3DS

3DS プログラミングマニュアル

無線通信編

2016-06-24

Version 1.6

目次

1. はじめに 16
2. 無線通信の概要 17
2.1. 無線通信の種類 17
2.2. 無線オンモードと無線オフモード 18
2.3. 無線通信の暗号化の処理負荷 18
2.4. 無線通信モジュールのエラー 18
3. フォアグラウンド通信 20
3.1. ローカル通信(UDS 通信) 20
3.1.1. UDS 通信の内部ステート 21
3.1.2. 初期化 23
3.1.3. 接続 23
3.1.3.1. ローカル通信 ID の生成 23
3.1.3.2. ネットワークの新規構築 24
3.1.3.3. 周囲にあるネットワークの探索 25
3.1.3.4. ネットワークへの接続 26
3.1.3.5. 独自データのビーコンへのセット 26
3.1.3.6. ビーコンにセットされた独自データの取得 27
3.1.4. 通信 27
3.1.4.1. 端点の生成 27
3.1.4.2. 送信 28
3.1.4.3. 受信 30
3.1.4.4. パケット消失の要因 30
3.1.4.5. 通信の効率を高めるには 31
3.1.5. 切断 32
3.1.5.1. Master による Client/Spectator の切断 32
3.1.5.2. Master によるネットワークの破棄 32
3.1.5.3. Client/Spectator による切断 32
3.1.5.4. Master による Client からの接続の制御 33
3.1.5.5. スリープ状態への遷移、無線オフモードへの切り替えによる切断 33
3.1.6. 終了処理 33
3.1.7. ネットワーク状態の同期 33
3.1.8. ノード ID の割り当て 34
3.1.9. 各種情報の取得 34
3.1.9.1. 接続状態の取得 34
3.1.9.2. リンクレベルの取得 35
3.1.9.3. ノード情報の取得 36
3.1.9.4. チャンネルの取得 36
3.2. ニンテンドー3DSダウンロードプレイ 36

3.2.1. 初期化 37
3.2.2. 状態の確認 39
3.2.3. セッションの開始と停止 40
3.2.4. 接続状況の確認 41
3.2.5. 接続の許可と拒否 42
3.2.6. 配信の開始 42
3.2.7. クライアントのリブート 43
3.2.8. 終了処理 44
3.2.9. 子機プログラムについて 44
3.2.9.1. 組み込み方法 45
3.2.9.2. 強制ダウンロード 45
3.2.9.3. デバッグ方法 45
3.2.9.4. 子機プログラムの判定 45
3.2.10. 擬似クライアント 45
3.2.10.1. 初期化 46
3.2.10.2. 状態の確認 47
3.2.10.3. サーバーのスキャン 48
3.2.10.4. スキャン結果の取得 49
3.2.10.5. セッションへの参加 51
3.2.10.6. 副次的な情報の取得 52
3.2.10.7. ダウンロードの終了 52
3.3. 自動接続 53
3.3.1. 初期化 53
3.3.2. 自動接続 53
3.3.3. 接続状態の取得 55
3.3.4. 切断 56
3.3.5. 終了 56
3.4. 信頼性のあるローカル通信(RDT 通信) 57
3.4.1. 初期化処理 57
3.4.1.1. Sender の初期化 58
3.4.1.2. Receiver の初期化 58
3.4.2. 状態の取得 59
3.4.3. 通信処理の進行 61
3.4.4. 接続の開始 62
3.4.5. データの送受信 62
3.4.6. 中断処理 63
3.4.7. 接続の終了 63
3.4.8. 終了処理 64
4. バックグラウンド通信 65
4.1. すれちがい通信 66

4.1.1. 動作の概要 66
4.1.1.1. すれちがい通信に関する諸条件 67
4.1.2. 初期化と終了 68
4.1.3. すれちがいボックス 69
4.1.3.1. アクセスの開始（オープンと作成） 69
4.1.3.2. 付随データの設定 71
4.1.3.3. 送信ボックス / 受信ボックスの情報 72
4.1.3.4. 削除 72
4.1.3.5. アクセスの終了（クローズとコミット） 73
4.1.3.6. すれちがいデーモンとの関係 73
4.1.3.7. すれちがい通信専用モード（デバッグ用途のみ） 73
4.1.4. すれちがいデータ 74
4.1.4.1. 新規作成 74
4.1.4.2. 送信ボックスへの登録 77
4.1.4.3. 情報の取得 79
4.1.4.4. 削除 79
4.1.4.5. 情報の更新 80
4.1.5. ヘッダ情報へのアクセス 80
4.1.6. 通信発生の通知 81
4.1.7. すれちがい通信での送受信の例 82
4.1.8. 効果的な設定 83
4.1.8.1. 推奨設定 84
4.1.8.2. 複数のデータを登録する 84
4.1.8.3. 送信対象の設定を利用する 85
4.1.8.4. 伝播可能回数の設定を利用する 85
4.2. いつの間に通信 85
4.2.1. すべてのタスクで共通する処理 86
4.2.1.1. BOSS ライブラリの初期化 87
4.2.1.2. タスクのプロパティ設定 87
4.2.1.3. タスクの動作設定 91
4.2.1.4. タスクの登録と実行 93
4.2.1.5. タスクの情報 95
4.2.1.6. タスク一覧の取得 98
4.2.1.7. タスクの変更 98
4.2.1.8. タスクの登録解除 99
4.2.1.9. BOSS ライブラリの終了 99
4.2.1.10. BOSS ライブラリのエラーハンドリング 99
4.2.2. Nintendo アーカイブダウンロードタスク（NADL タスク） 101
4.2.2.1. NS アーカイブ、NS データ 102
4.2.2.2. BOSS ストレージ 104

4.2.2.3. NADL タスクの処理に必要な手順 106
4.2.2.4. BOSS 利用のための準備 108
4.2.2.5. NADL タスクの登録 110
4.2.2.6. 新着フラグのチェック、データ新着イベントによる待ち受け 111
4.2.2.7. ダウンロードデータのチェック 112
4.2.2.8. ダウンロードデータの読み込み 114
4.2.3. NSA リスト 117
4.2.3.1. NsaList クラス 117
4.2.3.2. NSA リストの取得 117
4.2.3.3. 取得結果のチェック 119
4.2.3.4. 正当性と更新のチェック 119
4.2.3.5. NSA リストの解析 119
4.2.4. データアップロードタスク 121
4.2.5. DataStore アップロードタスク 121
4.2.6. DataStore ダウンロードタスク 122
4.3. プレゼンス機能 123
4.3.1. 概要 123
4.3.2. ユーザーアカウント 123
4.3.2.1. ユーザー ID の種類 123
4.3.2.2. アカウント内情報 124
4.3.3. フレンドリストの管理 124
4.3.3.1. フレンドリストへの登録 125
4.3.3.2. フレンド関係の状態 125
4.3.4. 初期化と終了 126
4.3.5. オンラインとオフライン 126
4.3.5.1. 自律接続とログイン 126
4.3.5.2. フレンドサーバーやフレンドとの情報の同期 128
4.3.5.3. 非公開モード 129
4.3.5.4. 自分の情報の取得 129
4.3.6. フレンドの情報の取得 130
4.3.6.1. フレンドキーの取得 130
4.3.6.2. フレンドの情報の取得 130
4.3.7. 通知 131
4.3.7.1. 通知の種類 131
4.3.7.2. 通知に関する API 132
4.3.8. エラーハンドリング 133
4.3.9. フレンド登録 133
5. 通信補助ライブラリ 135
5.1. 受信拒否リスト 135
5.1.1. 概要 135

5.1.2. 初期化と終了 135
5.1.3. ローカル受信拒否リスト 136
5.1.4. 受信拒否リスト該当チェック 136
5.1.5. UGC 閲覧モード 137
5.1.6. ローカル受信拒否リスト登録の流れ 137
5.2. NG ワードリスト 137
5.2.1. 初期化と終了 138
5.2.2. NG ワードのチェック 138
5.2.3. 数字のチェック 140
5.2.4. 文章のチェック 140
5.3. おしらせ 141
5.3.1. 初期化と終了 141
5.3.2. おしらせの投稿 141
5.3.3. URL 付おしらせの投稿 142
5.4. アカウント 143
5.4.1. 初期化と終了 143
5.4.2. ニンテンドーネットワークアカウントの情報 143
5.4.2.1. 年齢判定 144
5.4.3. ネットワーク時計 144
5.4.4. ログインアプレット 145
5.4.4.1. 独自サービス向けサービストークン 145
5.4.4.2. Miiverse 向けサービストークン 146
5.4.5. UUID の生成 146
6. デバッグ用ライブラリ 147
6.1. ソケット通信 147
6.1.1. 初期化 147
6.1.2. ソケットの作成 148
6.1.3. アドレスとポート番号の結び付け 149
6.1.4. 動作モード 149
6.1.5. 接続の待ち受け 150
6.1.6. リモートホストへの接続 151
6.1.7. データの受信 153
6.1.8. データの送信 155
6.1.9. オプション設定 156
6.1.10. ソケットの切断 156
6.1.11. ソケットの破棄 157
6.1.12. 終了 158
6.1.13. ユーティリティ関数 158
6.2. SSL 通信 161
6.2.1. 初期化 161

6.2.2. 接続クラスの生成 161
6.2.3. 通信先の設定 161
6.2.4. 証明書と CRL の設定 162
6.2.4.1. 証明書ストアのクラス 162
6.2.4.2. CRL ストアのクラス 163
6.2.4.3. クライアント証明書のクラス 164
6.2.5. ハンドシェイク 164
6.2.6. データの送受信 165
6.2.7. 接続の切断 165
6.2.8. 終了 165
6.3. HTTP 通信 166
6.3.1. 初期化 166
6.3.2. 接続クラスの生成 166
6.3.3. 通信設定 167
6.3.3.1. HTTPS 通信を行う場合の設定 167
6.3.4. 送信データの設定 170
6.3.4.1. POST データ遅延設定モード 170
6.3.5. 接続の開始 172
6.3.6. レスポンスの受信 172
6.3.7. 接続状況などの取得 173
6.3.8. 接続の終了 174
6.3.9. 終了 174
7. 付録:ローカル通信の通信性能 175
7.1. 測定環境および測定結果 175
8. 付録:無線通信環境に関する諸注意 180
8.1. 無線通信環境を用意する際に注意すべきこと 180
8.1.1. ほかの Wi-Fi 機器のチャンネル設定 180
8.1.2. Wi-Fi 機器以外の機器の影響 181
8.1.3. パケットキャプチャ使用時の注意 181
8.2. ほかの機器からの影響を受けない無線通信環境を用意する方法 181
8.2.1. 電波シールドの使用 181
8.2.2. 同軸ケーブルの使用 182
9. 付録:すれちがい通信中継 183
9.1. 通常のすれちがい通信との違い 183
9.1.1. すべてのすれちがいデータが処理の対象となる 183
9.1.2. フレンド向けのデータは送受信されない 183
9.1.3. 交換の相手が異なる 184
9.2. すれちがい通信中継のテスト 184
9.3. BossLotcheckTool を使用したテスト手順 184
9.3.1. Forced Task Start の実行結果 185

