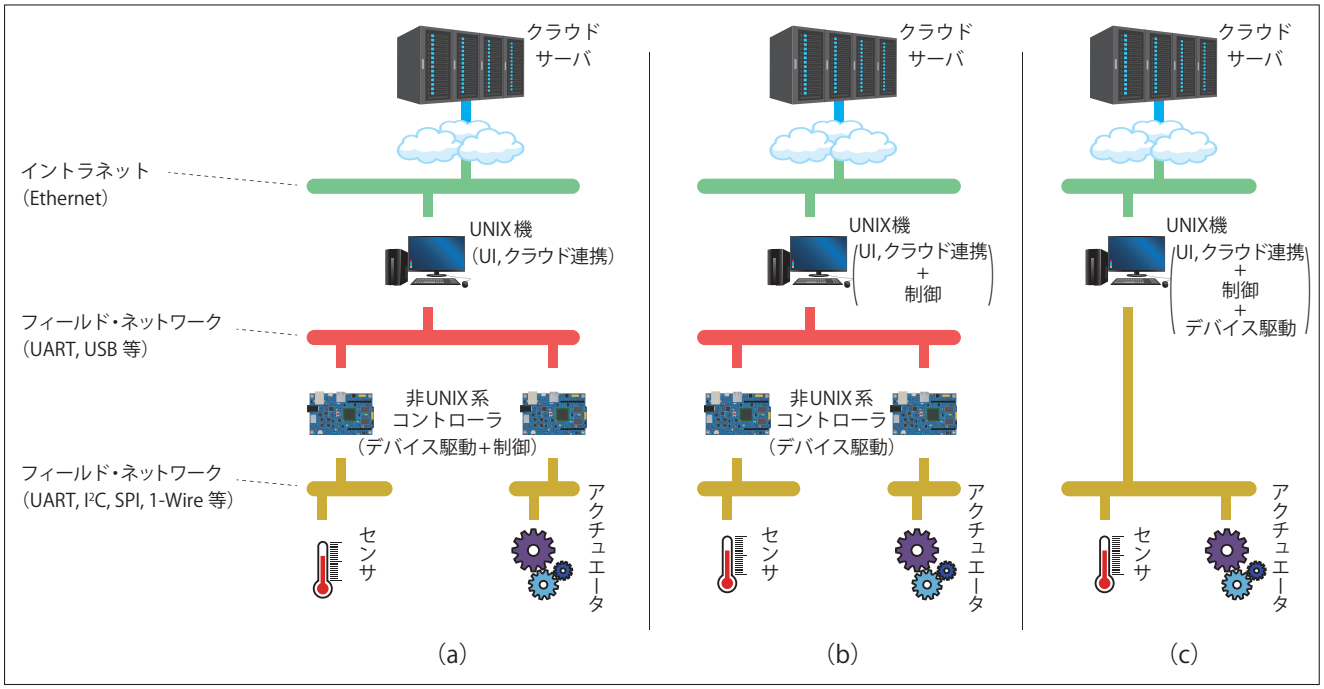

図 1. UNIX 機を利用した IoT 機器の構成例（文献 [1] の図を元に著者らが作成）

いはプログラミング言語で動作している．役割を単純化して分担を進めることにはシステム全体の見通しを良くする効果がある．

しかし，図 1 の (b) や (c) のような構成の方がより適切なケースも考えられる．(b) は制御の役割をコントローラから UNIX 機へ移したもの，(c) はさらにセンサやアクチュエータの駆動についても UNIX 機が担うことでコントローラを排したものである．これにより，以下の利点が得られる．

- UNIX 機に役割を委譲することによる，コントローラ側のソフトウェア・ハードウェアの簡素化や削減
- POSIX 仕様ベースでの実装の割合増加による，システム全体としての互換性・持続性の向上

特に後に示した利点は，著者らが「POSIX 中心主義」[2] と呼ぶものである．目的の機能・性能を得るために最適であるという理由のみでプラットフォーム（OS やライブラリ，プログラミング言語等）を選択すると，保守コストが増大したり持続性を損なうリスクが高い．反対に，持続性で高い実績のある POSIX に最大限準拠した開発を行うことで，長期にわたる持続性を得られる可能性が高いことを前述の論文で示した．

ところが，(b) や (c) のような構成にするには，少なくとも以下の問題がある．

1. プリエンプティブなスケジューラにより，各プロセスの動作タイミングの予測が困難
2. パイプラインのバッファリングにより，データの到来タイミングの予測が困難
3. タイミングを管理（記録や生成）するためのコマンドの不足

それぞれの詳細は次節で述べるが，上記 1 への対策としてはリアルタイム OS（RTOS）化された UNIX の規格や実装がある．例えば商用ライセンス製品では QNX[3] があり，POSIX のリアルタイム拡張である POSIX 1003.1b に準拠した API を持つ．一方オープンソースでは，RTLinux[4] や ART-Linux[5] が Linux カーネルに対するパッチとして提供されている．しかし，これらはそれぞれに動作可能なハードウェアが限定され，具体的なプログラミング方法が違うなどにより，非リアルタイムな通常の UNIX 機のような汎用性及び互換性を有してはいない．実際，RTLinux を導入しようとした場合，まず使用を予定しているハードウェアがそれをサポートしているかを確認し，さらに使用を予定している Linux カーネ

