```
url: "https://seimei.do/ja/blog/windows-on-arm-rcs380-driver/"
title: "x64 ドライバーしかないデバイスを Windows on ARM で動かす — RC-S380 用 PC/SC ドライバー自作記"
description: "Windows on ARM は x64 アプリを動かせるが、ドライバーはエミュレーションの対象外である。ベン
  ↪ダーが ARM64 ドライバーを出していない SONY PaSoRi RC-S380 を、UMDF2 の仮想リーダー + Python リレー
  ↪の二段構成で PC/SC リーダーとして動かした。WDK の MSBuild 統合が使えない問題の迂回、
  ↪SCARD_ATTR_CHANNEL_ID という文書化されていない必須属性、キャッシュが成功を偽装する障害など、公開情報
  ↪がほとんどない領域の実務記録。ソースは MIT で公開している。"
pubDate: 2026-08-10
tags: ["windows", "arm64", "driver", "nfc"]
series: ""
related: ["vmware-fusion-windows-on-arm-devenv"]
graphLabel: "RC-S380 WoA ドライバー"
graphWeight: 1
lang: ja
draft: false
ogImage: "/blog/windows-on-arm-rcs380-driver/og.jpg"
```

https://seimei.do/ja/blog/windows-on-arm-rcs380-driver/

特許の本人出願を Mac で進めるにあたり、まず VMware を導入した。

出願にはカードリーダーが必要なようで、素直に買うと高い。

ヤフオクで大量に出品されていて安かったので、非接触 IC カードリーダーの SONY PaSoRi RC-S380 を即決 895 円で買った。説明書なし・箱なし・ケーブル付き。

ところが、手元の Mac 上の Windows は Windows on ARM であり、ドライバーが対応していないため動かなかった。そんな理由でハードを買い足したくないから、ドライバーを自作した。始める前に Claude Code は開発期間を 4 週間と見積もったが、1 日もかからずに完成した。

それがこれ:

bakemocho/rcs380-windows-arm64-driver — SONY PaSoRi RC-S380 を Windows 11 ARM64 で PC/SC リーダーとして動かす自作ドライバー。UMDF2 仮想リーダー + Python リレー構成。

AI は自らの能力を過小評価しているきらいがあるよね。人間から学んでいるからだろうか？

お金がかかるからインストーラーは用意してない。AI に頼めば導入してくれると思います。

ARM64 版 Windows は対象外だと、買ってから気づいた。選択肢は 3 つあった。現行の後継機 RC-S300 を買い足す（ヤフオク相場はだいたい 3,500 円）。ネットカフェの Windows マシンで済ませる。あるいは自分でドライバーを書く。いちばん面白そうな 3 つ目を選んだ。

事実、今回の試行錯誤から Windows ドライバーを開発するノウハウを学べた。NT 技研工業さんから頂いたデータロガーの修理案件に知見を流用できそうで、早速役立っている。いちばん馬鹿らしい選択かと思いきや、ベストな選択だった。

ついでに、なぜ S380 がヤフオクでこんなに安いのかを推理しておく。

図 1　届いた SONY PaSoRi RC-S380 本体（黒・NFC ロゴ入り）と付属の USB ケーブル。箱も説明書もなく、本体とケーブルだけ

- 生産終了品で、新品も値崩れしている（価格.com の新品最安は 1,000 円台前半）
- 定番機として大量に出回ったぶん、中古の玉数が多い
- 公式ソフトの対応環境が狭く、環境が変わった人から手放されやすい
- 現行の RC-S300 があるので、いまさら旧機を選ぶ人が少ない

要するに「使える人には使えるのに、値段がつかない」状態になっている。

結果として S300 の買い足しは回避できた。ドライバー開発に費やした AI のトークン代は、サブスクとはいえ浮いた S300 代と同じくらいかかったかもしれない。それでも、この経験と蓄積は次のドライバー開発にそのまま効くし、何より楽しかった。安く転がっている S380 の価値を少し上げられたかもしれない。

特許も 1 件、代理人を立てずに自力で出願した。残り 4 件も準備している。内容はお楽しみに。

技術に興味のある方はこちら

Windows on ARM は x86/x64 アプリケーションをエミュレーションで実行できるが、**カーネルモードドライバーと UMDF ドライバーはエミュレーションの対象外**で、ARM64 ネイティブでなければロードされな

い。アプリが動くのと同じ感覚で周辺機器も使えるとは限らず、可否はベンダーが ARM64 版ドライバーを出しているかどうかで決まる。

出ていないデバイスは、デバイスマネージャーの「ほかのデバイス」にコード 28（ドライバー未インストール）で残り続ける。Windows Update の自動検索も一致するドライバーを返さない。ベンダーが出していない以上、待っていても解決しない。