更新履歴 187

コード

コード 3-1. UDS ライブラリの初期化 23
コード 3-2. ローカル通信 ID の生成 24
コード 3-3. ネットワークの構築 24
コード 3-4. 既存ネットワークの探索 25
コード 3-5. ネットワークへの接続 26
コード 3-6. 独自データのビーコンへのセット 26
コード 3-7. ビーコンにセットされた独自データの取得 27
コード 3-8. 端点とポート番号、ノード ID の関連付け 27
コード 3-9. データの送信 28
コード 3-10. 最大送信遅延時間の設定 29
コード 3-11. データの受信 30
コード 3-12. Client をネットワークから切断 32
コード 3-13. Spectator をネットワークから切断 32
コード 3-14. ネットワークの破棄 32
コード 3-15. ネットワークからの離脱 33
コード 3-16. Client からの接続の制御 33
コード 3-17. UDS ライブラリの終了処理 33
コード 3-18. 接続状態の取得 34
コード 3-19. nn::uds::ConnectionStatus 構造体の定義 34
コード 3-20. リンクレベルの取得 35
コード 3-21. ノード情報の取得 36
コード 3-22. チャンネルの取得 36
コード 3-23. 初期化 37
コード 3-24. 状態の確認 39
コード 3-25. セッションの開始と停止 40
コード 3-26. 接続状況の確認 41
コード 3-27. 接続の許可と拒否 42
コード 3-28. 配信の開始 43
コード 3-29. クライアントのリブート 43
コード 3-30. 終了処理 44
コード 3-31. 再接続情報の取得 44
コード 3-32. nn::dlp::RebootInfo 構造体の定義 44
コード 3-33. 子機プログラムの判定 45
コード 3-34. 擬似クライアントの初期化 46
コード 3-35. 擬似クライアントの状態の確認 47
コード 3-36. nn::dlp::ClientStatus の定義 48

コード 3-37. サーバーのスキャン 48
コード 3-38. スキャン結果の取得 49
コード 3-39. nn::dlp::TitleInfo の定義 50
コード 3-40. nn::dlp::ServerInfo の定義 50
コード 3-41. セッションへの参加 51
コード 3-42. 副次的な情報の取得 52
コード 3-43. パスフレーズの取得と終了処理 52
コード 3-44. AC ライブラリの初期化 53
コード 3-45. 接続条件の作成 53
コード 3-46. 自動接続 54
コード 3-47. 接続状態の取得 55
コード 3-48. 接続の切断 56
コード 3-49. AC ライブラリの終了 57
コード 3-50. Sender の初期化関数 58
コード 3-51. SenderConfig 構造体 58
コード 3-52. Receiver の初期化関数 59
コード 3-53. ReceiverConfig 構造体 59
コード 3-54. Sender/Receiver の状態の取得 59
コード 3-55. 通信処理の進行 62
コード 3-56. 接続の開始 62
コード 3-57. データの送受信 63
コード 3-58. 中断処理 63
コード 3-59. 接続の終了 63
コード 3-60. 終了処理 64
コード 4-1. 初期化と終了 68
コード 4-2. すれちがいボックスのオープン、作成 69
コード 4-3. 付随データの設定 71
コード 4-4. 送信ボックス / 受信ボックスの情報の取得 72
コード 4-5. インデックス指定によるメッセージ ID の取得 72
コード 4-6. すれちがいデータの取得 72
コード 4-7. すれちがいボックスの削除 73
コード 4-8. すれちがいボックスのクローズ 73
コード 4-9. すれちがいボックスのコミット 73
コード 4-10. すれちがい通信専用モードへの切り替え 73
コード 4-11. すれちがいデータの新規作成 74
コード 4-12. 拡張ヘッダ情報の設定 77
コード 4-13. 送信データの設定 77
コード 4-14. ボックスへのすれちがいデータの保存 78
コード 4-15. 情報の取得 79
コード 4-16. すれちがいデータの削除 80

コード 4-17. すれちがいデータの設定 80
コード 4-18. すれちがいデータへのアクセス 81
コード 4-19. 通信発生の通知を受け取るイベント、受信データの取得 81
コード 4-20. BOSS ライブラリの初期化 87
コード 4-21. nn::boss::Task クラスの初期化 89
コード 4-22. nn::boss::TaskPolicy クラスの初期化 89
コード 4-23. プロパティの設定、取得 90
コード 4-24. タスクの動作の設定関数 91
コード 4-25. 機器内蔵の証明書に関するプロパティの設定関数 92
コード 4-26. 独自証明書の設定 93
コード 4-27. タスクの登録 93
コード 4-28. タスクの実行、中止、完了待ち 93
コード 4-29. 即時実行専用タスクの登録 95
コード 4-30. タスクの情報の取得 95
コード 4-31. タスク一覧の取得 98
コード 4-32. nn::boss::TaskIdList クラス 98
コード 4-33. タスクの変更 98
コード 4-34. nn::boss::Task クラスでのプロパティ設定 99
コード 4-35. タスクの登録解除 99
コード 4-36. BOSS ライブラリの終了 99
コード 4-37. nn::boss::GetNsDataIdList() のエラーハンドリング例 100
コード 4-38. OptOut フラグの設定と取得 104
コード 4-39. BOSS ストレージの登録と登録解除 108
コード 4-40. nn::boss::NsaDownloadAction クラスの初期化 110
コード 4-41. 新着データの確認 111
コード 4-42. NS データのシリアル ID 一覧の取得 112
コード 4-43. nn::boss::NsDataIdList クラス 113
コード 4-44. NS データのシリアル ID 一覧の取得（アプリケーション限定） 114
コード 4-45. nn::boss::NsData クラスの初期化と読み込み関連関数 114
コード 4-46. 付加情報の設定・取得とデータの削除 115
コード 4-47. NsaList クラスのコンストラクタ/デストラクタ 117
コード 4-48. NSA リストを取得する関数 118
コード 4-49. 取得結果のチェックに使用する関数 119
コード 4-50. 正当性と更新のチェックに使用する関数 119
コード 4-51. NSA リストの解析に使用する関数 119
コード 4-52. NsaInformation 構造体の定義 120
コード 4-53. 初期化と終了の API 126
コード 4-54. ログイン API 127
コード 4-55. ログイン処理のサンプルコード 127
コード 4-56. 自分情報の更新 API 128

コード 4-57. 自分の情報を取得する API 129
コード 4-58. フレンドキーの取得 API 130
コード 4-59. ローカルフレンドコードのスクランブルを解除する API 130
コード 4-60. フレンドの情報取得 API 131
コード 4-61. 通知 API 132
コード 4-62. エラーコード取得 API 133
コード 4-63. アプリケーション内でのフレンド登録に使用する関数 133
コード 4-64. フレンド登録用の情報から情報を取得する関数 134
コード 5-1. UBL ライブラリの初期化と終了 136
コード 5-2. ローカル受信拒否リストへの登録 136
コード 5-3. 受信拒否リストのチェック 136
コード 5-4. nn::ngc::ProfanityFilter クラスの初期化 138
コード 5-5. nn::ngc::ProfanityFilter クラスの終了 138
コード 5-6. NG ワードのチェック 139
コード 5-7. 文字列に含まれる数字の数のチェック 140
コード 5-8. 文章内の NG ワードのチェック 140
コード 5-9. NEWS ライブラリの初期化と終了 141
コード 5-10. おしらせの投稿 141
コード 5-11. URL 付おしらせの投稿で使用する関数 142
コード 5-12. 独自サービス向けサービストークンを取得する関数 145
コード 5-13. 独自サービス向けサービストークンを取得する際のエラーハンドリングの例 146
コード 6-1. SOCKET ライブラリの初期化 147
コード 6-2. ソケットの作成 148
コード 6-3. アドレスおよびポート番号の結び付け 149
コード 6-4. 動作モードの設定・取得 150
コード 6-5. 接続の待ち受け 150
コード 6-6. リモートホストへの接続 151
コード 6-7. リモートホストへの接続の確認 152
コード 6-8. データの受信 154
コード 6-9. データの送信 155
コード 6-10. オプション設定 156
コード 6-11. ソケットの切断 157
コード 6-12. ソケットの破棄 157
コード 6-13. SOCKET ライブラリの終了 158
コード 6-14. ソケットアドレスの取得 158
コード 6-15. ネットワークアダプタ情報の取得 159
コード 6-16. DNS を介したリモートホスト情報の取得 159
コード 6-17. アドレス変換 160
コード 6-18. バイトオーダー変換 161
コード 6-19. SSL ライブラリの初期化 161

コード 6-20. 接続クラスのコンストラクタとソケットのアサイン 161
コード 6-21. 通信先の設定 161
コード 6-22. 証明書と CRL の設定 162
コード 6-23. 証明書ストアのクラス 163
コード 6-24. CRL ストアのクラス 163
コード 6-25. クライアント証明書のクラス 164
コード 6-26. ハンドシェイク 164
コード 6-27. SSL 接続経由でのデータの送受信 165
コード 6-28. SSL 接続の切断 165
コード 6-29. SSL ライブラリの終了 165
コード 6-30. HTTP ライブラリの初期化 166
コード 6-31. HTTP 接続クラスのコンストラクタと初期化 166
コード 6-32. 通信設定 167
コード 6-33. HTTPS 通信の設定 167
コード 6-34. nn::http::CertStore クラスの定義 168
コード 6-35. nn::http::CrlStore クラスの定義 169
コード 6-36. nn::http::ClientCert クラスの定義 169
コード 6-37. 送信前に行うヘッダ・データの追加 170
コード 6-38. POST データ遅延設定モードでの送信 171
コード 6-39. 接続の開始とキャンセル 172
コード 6-40. レスポンスの受信 172
コード 6-41. 接続状況などの取得 173
コード 6-42. 接続の終了 174
コード 6-43. HTTP ライブラリの終了 174

表

表 2-1. フォアグラウンド通信に属する無線通信 17
表 2-2. バックグラウンド通信に属する無線通信 17
表 2-3. 無線通信モジュールの状態と無線ランプの状態の対応表 18
表 2-4. 通信モード別 FATAL エラー一覧 19
表 3-1. ステート 22
表 3-2. 中間ステート 22
表 3-3. ステート名と定義名の対応 22
表 3-4. ネットワークへの接続モード 26
表 3-5. 予約済みノード ID 34
表 3-6. 切断理由 35
表 3-7. リンクレベル 35
表 3-8. Initialize() で返される可能性のある返り値 38
表 3-9. サーバーの状態 39