ルのバージョンのソースコード一式と，それに対応したRTLinux パッチを用意し，ビルドを成功させなければならない．また，リアルタイム性を確保するために，リアルタイムプロセスからは一般のシステムコールが利用できないなど，プログラミング上の特別な制約もある．加えて，組み込み機器はしばしば大規模システムの基盤としても活用されるため，製品寿命の長さは重要な性能の一つである．

なお，Linux 2.6.23 以降から導入された Completely Fair Scheduler（CFS) スケジューラ [6] では FIFO（First In, First Out）スケジュールポリシーを用いるなどでリアルタイム処理に対応することが可能である．しかし，これもアプリケーションの実行時にスケジュール・ポリシーを変更するなどの対応が別途必要である．

以上の背景に基づき，著者らは，広く普及している非リアルタイムな UNIX 機全般で通用する範囲，すなわちオプション扱いである POSIX 1003.1b を含まない POSIX（POSIX.1-2017 とする）[7] のみに準拠させながら，前述の問題解決を図った．

2. UNIX 機のタイミング管理の問題点

前節でも指摘したように，UNIX 機で実用的なタイミング管理を実現するには，次の三つの問題への対策が必要である．

2.1. スケジューラによる処理タイミングのぶれ

非リアルタイムかつプリエンプティブな通常の UNIX 機には，動作中の各プロセスにリソースを公平に割り当てるためのスケジューラが実装されており，処理タイミングにぶれが生じる．(POSIX の範囲ではないが）ナノ秒単位のタイムスタンプと stat コマンドを持つ UNIX 機であれば，次のコマンドによりその大きさを確認できる．

```
$ touch a; sleep 1; touch b; sleep 1; tou
ch c; sleep 1; ... ; touch k
$ stat [a-k]
```

このコマンドにより 11 個のタイムスタンプが得られ，順番的に隣り合うタイムスタンプ間の経過時間が 10 か所で求められる．それら経過時間のばらつきから，UNIX 機の処理タイミングのぶれの大きさがわかる．

この試験を，組み込み用途の UNIX 機としてもよく用いられている Raspberry Pi 3B+に Raspberry Pi OS (buster) を標準インストールした直後の環境で実行したところ，上記のようにして求めた経過時間同士の差の大半はナノ秒オーダであったものの，最も大きいものと小さいものでは最大 13 ミリ秒の差（処理タイミングのぶれ）があった．従って，少なくともこの環境では，マイクロ秒単位やそれ以下の精度を要求するときはプロセスのスケジューリングを意識し，優先度を変更する，あるいはスケジューリングポリシーを変更するなどの手続きが別途必要となる．

2.2. バッファリングよるデータ転送の遅延

標準入出力のバッファリングによっても発生する遅延がある．その様子は次のワンライナで観測できる．

○バッファリング遅延の再現

```
$ while sleep 1; do date; done | tr 1 1 |
 cat
```

○ stdbuf コマンドによる解決方法

```
$ while sleep 1; do date; done | stdbuf -
oL tr 1 1 | cat
```

このワンライナは 3 段階のコマンドから構成される．1 段目では，date コマンドによる現在時刻を 1 秒の休みを置きながら出力している．2 段目ではその文字列を tr コマンドが受け取るものの，結果的に何も手を加えず内容をそのまま次に送る．3 段目では，cat コマンドがやはり何の加工もせず次へ送るので，刻々と変わる現在時刻を画面に表示することになる．この時，最初の sleep コマンドの作用によって約 1 秒ごとに現在時刻が一行ずつ画面に表示されるわけではなく，実際には数十秒以上経ってから，それまで出力されなかった分の時刻が一斉に表示される．

この数十秒以上の遅延は，タスクスケジューラの影響ではなく，明らかにバッファリングによるものである．制御にとってこれは過大な遅延であるうえに，可変長データを扱う場合，どのタイミングでバッファリングされたデータがフラッシュされるのかの予測も困難である．

バッファリングモードを変更するコマンドとしては，GNU プロジェクトが stdbuf コマンドを公開している．これにより，Linux であれば 2 番目に例示したワンライナでこの問題が解決する．しかし stdbuf は POSIX に掲

載されているものではないので，すべての UNIX 機で使える保証がないうえに，効果が及ばない場合もある．実際，macOS にあるオリジナルの `tr` コマンドには作用しない．macOS 上で必要な場合は，GNU core utilities 版の `gtr` というコマンドをインストールして使わなければならない．また，コマンド自身がバッファリングモードを管理しているもの（`perl` や `python` コマンド等）にも通用しない．これらは，各コマンドのスクリプトやオプションで切り替えなければならない．このように，UNIX 機というだけではバッファリングに関して処理タイミングの互換性を担保できない．