手元でこの状態になったのが、非接触 IC カードリーダーの SONY PaSoRi RC-S380 だった。Windows 用のベンダードライバーは x64 専用。そこで、ARM64 ネイティブの PC/SC ドライバーを自作した。到達点から先に書く。

- Windows のスマートカードスタックに仮想リーダーとして登録され、`SCardTransmit` でカードと APDU を往復できる（レスポンス `9000`）
- OS 再起動・USB 切断のどちらからも、手動介入なしで復帰する
- ソース一式を MIT ライセンスで公開している: rcs380-windows-arm64-driver

この記事はその過程の記録である。ARM64 での UMDF2 ドライバービルドは公開情報が少なく、要所でつまずいた点はどれも検索でほぼ出てこなかったため、対処と併せて書き残す。

「USB で通信できる」と「PC/SC で使える」の間には段差がある

最初に潰しておくべき誤解がある。Linux の pcsc-lite では、libusb 経由のユーザー空間ドライバーがそのまま PC/SC リーダーとして登録できる。Windows では同じ方法が通用しない。

実際、開発の最初の段階（Stage 0）では、inbox の WinUSB を参照する自作 INF と自己署名テスト証明書で RC-S380 を WinUSB に束縛し、nfcpy から直接叩けるところまで進んだ。

USB パススルーから Port-100 プロトコル、ISO 14443 Type B、ISO-DEP の APDU 往復まで通る。この時点でカードとの通信自体は成立している。

それでも、PC/SC を要求するアプリケーションから見るとリーダーは存在しないままである。Windows の PC/SC（SCardSvr / リソースマネージャー）は、`Class=SmartCardReader` のカーネルモードまたは UMDF ドライバーしかリーダーとして認識しない。

ユーザー空間から USB で通信できるようになっても、カードとやり取りできるようになっただけ。スマートカードスタックへの入口はまだ開いていない。

では UMDF でスマートカードリーダーを書けるのか。書ける。Microsoft の公式ドキュメントに、ベンダー独自 UMDF リーダードライバーの INF 要件が明記されている（Installing Smart Card Reader Drivers）。要件は 2 点だけである。

- `Class=SmartCardReader`
- `UmdfKernelModeClientPolicy=AllowKernelModeClients`

カーネルモード用ライブラリの smclib は必須でない。Microsoft 自身の inbox CCID ドライバーも UMDF2 で書かれている。

「スマートカードリーダー = カーネルモード必須」という思い込みは、着手前に一次資料で潰せる。

層構成: ドライバーは仮想リーダーに徹する

採った構成は二段構えである。

```
Windows smart card stack (SCardSvr)
  → 自作 UMDF2 ドライバー (ARM64 ネイティブ)   ← Windows が自動起動
  → TCP 127.0.0.1:35963                          ← ドライバーが listen
  → Python リレー                                ← 手動起動
  → nfcpy
  → RC-S380
  → カード
```

ドライバーは仮想リーダーに徹し、実際の無線処理はユーザー空間の Python に任せる。理由は 3 つある。

1. RC-S380 の Port-100 プロトコルは nfcpy が完全実装しており、ドライバー側で書き直す価値がない
2. nfcpy は EUPL-1.1、ドライバー側は MIT にしたかった。TCP 越しの別プログラムに分ければ、同一著作物の中でライセンスが混ざらない
3. デバッグしやすい。リレー側は Python なので、ドライバーを一切ビルドせずにフェイクの相手と単体テストできる

層間のプロトコルは、vsmartcard プロジェクトの vpcd wire protocol をそのまま使った。TCP 上で 2 バイトビッグエンディアンの長さを前置するだけの素朴なプロトコルで、既存仕様に乗ったことで、リレー側は仕様調査なしで書けた。

ただし、このプロトコルには自明でない点が 2 つある。まず、イニシエータが層で逆転する。TCP 接続の確立はリレー側が connect し（ドライバーはサーバーとして listen/accept）、接続確立後の各交換はドライバー側が送ってから受ける。次に、返信の非対称性がある。

制御コード	意味	返信
0x00	powerOff	返してはいけない
0x01	powerOn	返してはいけない
0x02	reset	返してはいけない
0x04	getATR	必須
(それ以外)	APDU	必須