表 3-10. GetEventDesc()、GetState() で返される可能性のある返り値 40
表 3-11. OpenSessions()、CloseSessions() で返される可能性のある返り値 40
表 3-12. クライアントの状態（サーバーから確認できる状態のみ） 41
表 3-13. GetConnectingClients()、GetClientInfo()、GetClientState() で返される可能性のある返り値 42
表 3-14. AcceptClient()、DisconnectClient() で返される可能性のある返り値 42
表 3-15. StartDistribute() で返される可能性のある返り値 43
表 3-16. RebootAllClients() で返される可能性のある返り値 44
表 3-17. Finalize() で返される可能性のある返り値 44
表 3-18. Initialize() で返される可能性のある返り値 47
表 3-19. 擬似クライアントの状態（擬似クライアントで確認できる状態のみ） 48
表 3-20. StartScan() で返される可能性のある返り値 49
表 3-21. StartFakeSession() で返される可能性のある返り値 51
表 3-22. nn::ac::ApType 列挙子 55
表 3-23. nn::ac::LinkLevel 列挙子 56
表 3-24. RDT ライブラリで使用する用語 57
表 3-25. Sender の状態（nn::rdt::SenderState の定義） 60
表 3-26. Receiver の状態（nn::rdt::ReceiverState の定義） 61
表 4-1. 現時点で判明しているデーモンによるアプリへの影響 65
表 4-2. OpenMessageBox() が返す可能性のあるエラーとその対処 70
表 4-3. CreateMessageBox() が返す可能性のあるエラーとその対処 71
表 4-4. 送信対象を設定するフラグ 74
表 4-5. 送受信モード 75
表 4-6. 送受信モードの組み合わせによって行われる送受信 75
表 4-7. 拡張ヘッダの種類 77
表 4-8. NADL タスクを登録する際のプロパティ値の制約 88
表 4-9. nn::boss::TaskPolicy クラスで指定可能なプロパティ識別子 90
表 4-10. nn::boss::TaskAction の派生クラスと対応するタスクの種類 90
表 4-11. HTTP リクエストに付与される AP 情報 91
表 4-12. HTTP リクエストに付与する AP 情報の指定 91
表 4-13. HTTP リクエストに付与する本体設定情報の指定 92
表 4-14. タスクの状態 96
表 4-15. nn::boss::TaskStatus クラスで指定可能なプロパティ識別子 97
表 4-16. nn::boss::TaskError クラスで指定可能なプロパティ識別子 98
表 4-17. NS データの種類 103
表 4-18. NADL タスクの各処理の概要 107
表 4-19. nn::boss::NsaDownloadAction クラスで指定可能なプロパティ識別子 111
表 4-20. NS データの種別 112
表 4-21. ヘッダ種別 115
表 4-22. 属性の指定と取得判定 118
表 4-23. NsaInformation 構造体のメンバに格納されている情報 120

表 4-24. プレゼンス情報の設定と取得 124
表 4-25. フレンドカードに表示することのできる文字 128
表 4-26. 通知の種類 132
表 5-1. パターンリスト 139
表 5-2. おしらせの仕様 142
表 5-3. おしらせの投稿で返されるエラー 142
表 5-4. ニンテンドーネットワークアカウントの情報と取得関数の対応 144
表 6-1. デバッグ用ライブラリ 147
表 6-2. ソケット作成時に発生するエラー 148
表 6-3. アドレス、ポート番号の結び付け時に発生するエラー 149
表 6-4. 動作モードの設定・取得時に発生するエラー 150
表 6-5. 接続の待ち受け時に発生するエラー 151
表 6-6. リモートホストへの接続時に発生するエラー 152
表 6-7. 調査条件と調査結果に設定されるフラグ 153
表 6-8. リモートホストへの接続の確認時に発生するエラー 153
表 6-9. データの受信時に発生するエラー 154
表 6-10. データの送信時に発生するエラー 155
表 6-11. オプション設定時に発生するエラー 156
表 6-12. ソケットの切断時に発生するエラー 157
表 6-13. ソケットの破棄時に発生するエラー 157
表 6-14. ソケットアドレスの取得時に発生するエラー 158
表 6-15. nn::socket::AddrInfoFlag 列挙子 159
表 6-16. nn::socket::GetAddrInfo() によるホスト情報の取得時に発生するエラー 159
表 6-17. nn::socket::NameInfoFlag 列挙子 160
表 6-18. nn::socket::GetNameInfo() によるホスト情報の取得時に発生するエラー 160
表 6-19. サーバー検証オプションに設定可能な値 162
表 6-20. メソッドの一覧 166
表 6-21. 初期化時に発生するエラー 167
表 6-22. POST データのエンコード方法 170
表 6-23. POST データ遅延設定モードの送信データのタイプ 171
表 7-1. ローカル通信の性能測定で使用した測定要素 175
表 9-1. すれちがい通信中継タスクの実行結果 185
表 9-2. すれちがい通信中継サーバーから返される HTTP ステータスコードとその意味 186

図

図 3-1. 状態遷移図 21
図 3-2. 3DSダウンロードプレイで行う処理の流れ（サーバー側） 37
図 3-3. 3DSダウンロードプレイで行う処理の流れ（擬似クライアント側） 46
図 3-4. 双方向通信 58

図 3-5. Sender の状態遷移 60
図 3-6. Receiver の状態遷移 61
図 3-7. 接続開始時の状態遷移 62
図 3-8. 接続終了までの状態遷移 64
図 4-1. すれちがいボックスの走査 67
図 4-2. 伝播可能回数に 2 を設定したときの伝播の様子 76
図 4-3. グループの設定と処理の順番 79
図 4-4. すれちがいデータの送受信の例 83
図 4-5. BOSS の機能を利用したデータのダウンロード 86
図 4-6. タスクのプロパティ構成図 88
図 4-7. NADL タスクでダウンロードされたデータの流れ 102
図 4-8. OptOut フラグが有効に設定されている場合の動作 104
図 4-9. NADL タスクの処理に必要な手順 107
図 4-10. データアップロードタスクによるアップロード 121
図 4-11. DataStore アップロードタスクによるアップロード 121
図 4-12. DataStore ダウンロードタスクによるダウンロード 122
図 4-13. 短い間隔で自分のゲームモード説明文字列を更新した場合のバックグラウンド通信の例 128
図 5-1. ローカル受信拒否リストの登録の流れ 137
図 8-1. 2.4 GHz 帯のチャンネル割り当て 180
図 8-2. 各種電波シールド 182

1. はじめに

本ドキュメントは、3DS に搭載されている無線通信モジュールをアプリケーションで利用するために用意されている、各種ライブラリの概要や関数、プログラミング手順などについて説明したものです。

「2. 無線通信の概要」では、無線通信モジュールを利用した無線通信の種類と、それに属するライブラリの紹介など、無線通信の概要を説明します。

「3. フォアグラウンド通信」では、フォアグラウンド通信と呼ばれる、アプリケーションが意図的に行う無線通信に関するライブラリの概要などを説明します。

「4. バックグラウンド通信」では、バックグラウンド通信と呼ばれる、アプリケーションの裏で行われる無線通信に関するライブラリの概要などを説明します。

「5. 通信補助ライブラリ」では、不適切なユーザー作成コンテンツの広がりを防ぐための受信拒否リストなど、無線通信を利用する際にアプリケーションを補助するライブラリについて説明します。

「6. デバッグ用ライブラリ」では、製品には使用できませんが、デバッグ用途として用意されている無線通信のライブラリの使用方法などを説明します。

2. 無線通信の概要

この章では、アプリケーションで行うことのできる無線通信の種類や、無線オンモードと無線オフモードについて説明します。

2.1. 無線通信の種類

3DS の無線通信は、アプリケーションが意図的に行う通信（フォアグラウンド通信）とアプリケーションの裏で自動的に行われる通信（バックグラウンド通信）の 2 つに大きく分けることができます。

フォアグラウンド通信とバックグラウンド通信にはそれぞれ、用途に応じて複数のライブラリが用意されています。

表 2-1. フォアグラウンド通信に属する無線通信

無線通信の種類	ライブラリ名	名前空間	説明
ローカル通信（UDS 通信）	UDS	nn::uds	3DS 同士で直接通信を行う無線通信です。 1 つのネットワークに最大 16（TBD）台が接続可能で、ブロードキャストにも対応しています。
ニンテンドー3DSダウンロードプレイ	DLP	nn::dlp	通信用子機プログラムを配信するための無線通信です。 サイズが 32 MByte 以下のプログラムを配信することができます。
自動接続	AC	nn::ac	本体設定に従い、無線 LAN アクセスポイントなどを経由するインターネット接続を自動で確立するライブラリです。
信頼性のあるローカル通信（RDT 通信）	RDT	nn::rdt	1 対 1 の通信で欠落なしにデータを送受信するための無線通信です。 内部で UDS 通信を利用しています。

補足： UDS 通信では最大 16 台同時接続に対応していますが、13 台以上での同時接続は十分な動作確認が行われていません。そのため、現時点では 13 台以上は動作保証の範囲外とします。

表 2-2. バックグラウンド通信に属する無線通信

無線通信の種類	ライブラリ名	名前空間	説明
すれちがい通信	CEC	nn::cec	本体同士が近づいたときに通信相手を自動的に発見し、データの送受信を行う無線通信です。
いつの間に通信	BOSS	nn::boss	BOSS の機能を使った無線通信です。 アプリケーションで登録したタスクを、BOSS がバックグラウンドで処理します。
プレゼンス機能	FRIENDS	nn::friends	フレンドに通知するプレゼンス情報の設定やフレンドのプレゼンス情報の取得を行います。

2.2. 無線オンモードと無線オフモード

CTR および SPR 本体に搭載されている無線スイッチは、無線通信モジュールの使用をハードウェアで制御することができます。本体に物理的な無線スイッチを搭載していない SNAKE および FTR では、HOMEメニューに無線スイッチと同じ役割を持つボタンが用意されています。

無線スイッチによって無線通信モジュールが使用できる状態のことを「無線オンモード」、無線通信モジュールが使用できない状態のことを「無線オフモード」と呼びます。

無線オンモードであっても、UDS ライブラリやバックグラウンドで無線モジュールによる通信を行っていなければ、実際に無線電波の送受信が行われることはありません。

無線オフモードであれば、無線電波の送受信が行われることはありません。

無線通信モジュールの状態によって、無線ランプ（黄色）が、以下のように連動して変化します。

表 2-3. 無線通信モジュールの状態と無線ランプの状態の対応表

無線通信モジュールの状態	無線ランプの状態
無線オンモード	点灯
無線オフモード	消灯
無線電波の送信中	点滅

無線オンモード、無線オフモードを調べるための関数は用意されていません。無線オフモードの場合、ライブラリの初期化関数や無線通信で送受信を行う関数は無線オフモードを示す返り値を返します。

2.3. 無線通信の暗号化の処理負荷

認証方式に WEP や WPA が設定されているアクセスポイントとの通信で必要となる暗号化の処理は、無線通信モジュールがハードウェアで行います。WPA での接続で一部 CPU による処理が行われますが、システム側のコアで処理されます。