2.3. タイミング管理・生成用のコマンド不足

データ転送における情報の価値は，流れるデータの値のみならず，その送受信タイミングにもある．キーボードやマウスを始めとしたヒューマン・インタフェース・デバイスとの通信はその代表例である．

しかし，POSIX の範囲で提供されているコマンドで，時間調整に使えるコマンドはほとんど無い．時刻を知るコマンドとしては `date`，待機時間を作るコマンドとしては `sleep` があるが，どちらも POSIX 仕様としては秒単位までででの利用しか保証されていない．そのために，UNIX 機によって対応・非対応が分かれるうえ，例え対応していたとしてもコマンド書式が統一されていないので，例えば Linux なら Linux 用に限定したシェルスクリプトを書かざるを得ない．

また仮に，POSIX 仕様で定められた秒未満対応の `date`，`sleep` コマンドがあったとしても，シェルスクリプトの動作速度を考慮すると十分とは言えない．

3. UNIX 機のタイミング管理の改善

前節で述べた問題に対し以下の対応を行った．

3.1. データの行列指向テキスト形式化とタイムスタンプの付加

2.1 項では，処理タイミングのぶれが 10 ミリ秒の規模で生じる例を示したが，これ自体は非リアルタイムかつプリエンプティブな UNIX 機を使っている限り避けられない問題である．しかし，制御において，出力の計算にはサンプリング間隔 Δt の管理が重要であるため，仮に各周回の実行時間 Δt を精密に測定できれば，制御においては次の周回の出力値計算のために利用でき，UNIX 機上の処理タイミングのぶれを補償できる．そのためには，UNIX 機の中を通過する各周回の計測値・出力値データにタイムスタンプを付けられればよい．

3.1.1 ものグラミング 2

データにタイムスタンプを付加するために，今回は大野らが提唱する「ものグラミング 2」[8] というアプローチを採用した．

ものグラミング 2 は IoT デバイス開発手法の一つであり，その特徴はデバイスに対して次の三つの要求をすることにある．

- やり取りするデータは，バイナリ形式ではなく，原則として行列指向のテキスト形式とすること．
- UNIX 機からキャラクタデバイス（標準入出力に接続可能なデバイス）として扱えるインターフェース（例えばシリアルポート）でのデータ授受をすること．
- 複雑な計算は UNIX 機に任せ，センサ読み取りまたはアクチュエータ操作の役割に徹し，汎用性・持続性の高いシンプルなハードウェアの選択（例えば Arduino）に努めること．

デバイスがこれらのルールを守ることで，UNIX の持つ高い汎用性・持続性をシステム全体としても継続させやすくなる．また，デバイスに対するコストも抑えられ，多くの人が慣れ親しんでいる UNIX によるプログラミングの知識が活かせる領域が広がるために学習コストも抑えられる．

これがものグラミング 2 の概要である．

3.1.2 行頭へのタイムスタンプ列の追加

UNIX におけるテキスト形式は，基本的にデータ長は固定されておらず，次の改行コードまでが一つのレコードとして扱われる．その性質により基本的な UNIX コマンド (`awk` 等) を使って列の挿入・削除が簡単に行える利点を有している．従って，行列指向のテキストデータであれば，タイムスタンプの列を各行頭に挿入できるという柔軟性をもつ．

例えば，ものグラミング 2 の要求に従い，シリアルポートにテキスト形式で分解能 10 ビットの 3 軸加速度センサの読み値を 0.1 秒毎に送るデバイスであれば，図 2 のように行頭に UNIX 機への到達時刻を付加する．

```
20210310090000.000000 92 411 823
20210310090000.100000 98 412 831
20210310090000.200000 101 411 834
                    :
                    :
```

図 2. タイムスタンプ付加データの例

この例では，カレンダ日時形式（YYYYMMDDhhmmss）にマイクロ秒精度の小数を付けたフォーマットとしているが，年月日の並べ方の習慣やタイムゾーンで齟齬が生じる恐れがある場合には，UNIX 時間で表現する方がよい．いずれにせよ時刻が記録されていれば，各値がいつ到来したものかという情報が残り，次の行との時間差（Δt）も求められる．

UNIX 側でタイムスタンプを付加する場合には，既に述べたように，タスクスケジューラによる処理タイミングのぶれによる精度の限界はあるが，デバイス側にリアルタイムクロック（RTC）を搭載しなくてよいという利点がある．RTC の搭載によって利便性が向上するように見えるが，実際には搭載した RTC の精度を把握する必要があるだけでなく，RTC が保持する時刻を基準時に同期させる必要があるなど，タイムスタンプ問題をすべて解決するには至らない．