「返してはいけない」側に返信すると、ストリームが 1 メッセージ分ずれる。厄介なのは症状の出方で、その場では壊れず、後のどこかで交換が噛み合わなくなる形でしか観測できない。この表を先に把握しているかどうかで、デバッグの手間が大きく変わる。

ARM64 ビルドの実務: WDK の MSBuild 統合は Build Tools に入らない

WDK 10.0.26100.1 以降は ARM64 ネイティブでの開発を正式サポートしている (Building ARM64 Drivers)。ただし、実際にビルド環境を組むと最初に大きな壁がある。

WDK の Visual Studio 統合は VSIX で提供され、**VSIX は完全版 Visual Studio 専用**である。Visual Studio Build Tools には入らない。つまり `.vcxproj` を用意しても、Build Tools 環境の `msbuild` ではドライバーをビルドできない。

迂回路は UMDF2 の性質そのものにある。UMDF2 ドライバーは実質ユーザーモード DLL なので、MSBuild 統合を諦めて `cl.exe` / `link.exe` を直接叩くビルドスクリプトを書けば通る。

公開リポジトリの `driver/build-direct.ps1` がそれである。KMDF ではこの手は使えない。

結果的には、完全版 Visual Studio への依存が消えてビルドの再現性が上がった。

そこから先も細かい壁が続く。

現象	対処
`vs_buildtools` の `--log` オプションが引数エラーで exit 87	オプションを除いて再実行
`WdfDriverStubUm.lib` が `DbgPrintEx` を要求してリンクエラー	`ntdll.lib` をリンク
`initguid.h` が `winioctl.h` の GUID 定義と衝突 (C2374)	GUID をローカル定義して回避
WDF ヘッダが `/W4` で C4324 を出す	当該警告を抑制
文字コード関連のコンパイルエラー	`/utf-8` を付ける

一つひとつは小さいが、連続して踏むと経路自体が間違っているのかと疑いたくなる。実際には全部ツールチェーンの既知の性質で、ひとつずつ潰せば通る。

もう 1 つ、ARM64 マシンでビルドしていても、ツールチェーンの選択を誤ると x64 バイナリが出る。これを排除するため 2 箇所を確認した。

- コンパイラ: `bin\Hostarm64\arm64\cl.exe` が実在すること
- 出力: `dumpbin` が `AA64 machine (ARM64)` を返すこと

環境は Windows 11 24H2 (ARM64) / MSVC 14.44 / Windows SDK 10.0.26100.0 / WDK 10.0.26100.1。ツールチェーン全体で約 12GB だった。

なぜインストーラーを配らないのか

署名の現状も書いておく。正規の配布署名（Partner Center のアテステーション署名）は取っておらず、自己署名テスト証明書 + testsigning 有効で動かしている。`New-FileCatalog` でカタログを作って署名し、`pnputil` でパッケージを登録、`devcon` でルート列挙デバイスを作る。

この構成では、他人がこのドライバーを使うには Secure Boot を無効化して testsigning を有効にする必要がある。公開したのがインストーラーではなくソースなのはこのためで、ビルドできる人には使えるが、誰でもすぐ導入できるものではない。

踏んだ地雷たち

開発中に踏んだ問題のうち、一般化できるものを挙げる。共通するのは、どれも症状が真因と別の場所を指すことである。

地雷 1: `SCARD_ATTR_CHANNEL_ID` を実装しないとリーダー登録自体を拒否される

最重要の 1 件。現象は、リーダーが `SCardListReaders` に正しい名前で列挙されるのに、`SCardConnect` がすべて `SCARD_E_UNKNOWN_READER` で失敗するというものである。

「列挙されるのに unknown reader」という症状は、リーダー名の綴りや大文字小文字の不一致を疑わせるが、名前をいくら確認しても解決しない。真因は `SCARD_ATTR_CHANNEL_ID`（`0x00020110`）という属性を実装していないこと、その 1 点だった。

しくみはこうだ。PC/SC のリソースマネージャーは、リーダー追加時にこの属性を問い合わせる。`STATUS_NOT_SUPPORTED` を返すと、System イベント 610 を出して**リーダーの登録自体を拒否する**。

それでも名前は reader database に残るため、`SCardListReaders` には出続ける。登録に失敗しているのに、列挙には出る。この誤誘導が、症状を「名前の問題」に見せかける。

返すべき値は次のとおり。

- 属性の定義: `SCARD_ATTR_VALUE(SCARD_CLASS_COMMUNICATIONS = 2, 0x0110)`
- 値のエンコード: `0xDDDDCCCC`（上位ワード = 接続方式、下位ワード= チャネル番号）
- NFC は接続方式 `0x0100`。チャネル 0 なら返す値は `0x01000000`