そのため、無線通信の暗号化についてはアプリケーション側の処理負荷を考慮する必要はありません。

2.4. 無線通信モジュールのエラー

3DS 本体の個体差により、CPU と 無線通信モジュール間のエラーが低確率で発生することがわかっています。この種のエラーはソフトウェアで修正することができないため、回避策としてエラー発生時に自動復帰を行っています。

ここでいう自動復帰とは、システムが自動的に無線オフモードに移行にして無線通信モジュールをリセットしたあと、すぐに無線オンモードに戻す処理のことを指します。ただし、個体差によるエラーのうち、ローカル通信や自動接続の初期化時、およびバックグラウンド通信の通信モード変更時に極まれに発生する RSL:F9606C02（RSL は FATAL 画面で表示されるリザルトコード）のエラーは自動復帰の対象外となっています。このエラーが発生したときには FATAL 画面が表示されるため、アプリケーション側で何らかの対応をする必要はありません。

> **補足：** 開発ツールおよび開発実機において RSL:F9606C02 のエラーが頻発する場合は、個体差によるものではなく無線通信モジュールが壊れている可能性があります。そのような場合は弊社窓口までご相談ください。

無線通信モジュールのエラーが発生した場合、その時の通信モードによって以下のように挙動が異なります。

- ローカル通信(UDS 通信、RDT 通信および ニンテンドー3DSダウンロードプレイ)
  自動復帰が働きます。その結果、API 実行時に本エラーが発生した場合、無線オフモードであることを示す `nn::uds::ResultWirelessOff` や `nn::dlp::ResultWirelessOff` などが返ります。
- 自動接続(AC)
  自動復帰が働きます。その結果、AC の API 実行時に本エラーが発生した場合、無線オフモードであることを示す `nn::ac::ResultWifiOff` が返ります。
- バックグラウンド通信(本体が開いているとき)
  いずれの通信モード(すれちがい通信、いつの間に通信、プレゼンス機能)においても自動復帰が働きます。アプリケーション側で何らかの対応をする必要はありません。
- バックグラウンド通信(本体が閉じている時)
  自動復帰は行われず、本体の電源が切断されます。アプリケーション側で何らかの対応をする必要はありません。
- 無線オフモード
  通信エラーは発生しません。

以下の表は、通信モード別に発生する可能性のある FATAL エラーの一覧です。

表 2-4. 通信モード別 FATAL エラー一覧

通信モード	リザルトコードと発生率
ローカル通信	RSL:F9606C02 発生率:極めて低い
自動接続	RSL:F9606C02 発生率:極めて低い
バックグラウンド通信(本体が開いているとき)	RSL:F9606C02 発生率:極めて低い
バックグラウンド通信(本体が閉じているとき)	FATAL 画面を表示せずに電源を切断 発生率:極めて低い
無線オフモード	発生しない

3. フォアグラウンド通信

アプリケーションが意図的に無線通信モジュールを利用した無線通信を行うために、以下のライブラリが用意されています。

- ローカル通信(UDS 通信)
- ニンテンドー3DSダウンロードプレイ
- 自動接続
- 信頼性のあるローカル通信(RDT 通信)

この章では、それぞれのライブラリを使用したアプリケーションの開発に必要な情報とプログラミング手順について説明します。

3.1. ローカル通信(UDS 通信)

3DS ではローカル通信として、ゲームでの利用を想定した任天堂独自の通信方式である「UDS 通信」を使用します。

UDS 通信は以下の特徴を持っています。

- 1 つのネットワークに対して、最大 16 台(現時点では 13 台以上は動作保証外です)が同時接続可能
- パケットのブロードキャストをサポート
- ピクチャーフレームに同期しない
- 送信データの到達は保証されない
- 伝送経路は暗号化されている
- データの送受信を定期的に行わなくとも接続が維持される

注意: 開発用の本体(デバッガ、開発実機)と製品版の本体との間では UDS 通信を行うことができません。

UDS 通信には、CTR-SDK で用意されている UDS ライブラリを使用します。

UDS ライブラリで使用する用語は以下のとおりです。

ノード / Node

UDS 通信のネットワーク上にある、通信(データの送受信)を行う端末です。

後述する、Master と Client を総称したものです。

ノード ID

UDS 通信のネットワークへの接続時に割り当てられる ID です。

詳細については「3.1.8. ノード ID の割り当て」を参照してください。

Master

新規に UDS 通信のネットワークを開設し、必要に応じてネットワークの属性を変更する権限を持つノードです。

DS / DSi の MP 通信における MP Parent に相当します。

Client

Master が開設した既存のネットワークに接続し、データの送受信を行うノードです。

DS / DSi の MP 通信における MP Client に相当します。

Spectator

Master が開設した既存のネットワークに接続し、データの受信のみを行う端末ですがノードではありません。

Client と異なり、ネットワークの同時接続台数としてカウントされません。Spectator の同時接続可能台数にはライブラリによる上限はありません。また、アプリケーションからその数を制限することもできません。

Unicast 通信

Unicast 通信は、あるノードから別の 1 つのノードに対してパケットを送信する方式で、基本的に Master と Client 間でのみ通信が可能です。アプリケーションからは Client 同士での通信も同じように扱うことができますが、Master を経由する必要がありますので、ライブラリ内で 2 回の送信が行われます。この方式では、すべて Master との通信で処理されるため Client からの送信レイテンシは高くなりますが、送信先のノードが通信可能範囲外になることはありません。

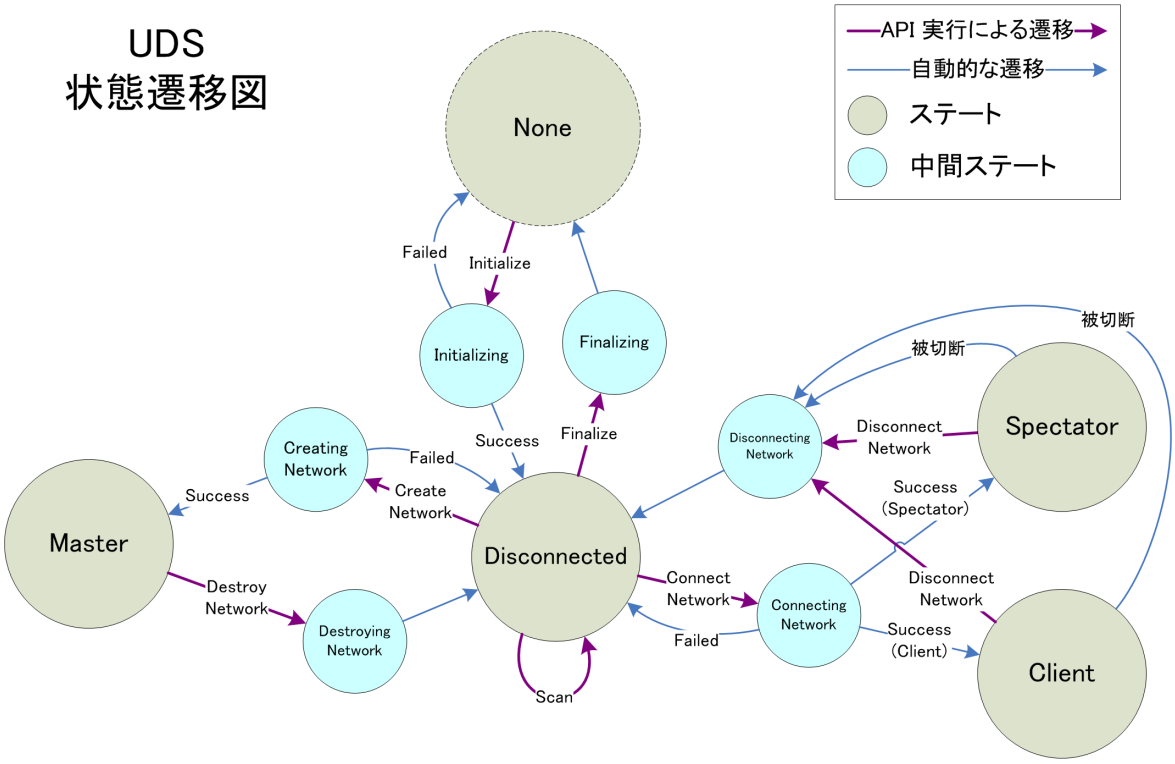
Broadcast 通信

Broadcast 通信は、あるノードから自分以外のすべてのノードに対してパケットを送信する方式で、1 回の送信で複数のノードにパケットを送信することができます。この方式では、ネットワーク内のノード数に関係なく 1 回の送信で処理されるため Client からの送信レイテンシも低くなりますが、Client から通信可能範囲外にあるノードに送信されるパケットが到達しなくなります。

3.1.1. UDS 通信の内部ステート

UDS ライブラリは、図 3-1 のように遷移する内部ステートを持っています。

図 3-1. 状態遷移図



内部ステートには状態を表すステートと、現在のステートから目的のステートへ遷移する際に経由する中間ステートがあります。中間ステートはライブラリ内部の処理用として存在しているステートのため、基本的にアプリ開発者が意識する必要はありませんが、もし現在のステートが中間ステートだった場合は、通常のステートに遷移するまで待つようにしてください。

UDS ライブラリはステートによって呼び出すことのできる関数に制限があり、そのステートで使用できない関数を呼び出した

場合は即座にエラーが返されます。

表 3-1. ステート

ステート名	説明
None	UDS ライブラリ初期化前の状態です。
Disconnected	UDS ライブラリの初期化が完了していて、ネットワークに属していない状態です。
Master	ネットワークを新規に構築し、Master として動作している状態です。
Client	既存のネットワークに Client として接続している状態です。
Spectator	既存のネットワークに Spectator として接続している状態です。

表 3-2. 中間ステート

中間ステート名	説明
Initializing	UDS ライブラリの初期化中です。成功した場合に Disconnected に遷移します。
Finalizing	UDS ライブラリを終了中です。終了後 None に遷移します。
Creating Network	設定に基づきネットワークを構築中です。終了後 Master に遷移します。
Connecting Network	既存のネットワークに Client/Spectator として接続中です。 成功した場合に Client/Spectator に遷移します。
Destroying Network	すべての Client/Spectator をネットワークから切断し、ネットワークを破棄しています。 終了後 Disconnected に遷移します。
Disconnecting Network	現在接続中のネットワークから離脱中です。終了後 Disconnected に遷移します。 外的要因によりネットワークから切断された場合にもこのステートになります。