3.2. stdbuf より効果のあるバッファリング回避方法

2.2 項でバッファリングによる遅延の発生と，`stdbuf` コマンドでもなおこの問題を解消できない例を示した．

`stdbuf` コマンドは，一部の UNIX 実装で提供されている LD_PRELOAD 機構を用い，動的リンクによってバッファリングモード切替ルーチンを対象コマンドに送り込み，対象コマンドの起動時に実行させている [9]．この機構は POSIX で保証されたものではなく，そもそも静的リンクでビルドされたコマンドには作用せず，stdbuf の効かないコマンドが存在する一因になっている．

著者らは，POSIX 文書に記載されているライブラリやシステムコールで，より簡単にこの問題を解決した．

3.2.1 標準出力への疑似端末（PTY）の接続

文献 [10] の 5.4 Buffering によれば，ANSI C の要請により，標準入出力ライブラリ `stdio.h` が提供するバッファリングモードは，標準エラー出力を除く標準入出力が対話的な装置に対する場合に限り，完全バッファリングではなく行バッファリングをデフォルトとしている．

そこで，`stdbuf` コマンド同様に，ラッパーコマンドを作る．ここではそれを `ptw`（Pseudo Terminal Wrapper）コマンドと呼ぶ．`ptw` コマンドは，疑似端末（PTY）を作成し，ラッピング対象コマンドの標準出力と次のコマンドの標準入力の間に介在する．すると対象コマンドの `stdio.h` ライブラリは，対話的な端末に接続されていると思い込んで自発的に行バッファリングモードにすると期待できる（図 3）．3.1 項のようにしてデータは既に行単位にしてあるため，行単位のバッファリングであれば実質的にバッファリングの影響は無い．

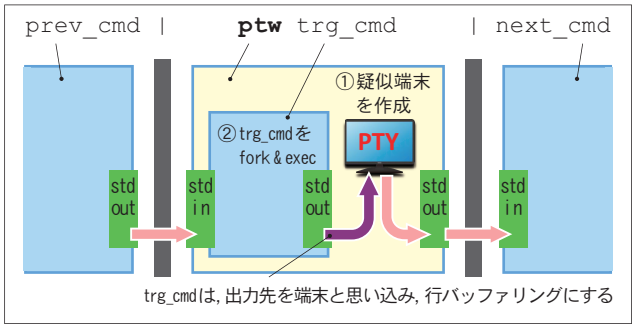

図 3. 疑似端末を利用した完全バッファリング回避

3.3. タイミング生成

2.3 項で述べたコマンドの貧弱さへの対策として以下を実施した．

3.3.1 タイムスタンプ付データの活用

これには 3.1 項で発案した行列指向テキストデータ化とタイムスタンプの付加が役立つ．タイムスタンプ付のデータを一行ずつ読み取り，そこに記されているタイムスタンプから前後の行との時間間隔に基づいた休みを挿入しつつ，タイムスタンプの列を除いた残りの文字列を標準出力に書き出すことでタイミングを再現できる．

タイミングを再現するのではなく生成したい場合には，前述のコマンドの前段階で，タイムスタンプを計算して

決めるプログラムを置けばよい．例えば awk コマンドを使い，出力値の変化が緩やかな区間では長い間隔で，激しい区間では短い間隔で出力時刻を定めることもできる．また，倍速再生などもできる．先程と同様に awk コマンドを使い，タイムスタンプの増分が 1/2 になるようにタイムスタンプを変更すればよい．

3.3.2 処理タイミングのぶれへの対策

UNIX 機の処理タイミングのぶれによってタイムスタンプの指示した時刻から実際の出力時刻がずれることが想定される．これに対しては，次の行の出力時刻はその都度 RTC を参照しながら決めることにより，局所的にはずれても，大域的には相殺できる．

ただし，遅れた場合にその遅れを次の周回で取り戻そうとして短時間しか休まなければ，瞬間的な速度が想定を上回り，その先に繋がれたデバイスがデータを取りこぼす恐れが考えられる．デバイス側が十分なバッファを搭載するなど，フロー制御が適切に動作している場合には問題ないが，そうでない場合には，一つの周回で取り戻してよい遅れの時間を制限したり，あるいは一切許さないなどの対策もしなければならない．

4. タイミング管理コマンドの実装

以上の解決案に基づき，著者らはタイミング管理機能を強化するための UNIX コマンドをいくつか作成し，公開した [11]．なお，本論文で冒頭から述べているように，UNIX の高い互換性・持続性は極めて重要であるため，作成したコマンドはすべて，C99（ISO/IEC 9899:1999）または POSIX 文書に掲載されている関数さえサポートされている環境であればビルドでき，かつ使用できることを保証した．

4.1. linets コマンド

標準入力または引数で指定したファイルからテキスト形式のデータを受け取ると，各行の到来時刻を行頭に付加して標準出力へ書き出す．到来時刻は，カレンダ日時（タイムゾーンは環境変数 TZ で変更可），UNIX 時間，コマンド起動時からの秒数，1 行目到来時刻からの秒数の 4 種類が選択でき，出力精度についても，秒，ミリ秒，マイクロ秒，ナノ秒から選択できる（ただし信頼度は UNIX 機の動作速度や処理タイミングのぶれによる制限を受ける）．