この属性はリーダー登録の必須条件でありながら、ドキュメントで「必須」と明示されていない。しかもイベントログはこの問い合わせを生バイト列でしか出さないため、ログを眺めていても拒否には気づけない。

特定の決め手は自作のファイルログ（`driver/log.c`）だった。リソースマネージャーが何をどの順で問い合わせ、こちらが何を返したか。それを自分のログで並べて初めて、拒否の瞬間を特定できた。

同じ属性は vsmartcard の Windows ドライバーでも未実装だったため、#324 として報告した。

おもしろいのは経緯で、メンテナー自身が以前この属性の同定までは正しく到達し、「実装は不要のはず」と判断していた。同定は合っていて、実装が必要だった。文書化されていない必須属性は、こういうすれ違いを生む。

地雷 2: 診断ツール自身のバグが同じ症状を作っていた

検証用の PowerShell スクリプトに、1 要素のパイプライン結果がスカラー文字列に落ちるという言語仕様由来のバグが混入していた。結果、`$readers[0]` がリーダー名ではなく先頭 1 文字を返し、「リーダー名 `'r'` は不明」と報告していた。

つまり一時期、地雷 1 の真因とツールのバグが同じ症状を作っていた。ドライバーのバグを追っているつもりで、実際には半分は自分の診断ツールを追っていたことになる。診断ツール自身を疑う手順を持たない診断は、こういう形で行き詰まる。

地雷 3: キャッシュが成功を偽装する

v0.3 で見つかった現象。USB を物理的に抜いても、PC/SC 層は `PRESENT` と古い ATR を返し続け、`SCardConnect` だけが失敗する。

最初の記録には「ドライバーの `recv` がタイムアウトなしでハングし、ロックを保持し続けるため」と書いた。もっともらしい説明だが、これは未検証の仮説で、実際には間違っていた。訂正まで含めて書いておく。

真因はリレー側にあった。一度センスしたタグをキャッシュし、以後はハードウェアへ一切触れないまま ATR を返し続けていたのである。USB が切れてもキャッシュは残るので `atr()` は成功し続け、ドライバーは「カードあり」と判定し続ける。ドライバーはハングしておらず、誤った応答を受け取り続けていただけだった。

修正は 2 箇所に入れ、役割を分けた。

リレー側は毎ポーリングで `tag.is_present` により実機の生存を確認し、USB の I/O エラー時はフロントエンドを閉じて 2 秒間隔で再オープンする。これが素早い検知と復帰の本命である。

ドライバー側は受信に 30 秒のタイムアウトを入れた。こちらは永久ハングだけは絶対に起こさない最後の砦で、通常運用では一切発火しないことも実測で確認した。

タイムアウト値の選び方は設計判断だった。5 秒程度に詰めると、正当に時間のかかる APDU を誤って切断するという新しい障害を作り込む。素早い検知はリレーが担うと決めたから、ドライバー側は長めでよい。障害検知をどの層に置くかを決めると、各層の数値は自然に決まる。

地雷 4・5: `oemNN` 番号とロケール依存の出力

スクリプトや手順書を書くときに踏みやすい 2 件。

`pnputil` で登録したドライバーパッケージには `oem9.inf`, `oem10.inf` …と番号が振られるが、この番号は更新のたびに変わる。手順書やスクリプトに焼き込むと、次の更新で静かに別のパッケージを指す。

確認は `DriverVer` で行うべきで、`oemNN` は識別子として使えない。なお古いパッケージを消す必要はなく、PnP は正しく最新版を選択する（実測確認済み）。

もう 1 つ、`pnputil` の出力は表示言語に依存する。日本語環境の出力文字列を前提にパースするスクリプトは、英語環境で静かに壊れる。同じ理由で、イベントログの日本語文面をそのまま公開文書に貼るのも避け、英語の実文字列を使うようにした。

地雷 6・7: upstream に報告した 2 件と、バグが存在できない構造

UMDF2 対応の作業中に、vsmartcard 側のコードで 2 つの問題を見つけて報告した。

1 つは完了ステータスの型混在（#335）。UMDF1 は `HRESULT` で完了し、UMDF2 は `NTSTATUS` で完了する。既存コードは `STATUS_NO_MEDIA` を `HRESULT` の文脈で使うなど両者を混在させており、UMDF2 化では全完了箇所の正規化が要る。作業自体は機械的だが、漏らすと SCardSvr が誤動作する。