これらのステートは `nn::uds::State` で定義されており、ステート名と定義名の対応は以下のようになっています。

表 3-3. ステート名と定義名の対応

ステート名	定義名
None	`STATE_NONE`
Disconnected	`STATE_DISCONNECTED`
Master	`STATE_MASTER`
Client	`STATE_CLIENT`
Spectator	`STATE_SPECTATOR`
Initializing	`STATE_PROCESSING_INITIALIZE` (このステートをアプリケーションで取得することはありません)
Finalizing	`STATE_PROCESSING_FINALIZE` (このステートをアプリケーションで取得することはありません)
Creating Network	`STATE_CREATING_NETWORK`
Connecting Network	`STATE_CONNECTING_NETWORK`

Destroying Network	STATE_DESTROYING_NETWORK
Disconnecting Network	STATE_DISCONNECTING_NETWORK

3.1.2. 初期化

UDS ライブラリの初期化は、nn::uds::Initialize() を呼び出して行います。

コード 3-1. UDS ライブラリの初期化

```
nn::Result nn::uds::Initialize(
        nn::os::Event* pStatusUpdateEvent,
        void* receiveBuffer, const size_t bufferSize);
nn::Result nn::uds::Initialize(
        nn::os::Event* pStatusUpdateEvent,
void* receiveBuffer, const size_t bufferSize,
        nn::cfg::UserName* pUserName);
```

pStatusUpdateEvent には nn::os::Event クラス型のインスタンスへのポインタを指定します。インスタンスは自動リセットイベントとして初期化されます。

Event インスタンスは nn::uds::Initialize() 内で初期化され、以下のタイミングでアプリケーションに対する通知が行われます。

- 自身がネットワークを構築した、もしくはネットワークに接続したとき
- 自身がネットワークから離脱した、もしくは切断されたとき
- 自身以外のノードがネットワークに接続したとき
- 自身以外のノードがネットワークから離脱した、もしくは切断されたとき

receiveBuffer と *bufferSize* には UDS ライブラリ内で使用する受信データバッファとそのサイズを指定します。バッファはアプリケーションで**デバイスメモリ以外から確保**し、先頭アドレスが nn::uds::BUFFER_ALIGNMENT (4096 Byte) アライメントでサイズが nn::uds::BUFFER_UNITSIZE (4096) の倍数でなければなりません。受信データバッファに指定したメモリ領域は、nn::uds::Finalize() による終了処理が完了するまで、アプリケーションからアクセスしないでください。

pUserName は Mii につけられた名前など、本体設定と異なるユーザー名を使用する場合に指定します。ユーザー名の取り扱いは UGC ガイドラインに従ってください。NULL を指定すると、*pUserName* を引数に持たない関数で初期化した場合と同様に、本体設定のユーザー名が使用されます。

なお、返り値に nn::uds::ResultAlreadyOccupiedWirelessDevice が返されたときは、無線通信モジュールがほかの通信で利用されているために UDS 通信を開始することができません。また、無線オフモードになっているときには、nn::uds::ResultWirelessOff が返されます。

3.1.3. 接続

UDS ライブラリの初期化後、UDS 通信のネットワークの構築を行います。ここでは、Master によるネットワークの構築および Client/Spectator によるネットワークへの接続の手順を説明します。

3.1.3.1. ローカル通信 ID の生成

UDS 通信のネットワークの構築や検索を行う関数の呼び出しには、ネットワークを識別するためにローカル通信 ID(32 bit)を指定しなければなりません。ローカル通信 ID は、以下の関数でユニーク ID(20 bit)から変換して生成します。

コード 3-2. ローカル通信 ID の生成

```
bit32 nn::uds::CreateLocalCommunicationId(bit32 uniqueId,
                                          bool isDemo = false);
```

uniqueId には、弊社業務部がタイトルごとに割り当てたユニーク ID を指定します。異なるタイトル間で通信を行う際は、どちらか片方の ID を指定してください。ID 0 はライブラリで予約されていますので指定しないでください。

isDemo には、ユニーク ID が共通となる製品版とダウンロードアプリ型体験版の間で UDS 通信を行いたくない場合に、体験版側のみ `true` を指定してください。体験版との通信を行うかどうかに関わらず、製品版では必ず `false` を指定してください。

> **補足:** 弊社からユニーク ID を割り当てられていないタイトルや実験プログラムで UDS 通信を使用する場合は、*uniqueId* にゲームソフト試作用コード(0xFF000～0xFF3FF)を指定してください。ただし、製品版では必ず弊社より割り当てられたユニーク ID を指定してください。

3.1.3.2. ネットワークの新規構築

UDS 通信のネットワークの構築は、Master が `nn::uds::CreateNetwork()` を呼び出すことで行われます。

コード 3-3. ネットワークの構築

```
nn::Result nn::uds::CreateNetwork(
        u8 subId,
        u8 maxEntry,
        bit32 localId,
        const char passphrase[], size_t passphraseLength,
        u8 channel);
nn::Result nn::uds::CreateNetwork(
        u8 subId,
        u8 maxEntry,
        bit32 localId,
        const char passphrase[], size_t passphraseLength,
        u8 channel,
        const void* pData, size_t dataSize);
```

subId は通信モード識別用 ID です。アプリケーション側で自由に設定することができます。たとえば、"通信対戦"や"データ交換" など、通信モードの区別に使用することを想定しています。*subId* に 0xFF は指定しないでください。0xFF は Client がすべての *subId* を対象にネットワークを探索する際に使用されます。

maxEntry にはネットワークに接続できるノードの最大数を指定します。Master、Client を含めて最大 16 まで指定可能ですが、13 台以上での同時接続は十分な動作確認が行われていません。そのため、現時点では 13 以上を指定した場合の動作は保証の範囲外とします。Master 自身が台数に含まれていることと、Spectator が台数に含まれないことに注意してください。

localId にはユニーク ID から生成したローカル通信 ID を指定します。

passphrase には無線レイヤの暗号化に使用する暗号鍵の種となる文字列(パスフレーズ)を指定します。*localId* と *passphrase* の組み合わせが、Master と Client/Spectator で一致する場合に通信が成立します。極端に短いものや連想されやすいもの、ほかのタイトルでも使っているものなどを指定して第三者に特定されたり、文字列そのものが漏洩したりしないように注意してください。

passphraseLength には *passphrase* の長さを指定します。指定可能な値の範囲は 8 以上、255 以下です。

channel には UDS 通信で使用するチャンネルを 0, 1, 6, 11 のいずれかから指定します。0 を指定した場合はチャンネルを自動で選択します。チャンネルの指定はデバッグ用途でのみ使用します。

pData と *dataSize* には、構築と同時にビーコンに独自データをセットしたい場合に、データとそのバイトサイズ指定します。*dataSize* に 0 を指定したときに限り、*pData* に NULL を指定してもエラーにはなりません。ビーコンにセットするデータについては「3.1.3.5. 独自データのビーコンへのセット」を参照してください。

> **注意:** 製品版の本体での実行時には、チャンネルが常に自動で選択されます。
>
> チャンネルを指定したネットワークの構築を行うには、デバッグモードに設定(Config ツールで「Debug Setting」の「Debug Mode」を「enable」に設定)する必要があります。
>
> UDS 通信ではネットワークを再構築したことを識別するために、ネットワークを構築するたびに TemporaryID を乱数生成しています。ただし再構築までの時間が短い場合、かつ Client からの接続がなく、機器間の通信が成立していない場合は TemporaryID を更新しません。

3.1.3.3. 周囲にあるネットワークの探索

Client/Spectator は、同じローカル通信 ID で構築されたネットワークが周囲にあるかどうかを nn::uds::Scan() を呼び出して探索する必要があります。

コード 3-4. 既存ネットワークの探索

```
nn::Result nn::uds::Scan(
        void * pBuffer, size_t bufferSize,
        u8 subId,
        bit32 localId);
```

pBuffer には 探索で発見したネットワークの情報を格納するバッファのアドレスを指定します。バッファに必要なサイズはネットワーク 1 つあたり 1 KByte 程度です。バッファのサイズを超える分のネットワークの情報は格納されません。

subId は探索の対象とする通信モード識別用 ID です。この引数で指定した値と同じ *subId* で構築されたネットワークが探索の対象となります。*subId* に 0xFF を指定したときは、すべてのネットワークが探索の対象となります。

localId にはユニーク ID から生成したローカル通信 ID を指定します。

> **注意:** 開発用の本体(デバッガ、開発実機)と製品版の本体とでは、双方とも相手をスキャンで発見することができません。

次に、nn::uds::Scan() を実行して得られたバッファの内容を nn::uds::ScanResultReader クラスのコンストラクタに渡して、探索結果を解析するインスタンスを生成します。

生成したインスタンスからは、メンバ関数の GetFirstDescription() で最初の探索結果を受け取ることができます。GetNextDescription() では、その次の探索結果を受け取ることができます。なお、探索結果は受信信号強度(RSSI 値)の大きなものから格納されています。

探索結果は nn::uds::NetworkDescriptionReader クラス型のインスタンスです。このインスタンスのメンバ関数でネットワーク情報の解析を行うことができ、GetNetworkDescription() でネットワーク情報を、GetNodeInformationList() でノードの一覧を、GetRadioStrength() で電波強度(4 段階)を取得することができます。また、CompareWith() ではネットワーク情報の比較を行い、比較結果を「完全に一致」、「同じネットワークだがデータが異なる」、「違うネットワーク」の 3 種類で取得することができます。

3.1.3.4. ネットワークへの接続

Client/Spectator は、解析された探索結果から得られる情報のうち、GetNetworkDescription() で取得したネットワーク情報を元に nn::uds::ConnectNetwork() を呼び出して既存のネットワークに接続することができます。

コード 3-5. ネットワークへの接続

```
nn::Result nn::uds::ConnectNetwork(
        const NetworkDescription & networkDescription,
        ConnectType type,
        const char passphrase[], size_t passphraseLength);
```

networkDescription にはネットワークの探索で得られたネットワーク情報を指定します。

type には接続する端末の種別を指定します。端末の種別は以下の 2 種類から選択することができます。

表 3-4. ネットワークへの接続モード

設定値	説明
CONNECT_AS_CLIENT	Client（送受信が行え、接続時にノード ID が付与されるノード）としてネットワークに接続します。
CONNECT_AS_SPECTATOR	Spectator（受信のみが可能な、接続時にノード ID が付与されないノード）としてネットワークに接続します。

Spectator としてネットワークに接続できるかどうかは、ネットワーク情報（nn::uds::NetworkDescription クラス）のメンバ関数 CanConnectAsSpectator() の返り値が true であるかどうかで判断することができます。

passphrase には無線レイヤの暗号化に使用する暗号鍵の種となる文字列（パスフレーズ）を指定します。Master が構築したネットワークのローカル通信 ID とパスフレーズの組み合わせと一致する場合に通信が成立します。

passphraseLength には *passphrase* の長さを指定します。指定可能な値の範囲は 8 以上、255 以下です。

> **補足：** 満員のネットワークに対して接続したときに返されるエラーは ResultAlreadyNetworkIsFull、解散済みのネットワークに対して接続したときに返されるエラーは ResultNotFoundNetwork です。