また，制御の分野では時間差（Δt）が利用されることも考慮し，タイムスタンプ列の後（2 列目）に前行からの経過秒数を追加するオプションも用意した．

なお，各行の到来日時のタイムスタンプを付記するコマンドとして moreutils[12] の ts コマンドがあるが，Perl で書かれており，かつ一部オプションを利用するには追加 Perl モジュールを要するなどにより POSIX 中心主義に反するため，独自実装した．

4.2. ptw コマンド

これは，2.2 項で述べた仕組みを実現するためのラッパーコマンドである．

書式は単純で，対象となるコマンドの手前に半角空白を一つ置いて “ptw” の記述を追加するだけである．

ライブラリを静的リンクしたコマンドなど，stdbuf コマンドでは効果の及ばなかったものの多くに効果があるが，次のような一部のコマンドには効果が無い．

- 意図的に完全バッファリングモードを指定している場合
- mawk コマンドなど，ANSI C や stdio.h の要請に従わず，出力先が端末でも行バッファリングにしない場合（mawk では，-W interactive オプションが必要）
- シェルスクリプトで書かれており，内部でパイプによって別のコマンドをカスケーディングしている場合（内部で使用しているコマンドが “*cmd*” である時，シェルスクリプトの冒頭で “alias='ptw *cmd*'” などとエイリアスを定義することで対応可能）

4.3. tscat コマンド

タイムスタンプを意識した cat コマンドという由来で，linets コマンドの反対の動作をするものであり，linets の生成するタイムスタンプ形式を理解する．すなわち，1 列目のタイムスタンプが示す時刻が到来した時に 2 列以降行末までを標準出力に書き出し，次の行を読み込む．ただしこれではタイムスタンプが過去のものであった場合には意味が無いので，コマンド起動時の時刻または 1

行 1 列目に記されている時刻からの相対時間で出力時刻を決めるオプションも用意した．

なお，linets コマンド同様，タイムスタンプはナノ秒単位まで対応しているが，実際の信頼度は UNIX 機の動作速度や処理タイミングのぶれによる制約を受ける．ただし，1 行毎に clock_gettime() 関数が返す現在時刻を参照しながら次の出力時刻を決めているため，大域的にはそれらの影響が打ち消される．

4.3.1 linets, ptw, tscat の応用例

著者らの web サイトに，以上の三つのコマンドの使用例を掲載した [13]．今回作成したコマンドの他，awk や cat，tee など，基本的な UNIX コマンドをパイプ（|）でカスケーディングすることで，ロボットの手動操作を記録・再生，0.5 倍速再生するという応用もできる．

4.4. その他のコマンド

タイミング管理のため，他にコマンドを二つ作成した．

4.4.1 valve コマンド

「弁」が由来のこのコマンドは，標準入力またはファイルから到来したデータを 1 バイト毎または 1 行毎に，それぞれが一定間隔になるように標準出力に送り出すコマンドである．

例えば，ファイル sensor1 の内容を，300 bps で出力したければ “valve 300bps sensor1” と書け，行単位に各行を 100 ms 毎に出力したければ “valve -l 100ms sensor1” と書ける．

このコマンドを作成した理由の一つは，3.3.2 の後半で述べたオーバーフロー対策である．タイムスタンプに従う tscat コマンドに実装すると複雑になるため，別コマンドとして作成し，二つのモードを用意した．瞬間的にも速度超過を避けるため，例え遅れたとしても次の周回で後れを取り戻すことを一切しないモードと，一定の範囲内であれば取り戻しを許容するモードである．

なお，流速を調整できるものとして pv コマンド [14] が知られているが，そちらはデータ送出周期が約 90ms（バージョン 1.6.6 の場合）で，その周回毎に設定流速に沿った量のデータをまとめて送るのに対し，valve コマンドは 1 バイトまたは 1 行送るごとに POSIX の nanosleep() 関数を呼び出すことで，より高品質にタイミング再現ができるように設計した．