もう 1 つはキャンセル処理のイテレータ無効参照（#334）。`vector` から `erase` した後もイテレータを使い続けていた。

自作ドライバーでは、後者の構造自体を持たないことにした。挿入/抜去待ちのリクエストを自前の `vector` で管理するのをやめ、UMDF2 の manual queue に委ねる（`driver/queue.c`）。手作業の `MarkCancelable` 管理が不要になり、イテレータ無効参照というバグの入る余地そのものが消えた。

バグを直すのではなく、バグが存在できない構造にする。書き直しの機会にだけ選べる選択肢である。

検証: 変更した経路を通らないテストは、検証ではない

新しい版数を「実際に使う構成」に載せる条件は、3 テストすべて合格と決めた。

#	テスト	何を見るか
1	e2e カード読み取り	`SCardTransmit` が `response: 9000` を返す
2	OS 再起動からの自動復帰	ドライバー自動起動 / 有効版数が旧版に落ちていない / testsigning 維持 / USB 割り当て維持
3	USB 切断からの復旧	切断中の状態更新と、手動介入ゼロの自動復帰の両方

3 つに増やした経緯も書いておく。当初は「9000 が返ることを再確認する」だけを昇格条件にしようとしていた。しかしある版の変更点は切断・再オープン経路であり、正常系の 9000 テストはその経路を一切通らない。

実際その版の検証では、Test 1 が合格した後、Test 3 で初めて本命の挙動を確認できた。Test 1 だけで昇格させていたら、直したはずの経路が直っていない可能性を抱えたまま常用構成に載せていたことになる。

一般化すると、正常系の疎通確認を昇格条件にすると、変更の中身を一度も試さずに昇格が通ってしまう。変更した経路そのものを試していない検証は、検証ではない。

Test 2 には副産物もあった。再起動リハーサル中、リレー起動前の時点で既に TCP ポートが LISTENING になっていることが観測できた。ドライバーはブート時に自動ロードされ、vpcd トランスポートの TCP サーバーまで自力で立ち上がっている。

つまり再起動後に必要な手動操作はリレー起動の 1 コマンドだけ、と確定し、手順書からドライバー起動の記述を削れた。「復帰するか」を確かめに行って「どこまで自動で復帰するか」が分かる。検証は仕様の発見でもある。

v0.3 → v0.4 の差は Test 3 で確認できた。

	v0.3	v0.4
切断中の PC/SC 状態	stale な `PRESENT` のまま	`EMPTY | CHANGED` へ即時更新
検知	しない	1 ポーリング周期
復帰	リレーの手動再起動が必要	再接続だけで `9000` まで完全自動
ログ	トレースバックが延々	`reader not available at usb:054c:06c1 ([Errno 19] No such device)` の 1 行

延々と出るトレースバックは、次に何かが起きたとき真因を埋めてしまう。ログの質も検証項目に入れてよい。

最後に、検証ツール側の落とし穴を 2 つ。

- 検証スクリプトが終了コードを返しておらず、失敗しても呼び出し側が成功と見なしうる状態が後から見つかった
- `pnputil` 出力のロケール依存パース（地雷 5）のせいで、検証したはずの版数と実際に有効な版数がずれる寸前だった

テストが通ったという報告は、テストが実際に走ったことを前提にしている。検証ツールは、検証対象と同じ厳しさで検証しなければならない。

まとめ

公開リポジトリ rcs380-windows-arm64-driver には、UMDF2 ドライバー（`driver/`）、Python リレー（`relay/`）、検証ツール（`tools/`）の一式が入っている。この記事の各節は、おおよそ次のファイルに対応する。

記事の内容	リポジトリの該当箇所
層構成と vpcd プロトコル	`driver/vpcd.c`, `relay/vpcd_protocol.py`
ビルドの壁と迂回	`driver/build-direct.ps1`

記事の内容	リポジトリの該当箇所
SCARD_ATTR_CHANNEL_ID	driver/reader.c, driver/log.c
キャッシュされた成功	relay/card_backend.py
manual queue による構造的回避	driver/queue.c
3 テストの検証	tools/pcsc-e2e.ps1

Windows on ARM でコード 28 のまま止まっているデバイスも、UMDF2 で書ける種類のものであれば、ベンダーを待つ以外の道がある。要点は 3 つ。

- WinUSB で通信できることと、PC/SC で使えることは別である
- WDK の MSBuild 統合が使えなくても、cl.exe 直叩きで迂回できる
- SCARD_ATTR_CHANNEL_ID を実装しないと、リーダー登録自体が拒否される

同種の移植を試す際の参考になれば幸いである。