3.1.3.5. 独自データのビーコンへのセット

ネットワークを構築した Master は、最大 NET_DESC_APPDATA_SIZE_MAX バイトの独自データを、ネットワークのビーコンに設定することができます。

コード 3-6. 独自データのビーコンへのセット

```
nn::Result nn::uds::SetApplicationDataToBeacon(
                        const void* pData, size_t dataSize);
```

pData と *dataSize* には、独自データの先頭アドレスとそのサイズを指定します。

ビーコンに設定する独自データは、セッションの状態（対戦者待ちや対戦中など）やコメントなどを設定し、ネットワークを探索してきた Client や Spectator にネットワークの状態を知らせるといった、アプリケーション独自の接続状況の管理に使用することを想定しています。ただし、独自データをセットし直しても、すでにネットワークに接続されている Client や Spectator には変更が通知されないことに注意してください。

独自データの内容は暗号化されません。ビーコンは PC などで容易に傍受できることに注意してください。ビーコンの改竄

は検出できますが、傍受したビーコンをそのまま利用される可能性があります。特に、ビーコンで配信した特別なデータが不正に拡散されないようにする場合は、傍受したビーコンが別の場所で配信されても、ビーコンを受信するだけでは受け取れないようにする対策（Spectator としてネットワークに接続するなど）が必要になります。

3.1.3.6. ビーコンにセットされた独自データの取得

Master がビーコンにセットした独自データを Client/Spectator が取得できるタイミングは、ネットワークを探索した際に取得したネットワーク情報のメンバ関数を呼び出したときと、ネットワーク接続中に取得用の関数を呼び出したときです。

コード 3-7. ビーコンにセットされた独自データの取得

```
size_t nn::uds::NetworkDescription::GetApplicationData(
                          bit8* buffer, const size_t bufferSize) const;
nn::Result nn::uds::GetApplicationDataFromBeacon(
                          void* pBuffer, size_t* pDataSize, size_t bufferSize);
```

buffer と *bufferSize*、*pBuffer* と *bufferSize* にはそれぞれ、独自データを格納するバッファの先頭アドレスとそのサイズを指定します。バッファのサイズには独自データの最大サイズ（NET_DESC_APPDATA_SIZE_MAX）を指定し、バッファは最大サイズで確保してください。

取得した独自データのサイズは、nn::uds::NetworkDescription::GetApplicationData() は返り値で、nn::uds::GetApplicationDataFromBeacon() は *pDataSize* で、それぞれ確認することができます。

3.1.4. 通信

これまでの処理で UDS 通信でデータの送受信を開始する準備が整いました。続いてデータの送受信について説明します。

3.1.4.1. 端点の生成

データの送受信を行う前に、ネットワークの端点（Endpoint）を生成する必要があります。端点はデータの送受信を行う "入り口" のようなもので、nn::uds::CreateEndpoint() で生成することができます。

> **補足：** 現時点では、同時に生成可能な端点は 16 個までです。

送信用と受信用に別々の端点を生成することを推奨します。同じ端点を送信と受信で使うことは可能ですが、送受信を行っている最中の端点を別の送受信に使用しないように注意してください。

受信用の端点は、nn::uds::Attach() を呼び出し、ポート番号とノード ID を関連付けることでデータを受信できるようになります。この時、ライブラリは nn::uds::Initialize() で確保したメモリブロック上に受信用バッファを作成します。

コード 3-8. 端点とポート番号、ノード ID の関連付け

```
nn::Result nn::uds::Attach(
    EndpointDescriptor * pEndpointDesc,
    u16 srcNodeId,
    u8 port,
    size_t receiveBufferSize = ATTACH_BUFFER_SIZE_DEFAULT);
```

pEndpointDesc には nn::uds::CreateEndpoint() で生成した端点を指定します。

srcNodeId には送信元となるノード ID を指定します。指定されたノード以外から送られたデータは受信用バッファに格納されません。BROADCAST_NODE_ID を指定すると、すべてのノードから送信されたデータを受信用バッファに格納します。

port には受信するポート番号を指定します。指定したポート番号以外からのデータは受信用バッファに格納されません。ポート番号 0 はライブラリで予約されていますので指定しないでください。

receiveBufferSize には、確保する受信用バッファのサイズを ATTACH_BUFFER_SIZE_MIN 以上で指定しなければなりません。nn::uds::Initialize() で確保したメモリブロックから確保されますので、受信バッファの合計が確保したメモリブロックのサイズよりも大きくならないよう注意してください。また、32 Byte ごとにメモリを確保しますので、32 の倍数にすることでメモリの使用効率がよくなります。デフォルトでは ATTACH_BUFFER_SIZE_DEFAULT(UDS_PACKET_PAYLOAD_MAX_SIZE の 8 倍)の領域を受信用バッファとして確保します。

データの受信時に送信元のノード ID を取得することもできますので、複数のノードが同じポートに送信するデータを 1 つの端点で受け取っていても送信元を特定することができます。ただし、受信用バッファはリングバッファのため、サイズが十分でなければ先に受信したパケットから消失する可能性があります。

ネットワークからの切断時など、端点を介しての通信が不要になったときは、nn::uds::DestroyEndpoint() で端点を破棄してください。受信用の端点を破棄したときは、nn::uds::Attach() を呼び出したときに作成した受信用バッファはライブラリによって解放されます。

3.1.4.2. 送信

生成した端点を通して、ノードに nn::uds::SendTo() でデータを送信します。送信処理が完了するまでブロックされることに注意してください。

コード 3-9. データの送信

```
nn::Result nn::uds::SendTo(
        const EndpointDescriptor & endpointDesc,
        const void * data, size_t dataSize,
        u16 destNodeId, u8 port,
        bit8 option = 0x00);
```

endpointDesc には生成した端点を指定します。

data には送信するデータへのポインタを、*dataSize* にはそのサイズを指定します。1 回の呼び出しで送信できるデータの最大サイズは UDS_PACKET_PAYLOAD_MAX_SIZE で定義されています。*dataSize* に最大サイズを超える値を指定した場合、ResultTooLarge のエラーが返されます。送信データの先頭アドレスが 4 Byte アライメントでない場合、ResultMisalignedAddress のエラーが返されます。

destNodeId には送信先のノード ID を指定します。BROADCAST_NODE_ID(= 0xFFFF)を指定すると送信データはブロードキャストされます。Spectator が受信することのできるデータは、この方法でブロードキャストされたデータのみです。特定の Spectator に向けての送信はできません。また、Spectator はデータの送信を行うことができません。

port にはポート番号を指定します。ポート番号 0 はライブラリで予約されていますので指定しないでください。

option には、以下のフラグの論理和を指定します。これらのオプション設定を利用しない場合は、この引数の指定を省略する(0x00 を指定する)ことができます。オプション設定を使用しない場合は、Client から Client への Unicast 通信は Master を経由します。また、経由のために Master へ送信されるパケットを除き、送信バッファによるパケットのバッファリングが有効になります。

- NO_WAIT
  このフラグを指定した場合、UDS 内部の送信バッファにデータをためず、即座に送信を行いますが、送信が完了するまで処理がブロックされます。
  このフラグを指定しない場合は、なるべく送信遅延が起こらないように、送信バッファを無線送信の一回分(UDS_PACKET_PAYLOAD_MAX_SIZE とほぼ同等)という小さなサイズで確保します。これが一杯になった、もしくはバッファにある最も古いデータがためられてから一定時間(最大送信遅延時間)経った場合に、ライブラリがバッファ内

のデータを送信します。そのため、このフラグを指定せずに送信を繰り返し実行した場合の処理には、バッファにデータをためるだけのケースとバッファが一杯になり送信処理も行うケースが存在し、後者は前者に比べてブロック時間が長くなります。
なお、最大送信遅延時間のデフォルトは 10 ms に設定されていますが、nn::uds::SetMaxSendDelay() により、5～100 ms の範囲で指定可能です。

> **補足:** NO_WAIT を指定するかどうかは、通信に求めるリアルタイム性に依存します。
>
> 送信先にデータが到達するまでの時間は最小限であることが望ましく、データ交換にかかる時間をなるべく短くしたい場合は NO_WAIT を指定してください。
>
> インターネット通信と同程度のレイテンシが許容できる場合は、NO_WAIT を指定しないことで、より多くのパケットが扱えるようになります。複数のパケットをまとめて送信するようになるため、サイズの小さいパケットを大量に送信したときは、見た目の通信帯域が向上します。スループットは向上しませんが、1 秒間に送信できるパケットは増えます。

- FORCE_DIRECT_BC
  どの送信先に対してでも、Broadcast 通信による送信を行います。Client 間の Unicast 通信も Master を経由しません。送信レイテンシは低くなりますが、通信可能範囲外のノードを考慮する必要があります。

UDS 通信による送信の流れ

SendTo() で処理されたパケットは、NO_WAIT オプションが設定されていなければ送信バッファに可能な限りためられてから、NO_WAIT オプションが設定されていればすぐに、ライブラリによって無線フレーム(無線規格の通信単位)に詰められて送信されます。

各ノードが受信した無線フレームはライブラリによってパケットに分解され、それぞれの受信バッファに振り分けられます。このとき、Master 以外は自分宛ではないパケットを捨てますが、スター型ネットワークである UDS 通信で Client 間通信を行うために、Master はほかのノード宛のパケットを自身の送信バッファにためて再送するように設計されています。

送信パケットの順序の保証について

同じ送信先に同じオプション設定(FORCE_DIRECT_BC)で送信する場合には、パケットの順序は保証されています。

オプションを指定しない場合やオプションを使い分けた場合、経由するパスの違いから Unicast 通信と Broadcast 通信の間でパケットの順序が入れ替わる可能性がありますが、Unicast 通信で送信するパケットと Broadcast 通信で送信するパケットの順序が入れ替わること自体が問題となるケースはないと思われますので、基本的にはパケットの順序が保証されると考えてください。

送信の遅延について

ライブラリは無線通信の物理レイヤでの効率を良くするために、サイズの小さなパケットをまとめて送信するように設計されています。そのため、nn::uds::SendTo() により送信されたデータは設定された送信遅延時間の間、送信を待つ可能性があります。この送信遅延を回避し、回線効率よりもレイテンシを重視したい場合は nn::uds::SendTo() の引数 *option* に NO_WAIT を含む指定を行ってください。

nn::uds::SetMaxSendDelay() では、送信遅延時間の最大値を設定することができます。

コード 3-10. 最大送信遅延時間の設定

```
nn::Result nn::uds::SetMaxSendDelay(nn::fnd::TimeSpan maxDelay);
```

maxDelay には、最大送信遅延時間を 5～100 ms の範囲で指定してください。デフォルトは 10 ms です。

この関数は、ネットワークに接続していない状態でのみ実行可能です。

3.1.4.3. 受信

Attach() によってポート番号とノード ID との関連付けられた端点を介してデータの受信を行います。送信元のノード ID を取得する必要がない場合は nn::uds::Receive() を、取得する必要がある場合は nn::uds::ReceiveFrom() を呼び出してください。ここでは、nn::uds::ReceiveFrom() について説明します。