4.4.2 秒未満に対応した sleep コマンド

これは，一部の UNIX 実装で提供されている sleep コマンドと同様に，引数に指定できる待機時間を小数を含む値（ナノ秒単位まで）にも対応させたものである．

車輪の再発明をした理由は，一部の UNIX 実装にしか存在しないそのような sleep コマンドを POSIX 環境で保証するためである．

なお，本項に記した二つのコマンドも他のコマンドと同様，実際の信頼度は UNIX 機の動作速度や処理タイミングのぶれによる制約を受ける．

5. 使用事例

5.1. 非 RTOS 化 Linux による倒立振子制御

linets と ptw の二つのコマンドにより，同軸二輪車型倒立振子（名称: クララ[*2]）を立たせることに成功した．

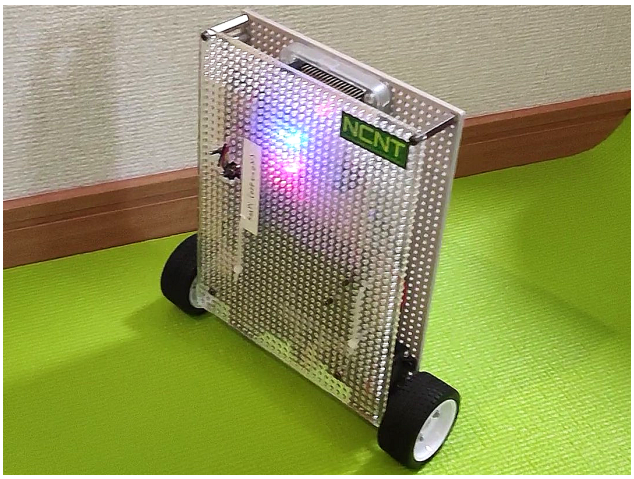

図 4. 倒立振子「クララ」

クララは，3 軸の加速度センサとジャイロセンサ，ロータリーエンコーダを装備し，逐一それらを計測し，カルマンフィルタによる真値推定と，制御による車輪モータへの出力値を計算しながら倒立する．柳戸らの報告 [15] では，制御アルゴリズムとして最終的に線形 2 次レギュレータ（Linear Quadratic Regulator:LQR）を採用した

[*2] Klara．正式名称は “Kinetical Leg Astability Runs Away.”

が，選定段階では古典制御論の重要な成果物である PID 制御でも実装したので，本稿ではこちらを解説する．

この実験の重要な点は，前述の真値推定と制御計算を，カーネルに一切カスタマイズや調整を加えていない通常の Linux（Raspberry Pi Zero + Raspbian[*3]）で実現した点である．Linux カーネル 2.6.23 以降ではプロセスのスケジューラは CFS が標準となっているが，その Linux カーネルの環境において別途リアルタイム処理などの設定などの適切な指定を行わないと，2.1 項で示したような処理タイミングのぶれが生じる可能性がある．そこで `linets` コマンドによって，センサから送られてきた値に Raspberry Pi 側のタイムスタンプと前回測定時刻との差 Δt を付加し，カルマンフィルタと PID 制御のプログラムで利用することで処理タイミングのぶれを考慮した制御を実現した．

図 5 に，倒立動作中にクララが記録した，時刻対角度のグラフを示す．

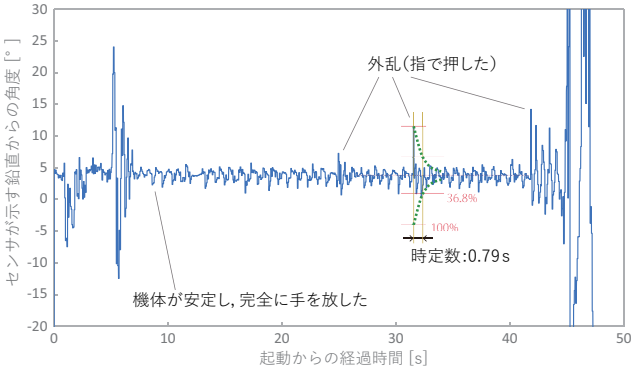

図 5. 倒立振子「クララ」の時定数推定

グラフによれば，起動から 9 s 頃に機体が安定したために手を放され，加速度センサに基づく鉛直からの傾き角が約 3.8 °で安定し，その後　25 s，31 s，42 s 頃に機体を指で軽く押されて外乱を加えられたという，一連の動作が観測されている．そして，外乱を加えられるたびに過渡現象が発生しているが，2 番目の 31 s 付近のものから時定数を推定したところ，0.79 s であった．このことから，POSIX のコマンドでは用意されていない 1 秒未満のより精密なタイミング管理を目指して作成したコマンドが，実際に 1 秒未満の時定数を持つ系の制御に応用できることが示された．

なお，当初作成した PID 制御用のコマンドは，時刻参照をコマンド自身が内部で実施していた．しかし，正確な時刻を得るためには値の入力・計算・出力を非同期にするなどでプログラムが複雑になることや，PID 制御のシミュレーションが実時間でしかできないなどといった問題があり，外部の `linets` コマンドに任せる形で作成し直した．これは，単機能の組み合わせで課題を解決するのがよいと説く UNIX 哲学の効果の顕れと言える．

5.2. gnuplot を用いた心電図情報の動的表示

著者の一人（大野）は，医療用「ECG マルチパラメーターモニターモジュール PM6750」[16]（以下，本項では ECG センサ）から受け取った生体情報を複数の関係者とクラウドを介して共有する試みを続けている．

この ECG センサは，PC とシリアルインタフェースを介して接続でき，PC 上の専用ソフトウェアは ECG センサから心臓の電位情報（これを図示したものが心電図，ECG（Electrocardiogram））, 心拍数，血圧などをリアルタイムで取得し直ちにグラフとして描画できる．このソフトウェアは掃引（スイープ）時間が 5 秒以上のオシロスコープを模しており，表示されるグラフの波形はスイープの進捗に合わせて変化する．