コード 3-11. データの受信

```
nn::Result nn::uds::ReceiveFrom(
        const EndpointDescriptor & endpointDesc,
        void * pBuffer,
        size_t * pReceivedSize,
        u16 * pSrcNodeId,
        size_t bufferSize,
        bit8 option = 0x00);
```

endpointDesc には nn::uds::CreateEndpoint() で生成し、nn::uds::Attach() で関連付けた端点を指定します。

pBuffer には受信データ格納先のポインタを指定します。バッファの先頭アドレスとサイズは 4 Byte のアライメントである必要があります。バッファの先頭アドレスとサイズが 4 Byte アライメントでない場合、それぞれ ResultMisalignedAddress と ResultMisalignedSize のエラーが返されます。

pReceivedSize には受信データサイズの格納先のポインタを指定します。1 回の呼び出しで受信できるデータの最大サイズは UDS_PACKET_PAYLOAD_MAX_SIZE で定義されています。

pSrcNodeId には送信元のノード ID の格納先のポインタを指定します。

bufferSize には受信データ格納先のバッファのサイズを指定します。

option に NO_WAIT を指定すると、データを受信していない場合でも即座に終了します。*option* を指定しない場合は、データを受信もしくはエラーが発生するまで受信を終了しません。

受信パケットが壊れている可能性について

UDS 通信では、データ破壊のチェックはハードウェアでは行われていますが、速度と適応性を重視しているため、ソフトウェアレベルでパケット単位の完全性チェックは行っていません。

壊れたり改竄されたりすると困るようなデータ(アイテム交換など)の場合は、アプリケーションで、ファイル単位などの適切な粒度でチェックを行ってください。改竄されても特に困らない場合やリアルタイム性が要求される場合は、パケット単位でのチェックをする必要はありません。

3.1.4.4. パケット消失の要因

ここでは、UDS 通信でパケットが消失する要因を説明します。

環境の悪化

通信する本体間の距離による電波強度の低下、第三者が送信する電波による干渉、ノイズなどの外的要因によって環境が悪化するとパケットの消失率が高くなります。

パケット消失率の実測値(ライブラリで保証するものではありません)は、良好な環境では 1% 程度、接続が維持できる程度に劣悪な環境では最大 10% 程度です。経験的な情報ですが、システムやデバイスでの処理が追いつかなくなると、それまでは低かった消失率が急激に高くなる傾向があります。

回線速度

UDS 通信の最大通信量の目安は、ネットワーク全体で秒間 500～600 パケット程度で、これを大きく超えた場合や周囲に別のネットワークが存在することで通信帯域が圧迫された場合にパケットが消失することがあります。

このケースでは、消失率は送信されるパケット量に左右されますので、接続台数を減らすとパケットの消失率が低くなります。

現在、com_demo1 デモのように単純な通信プログラムで単一のネットワークのみが通信を行った場合、秒間 800 パケットを超えたあたりで消失率が高くなる現象が確認されています。

受信バッファのバッファあふれ

アプリケーションが受信関数を呼び出す頻度よりもパケットを受信する頻度が高い場合、受信バッファがあふれて、もっとも古いパケットが消失することがあります。このケースでは、通信負荷の上昇に伴って徐々に消失率が高くなります。受信バッファのサイズを大きくすることで一時的な負荷の増大には対応できますが、恒常的なバッファあふれには対応できないことに注意してください。

> **注意:** SNAKE と CTR のプロセッサの処理速度の違いで通信に問題が生じます。そのため、プロセッサの処理速度に大きく依存した通信処理にしないでください。
>
> たとえば、送信過多にならないようにパケットの送信後にタイマーを用いた遅延処理を行う、送信側は受信側の受信状況を必要に応じて確認するなどの対応を行ってください。

3.1.4.5. 通信の効率を高めるには

UDS 通信のベースとなっている規格(IEEE 802.11b/g)では、同一チャンネルのネットワーク全体で無線帯域を共有しています。また、無線フレームの送信には干渉回避のための無通信時間(バックオフ)があるため、同じサイズのデータを送信する場合でも、無線フレームで送信可能な最大のデータサイズまでパケットをまとめるなどして、無線フレームの送信回数を減らすことで効率がよくなります。

UDS 通信のネットワークトポロジは、Master にすべての Client が接続するスター型です。通常の Unicast 通信では、Master から Client または Client から Master への送信が 1 パスで行われますが、Client から Client への送信は Master を経由するため 2 パスで行われます。同じデータを複数のノードに送信する場合は、Broadcast 通信を利用することによりネットワーク全体で送信されるパケットの数を減らすことができます。ネットワークに接続されているすべてのノードに一斉に送信する Broadcast 通信は 1 パスで行われ、しかもノードの個数に関わらず 1 回で送信することができます。ただし、Master からの Broadcast 通信ではすべてのノードが通信可能範囲に存在することが保証されていますが、Client からの Broadcast 通信では通信可能範囲外にノードが存在する可能性を考慮する必要があります。

メッシュ型のネットワークを UDS 通信で実装する

メッシュ型のネットワークではすべてのノードが同じ処理を行うため、通信処理の実装が比較的容易になります。しかし、すべてのノードがほかのすべてのノードにパケットを送信するため、通信台数が増加するとネットワーク全体で単位時間に送信するパケットの数が指数関数的に増加してしまいます。

UDS 通信では、Broadcast 通信を利用することで送信するパケットの数をノード数に比例する値にまで減らすことができますが、Client からの Broadcast 通信ではパケットが届かないノードが存在する可能性に注意しなければなりません。そのため、パケットの不達を検知した Client がパケットの再送を要求する処理が必要となります。再送処理は Unicast 通信で行うことになりますので、往復かつ Master を経由することによるレイテンシを考慮しなければなりません。

通常、Broadcast 通信ではすべてのノードに同じパケットを送信します。すべてのノードに同じデータを送信する場合は問題ありませんが、ノードごとに送信するデータが異なる場合はパケットの数を減らすことができません。送信オプションに FORCE_DIRECT_BC を指定した場合は Unicast 通信も Broadcast 通信のように 1 パスで送信され、受信したノードで自分宛ではないパケットが破棄されます。この機能とパケットをまとめて送信する機能を利用することで、ネットワーク全体で単位

時間に送信するパケットの数を減らすことができます。ただし、この方法では通信可能範囲外にノードが存在する可能性を考慮する必要があります。

3.1.5. 切断

ここでは Master による Client/Spectator の切断および、Client/Spectator によるネットワークからの切断の手順を説明します。

3.1.5.1. Master による Client/Spectator の切断

Master は次の方法で Client/Spectator をネットワークから切断することができます。

- 特定の Client をネットワークから切断
- すべての Client をネットワークから切断
- すべての Spectator をネットワークから切断

Client をネットワークから切断するには、`nn::uds::EjectClient()` を呼び出してください。

コード 3-12. Client をネットワークから切断

```
nn::Result nn::uds::EjectClient(u16 nodeId);
```

特定の Client をネットワークから切断するには、*nodeId* に切断したい Client のノード ID を指定してください。

すべての Client を一括でネットワークから切断する場合は、*nodeId* に `BROADCAST_NODE_ID`(= 0xFFFF)を指定してください。

Spectator をネットワークから切断するには、`nn::uds::EjectSpectator()` を呼び出してください。

コード 3-13. Spectator をネットワークから切断

```
nn::Result nn::uds::EjectSpectator(void);
nn::Result nn::uds::AllowToSpectate(void);
```

`nn::uds::EjectSpectator()` は、すべての Spectator をネットワークから切断し、以降は Spectator の接続を拒否するようになります。ネットワークを構築したときの初期設定では、Spectator の接続が許可されています。再度 Spectator の接続を許可するには、`nn::uds::AllowToSpectate()` を呼び出してください。

3.1.5.2. Master によるネットワークの破棄

Master はすべての Client/Spectator を切断し、ネットワークを破棄することができます。ネットワークを破棄するには、`nn::uds::DestroyNetwork()` を呼び出してください。

コード 3-14. ネットワークの破棄

```
nn::Result nn::uds::DestroyNetwork(void);
```

> **補足:** Master がネットワークを破棄した場合、Client が受け取る切断要因(`nn::uds::ConnectionStatus` 構造体の `disconnectReason`)は `DISCARDED_FROM_NETWORK` です。ただし、通信環境によっては `CONNECTION_LOST` となる可能性があります。

3.1.5.3. Client/Spectator による切断

Client/Spectator が現在接続中のネットワークから離脱するには、`nn::uds::DisconnectNetwork()` を呼び出してください。

コード 3-15. ネットワークからの離脱

```
nn::Result nn::uds::DisconnectNetwork(void);
```

3.1.5.4. Master による Client からの接続の制御

Master は Client のネットワークへの接続の拒否や許可を制御することができます。

コード 3-16. Client からの接続の制御

```
nn::Result nn::uds::DisallowToConnect(bool isDisallowToReconnect = false);
nn::Result nn::uds::AllowToConnect();
```

`nn::uds::DisallowToConnect()` は、現在の接続を維持したまま、以降の Client からの接続を拒否します。ただし、*`isDisallowToReconnect`* に `false` を指定した場合は、この関数を呼び出したあとに切断された Client であれば再接続を許可します。ネットワークを構築したときの初期設定では、Client の接続が許可されています。

`nn::uds::AllowToConnect()` は、Client からの接続の拒否を解除します。`nn::uds::EjectSpectator()` による Spectator の接続拒否は解除されません。

3.1.5.5. スリープ状態への遷移、無線オフモードへの切り替えによる切断

事前に UDS 通信を終了させずにスリープ状態へ遷移した場合や、無線スイッチにより無線オフモードに切り替わった場合、UDS 通信の内部ステートはエラー(`nn::uds::STATE_ERROR`)へと遷移します。この状態で `nn::uds::Finalize()` 以外の関数を呼び出すと、スリープ状態へ遷移していた場合は `nn::uds::ResultInvalidState` を、無線オフモードに切り替えていた場合は `nn::uds::ResultWirelessOff` を返すようになります。そのため、内部ステートがエラーから復帰する場合は `nn::uds::Finalize()` で UDS 通信を終了させ、再び初期化から行わなければなりません。無線オフモードへの切り替えで切断された場合は、無線オンモードに切り替わってから初期化する必要があります。

> **補足:** UDS 通信中にスリープ状態へと遷移する場合、アプリケーションは事前に UDS 通信を終了してからスリープ状態に遷移することを推奨します。

3.1.6. 終了処理

`nn::uds::Finalize()` を呼び出して、UDS 通信を終了します。

コード 3-17. UDS ライブラリの終了処理

```
nn::Result nn::uds::Finalize(void);
```

このとき、ライブラリは `nn::uds::Initialize()` で確保したメモリブロック領域と `nn::os::Event` インスタンスを解放します。UDS 通信を使用するアプリケーションは、その終了までに UDS 通信を終了させていなければなりません。「3DS プログラミングマニュアル - システム編」の「終了処理の注意事項」を参照してください。