この専用ソフトウェアが表示するグラフの見栄えは優れているが表示中のデータをクラウドに転送する機能はない．しかし，ECG センサから PC に送られるデータは無手順のバイナリデータであり，さらにバイナリのデータフォーマットは販売元から入手可能で，既存のコマンドで形成できるほど単純な形式であった．そこで (1)ECG センサから送られてくるバイナリデータを (2)`cu` コマンド[*4]で受け取り，(3)`awk` での処理に配慮して各バイトを 10 進数表記のアスキー文字列に加工した上で，(4) これを集めてアスキー形式のデータフレームにし，(5) さらに 1 フレーム/行の形にした上で，(6) 行頭にタイムスタンプをつけて，(7) クラウド上の MQTT ブローカに publish（以下「出版」）した．このデータを MQTT ブローカから subscribe（以下「購読」）するとデータは以下の形式になる．

1 秒間に購読できるデータ数は生体情報の種類によって異なり，心電図情報の場合は毎秒 25 回前後，心拍数などのデータの場合は毎秒 1 回前後である．著者らは，掃引時間を 5 秒ないし 10 秒に設定したオシロスコープ

[*3] 現在の Raspberry Pi OS.

[*4] もともとはシリアルポートを介して他のシステムと直接あるいはモデム経由で接続するためのコマンドだが，ここではシェルの標準入出力とシリアルポートを相互につなぐ目的で利用している．

```
タイムスタンプ, 生体データの種類, データの数, データ 1, データ 2,... データ N, チェックサム
```

図 6. タイムスタンプを追加したデータフレーム

を gnuplot[17] を用いて再現し，購読した心電図データの動的表示を試みた．グラフを表示する方法はいくつもあるが，今回は POSIX 環境で確実に利用でき，著者らが長年使い慣れているという理由で gnuplot を選択した．

gnuplot は，与えられたデータを少ない手順でグラフにできるが，オシロスコープのように素早く変化するデータを動的に表示する機能はない．しかし，標準入力からのデータ読み込みとグラフ描画を 0.1 秒程度ごとに繰り返すようにすればデータを動的に表示しているように見える．そこで，MQTT からデータを購読してリングバッファに溜め込む部分と，このスクリプトからデータを読み出してはタイムスタンプを参照してプロットを行う部分からなる図 7 のようなスクリプトを用意した．

一つ目のスクリプトでは MQTT から購読したデータを一つのファイルに蓄積し，この例では 5 秒ごとに一つ目のファイルに蓄積されたデータを二つ目のファイルに移している（実際には `mv` している）．二つ目のスクリプトでは最も新しいデータを基準に過去 5 秒間のデータを `dg3` コマンドに渡している．`dg3` コマンドは内部で gnuplot を `popen()` を使って呼び出し，標準入力から読み込んだひとまとまりのデータを使ってグラフを描く．

ここでは ECG センサから送らたデータを直ちにグラフに描いたが，タイムスタンプが付与されているので，データが到着したタイミングを正確に再現した描画がいつでもできる．

6. おわりに

UNIX はパイプラインの仕組みを持っており，それはコマンドやシェルスクリプトから簡単に利用できる．一方，自動制御や信号処理の分野では，しばしば各種フィルタをカスケーディングさせて実装するため，両者の相性は本質的に良いと考える．しかし，POSIX で定められいる必要最低限の UNIX コマンドはもとより，GNU core utilities など広く普及している UNIX コマンドまで見渡しても，シェルスクリプトベースで IoT 機器制御を実現するためのコマンドは充実してるとは言えず，先行事例も少ない．その主な原因の一つはタイミング管理

```
# === (1) データを蓄積する側 ===
mosquitto_sub -h HOSTNAME -t ECG/demo     |
linets -3e                                |
ptw awk '{print int($1/5)%2,$0;}'         |
awk '                                     #
 BEGIN{n=0}                               #
 n!=$1{system("mv -f 1 0");n=$1}          #
 {                                        #
  print substr($0,index($0," ")+1) >> "1" #
  close("1")                              #
 }'

# === (2) データを取り出す側 ===
yes | valve -l 200ms | while read dummy; do
  latest_ts=$(cat 0 1                |
              sed -n '${s/ .*//;p;}')
  cat 0 1                            |
  awk -v t=$latest_ts '$1>=(t-5)' |
  dg3
done
```

図 7. gnuplot で直近 5 秒間のデータを繰り返し描くシェルスクリプト

のためのコマンドが不十分であるからだと考え，いくつかのコマンドを作成し，シェルスクリプトベースのプログラミングにより倒立振子を立たせることに成功するなど，いくつかの事例において作成したコマンドの有用性が示された．

2021 年現在，IoT 機器の開発では Python などの高級言語が多用されているが，それらが製品・開発基盤の寿命の長さなど，業界の要求に応えられるのか疑問である．もし廃れてしまえば資産は失われて開発の大部分がやり直しになり，あるいは廃れなくても後方互換性の無い仕様変更がなされれば更新を迫られ，システム障害のリスクが増してしまう．UNIX 機を用いたシェルスクリプトベースの開発はそういった問題を避けるための良策である．また，UNIX 哲学にも即している．macOS や WSL の登場などが示唆しているように，UNIX 的アプローチや実装は今，多くの開発現場で支持されている．

著者らは，IoT 機器の開発においても UNIX 哲学的アプローチを積極的に採用すべきであると考える．その際，基本的な UNIX コマンドで不十分であると感じるので

あれば，高級言語プラットフォーム導入の前に，POSIX に即したコマンドの開発を検討すべきである．

謝辞

この論文は，研究活動を支えてくださった多くの方々のご厚意・ご協力によって完成に至ったものである．まずは，同じ研究チーム「NCNT プロジェクト」の北嶋完基氏と矢嶋遼氏，そしてチームの良きアドバイザーとしてご助言くださった田口淳一氏に感謝を申し上げる．さらに，研究内容をご理解くださって色々な面でご協力くださった USP 研究所の皆様，並びに金沢大学の皆様にも御礼を申し上げる．最後に，私達が作成したコマンドに GitHub 上で意見をくれた秘密結社シェルショッカー構成員のミツイ氏，そして POSIX 中心主義アプローチを支持してくれるすべての皆様に感謝する．

参考文献

[1] 株式会社マクニカ, “産業用ネットワークの種類･シェアの最新情報【RS485 はもう古い？】”, https://emb.macnica.co.jp/articles/4277/, 2021-03-15 閲覧.

[2] 松浦智之, 大野浩之, 當仲寛哲, “ソフトウェアの高い互換性と長い持続性を目指す POSIX 中心主義プログラミング”, **デジタルプラクティス**, vol. 8, no. 4, pp. 352–360, 2017.

[3] BlackBerry Limited, “BlackBerry QNX”, https://qnx.com/, 2021-03-14 閲覧.

[4] Yodaiken, V. “The RTLinux Manifesto”, *Proceedings of the 5th Linux Expo*, 1999.

[5] 石綿陽一, 松井俊浩, 国吉康夫, “高度な実時間処理機能を持つ Linux の開発”, **日本ロボット学会学術講演会予稿集**, vol. 16, no. 1, pp. 355–356, 1998.

[6] openSUSE ドキュメンテーション, “openSUSE 13.1: 第 14 章 タスクスケジューラのチューニング”, https://manual.geeko.jp/ja/cha.tuning.taskscheduler.html, 2021-03-15 閲覧.

[7] The IEEE and The Open Group, “The Open Group Base Specifications Issue 7, 2018 Edition, IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)”, https://pubs.opengroup.org/onlinepubs/9699919799/, 2021-03-13 閲覧.

[8] 大野浩之, 松浦智之, 森祥寛, “ものグラミング 2 - 諸機能の選択と集中を徹底した POSIX 中心主義に基づく IoT 開発方式の提案”, **研究報告インターネットと運用技術（*IOT*）**, vol. 2019-IOT-46, no. 16, pp. 1–8, 2019.

[9] Free Software Foundation, coreutilitles/stdbuf.c, https://github.com/coreutils/coreutils/blob/master/src/stdbuf.c, 2021-03-13 閲覧.

[10] Richard, W. S., and S. A. Rago “Advanced Programming in the UNIX Environment Third Edition”, *Addison-Wesley Professional*, 2013.

[11] USP NCNT プロジェクト, “Timing Management Commands in POSIX”, https://github.com/NCNT/TimingCmds_in_POSIX, 2021-03-13 閲覧.

[12] Joey Hess, “moreutils”, https://joeyh.name/code/moreutils/, 2021-05-17 閲覧.

[13] USP NCNT プロジェクト, “標準入出力上のデータの流れるタイミングを記録・再生する方法”, https://growi.ncnt.work/NCNT/info/linets_and_tscat.ja, 2021-03-13 閲覧.

[14] Andrew Wood, “Pipe Viewer”, http://www.ivarch.com/programs/pv.shtml, 2021-05-17 閲覧.

[15] 柳戸新一, 松浦智之, 鈴木裕信, 大野浩之, “シェルスクリプトを用いた UNIX 哲学に基づくリアルタイム制御”, **ソフトウェア・シンポジウム *2021*（投稿中）**, 2021.

[16] Shanghai Berry Electronic Tech Co.,Ltd, “ECG マルチパラメーターモニターモジュール PM6750”, https://www.medicalexpo.com/ja/prod/shanghai-berry-electronic-tech-co-ltd/product-122578-866840.html, 2021-03-15 閲覧.

[17] Williams, Thomas, et al. “gnuplot homepage”, http://www.gnuplot.info/, 2021-03-15 閲覧.