3.1.7. ネットワーク状態の同期

UDS ライブラリは、ネットワークに接続しているノードの以下の情報について、自動的に同期を取っています。

- 現在の接続数
- 各ノードのノード ID
- 各ノードのローカルフレンドコード(プライバシー保護のためスクランブルがかけられています)
- 各ノードのユーザー名

アプリケーションでノードの情報の更新を明示的に通知する必要はありません。

また、これらの情報は Master/Client のすべてのノードで同期を取りますので、Client 側でも各ノードの情報を取得したり、新規接続・切断を検知したりすることができます。

3.1.8. ノード ID の割り当て

UDS 通信では、ネットワークに接続しているノードをノード ID という 16 bit の番号で管理しています。Master と Client にはノード ID が割り当てられますが、Spectator にはノード ID が割り当てられません。

同じネットワーク内では、MP 通信の AID と異なり、一度使用されたノード ID がその後別のノードに割り当てられることはありません。また、Client が同じネットワークに再接続すると、前回の接続時と同じノード ID が割り当てられます。

再接続時に前回と同じノード ID を割り当てるため、Master は内部で 64 台分の切断済みノード情報を保持しています。これを超える台数の切断が発生した場合は、もっとも昔に切断した Client のノード情報が破棄されます。切断済みノード情報から破棄されたノードがネットワークに再接続されると、前回とは異なるノード ID が割り当てられます。 なお、以下のノード ID は予約されているため、Client に割り当てられることはありません。

表 3-5. 予約済みノード ID

予約済みノードID	説明
0	存在しないノードです。ライブラリの内部処理で使用します。
1	Master に固定で割り当てられます。
0xFFFF(BROADCAST_NODE_ID)	ブロードキャストする場合に指定するノード ID です。

3.1.9. 各種情報の取得

自身の接続状態、リンクレベルなどの情報を取得することができます。

3.1.9.1. 接続状態の取得

`nn::uds::GetConnectionStatus()` を呼び出して、現在の接続状態を取得することができます。

コード 3-18. 接続状態の取得

```
nn::Result nn::uds::GetConnectionStatus(nn::uds::ConnectionStatus* pStatus);
```

pStatus には接続状態を受け取る nn::uds::ConnectionStatus 構造体へのポインタを渡してください。構造体は以下のように定義されています。

コード 3-19. nn::uds::ConnectionStatus 構造体の定義

```
struct nn::uds::ConnectionStatus
{
    State               nowState;
    DisconnectReason    disconnectReason;
    u16                 myNodeId;
    bit16               updateNodeBitmap;
    u16                 nodeIdList[NODE_MAX];
    u8                  nowEntry;
    u8                  maxEntry;
    bit16               slotBitmap;
};
```

nowState には現在のステートが格納されます。ステートについては「3.1.1. UDS 通信の内部ステート」を参照してください。

disconnectReason には切断理由が格納されます。格納される値と切断理由の対応は下表のとおりです。

表 3-6. 切断理由

値	切断理由
BEFORE_COMMUNICATION	まだ通信をしていない状態です。
NETWORK_IS_AVAILABLE	切断されていません。接続は維持されています。
REQUEST_FROM_MYSELF	自身の操作により接続を切断しました。
REQUEST_FROM_SYSTEM	スリープ状態や無線オフモードへの遷移など、3DS 本体からの要求で接続が切断されました。
DISCARDED_FROM_NETWORK	Master の操作により接続が切断されました。
CONNECTION_LOST	通信状況の悪化により接続が維持できなくなりました。
UNKNOWN_DISCONNECT_REASON	切断の原因は不明です。

myNodeId には自身のノード ID が格納されます。Spectator の場合は 0xFFFF が格納されています。

updateNodeBitmap には、前回の呼び出しから変更のあったノードを示すビットマップ情報が格納されます。1 になっているビットに対応する要素が変更のあったノードのノード ID です。ビットマップの 1 ビットが nodeIdList の 1 要素に対応し、下位ビットから上位ビットの順に先頭から最後の要素へと対応しています。

nowEntry と maxEntry には、ネットワークに接続しているノードの数と同時接続の最大数が格納されます。nowEntry には現在の状況が反映されますが、maxEntry は通信中に変化することはありません。

slotBitmap には、ノード情報が格納されているスロットを示すビットマップ情報が格納されます。前回の差分ではなく、現在の状況が反映されています。

3.1.9.2. リンクレベルの取得

nn::uds::GetLinkLevel() を呼び出して、現在のリンクレベル(通信品質)を取得することができます。

コード 3-20. リンクレベルの取得

```
nn::Result nn::uds::GetLinkLevel(nn::uds::LinkLevel* pLinkLevel);
nn::uds::LinkLevel nn::uds::GetLinkLevel();
```

pLinkLevel にはリンクレベルを受け取る変数へのポインタを渡してください。リンクレベルは下表のように定義されています。

表 3-7. リンクレベル

値	説明
LINK_LEVEL_0	通信品質が非常に悪い状態、もしくは通信が成立していません。
LINK_LEVEL_1	通信品質が悪い状態です。
LINK_LEVEL_2	通信品質はあまり良くありません。
LINK_LEVEL_3	通信品質は良好です。

電波強度はアプリケーションで常に表示しなければならない情報ではなく、この関数は処理をブロックするため、高頻度（ゲームフレームごとなど）で呼び出すことは推奨しません。

Client/Spectator がネットワークに接続する前に、指標として電波強度を表示する場合は、nn::uds::Scan() の結果から取得した nn::uds::NetworkDescriptionReader クラスの GetRadioStrength() を利用してください。

3.1.9.3. ノード情報の取得

nn::uds::GetNodeInformation() を呼び出して、指定したノードのユーザー情報を取得することができます。

コード 3-21. ノード情報の取得

```
nn::Result nn::uds::GetNodeInformation(nn::uds::NodeInformation* pNodeInfo,
                                       u16 nodeId);
```

pNodeInfo には、ノード情報を格納する nn::uds::NodeInformation 構造体へのポインタを指定します。

nodeId には、ノード情報を取得したいノードのノード ID を指定してください。

3.1.9.4. チャンネルの取得

nn::uds::GetChannel() を呼び出して、現在のチャンネルを取得することができます。

コード 3-22. チャンネルの取得

```
nn::Result nn::uds::GetChannel(u8* pChannel);
```

pChannel にはチャンネルを受け取る変数へのポインタを渡してください。

3.2. ニンテンドー3DSダウンロードプレイ

ニンテンドー3DSダウンロードプレイ（以降、3DSダウンロードプレイと呼びます）は、DSダウンロードプレイと同じように、無線通信で親機から子機プログラムを配信する機能です。3DSダウンロードプレイは以下のような特徴を持っています。

- 配信可能なプログラム（子機プログラム）のサイズは 32 MByte（33,554,432 Byte）まで
- 配信可能なプログラム（子機プログラム）は標準モードで動作するアプリのみ
- 必要であれば子機プログラムの配信前に、無線通信でシステム更新を行う
- 子機プログラムでローカル通信をするときに、親機を識別するための情報を取得可能
- 子機プログラムは専用領域に保存され、ほかの子機プログラムがダウンロードされたときに上書きされる
- 子機プログラムは HOME メニューには表示されない（HOME メニューからは起動できない）

注意： 開発用の本体（デバッガ、開発実機）と製品版の本体との間では 3DSダウンロードプレイを行うことができません。

補足： 子機プログラムのサイズはインポート後のサイズです。DevMenu の SDMC タブまたは HIO タブで子機プログラムを選択したときに表示される Required Size で確認することができます。

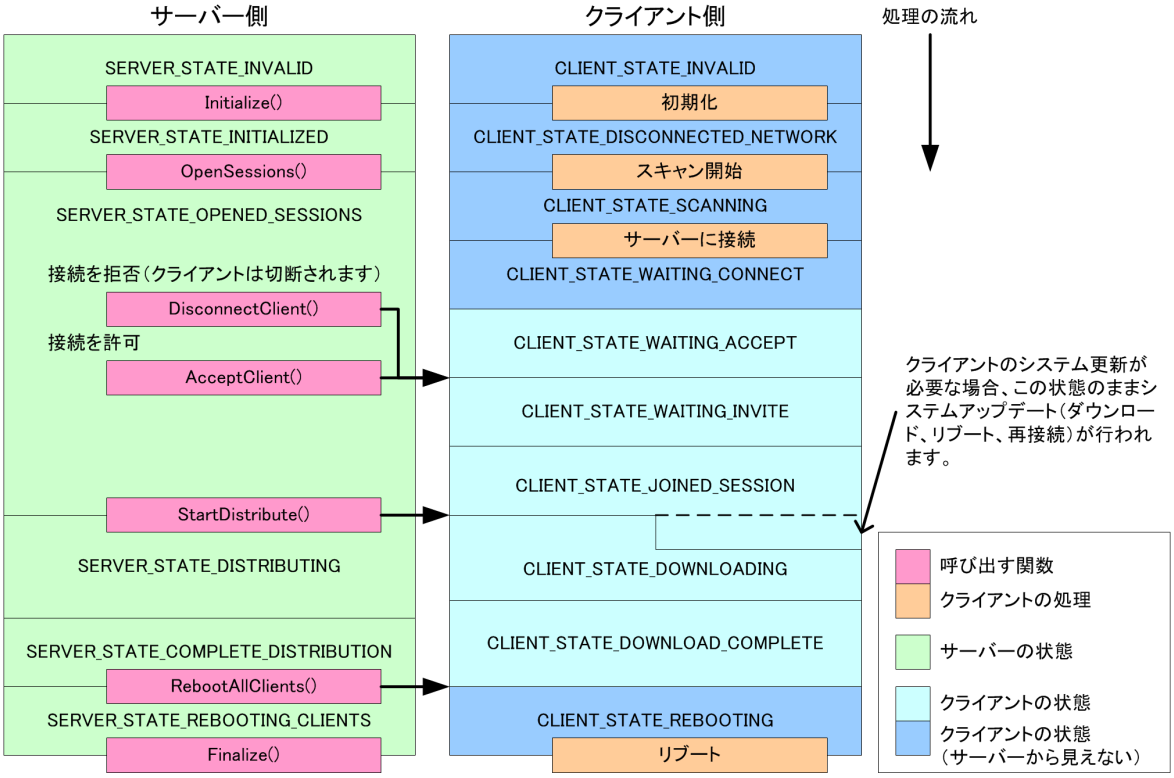
アプリケーションで 3DSダウンロードプレイに対応するには、CTR-SDK で用意されている DLP ライブラリを使用します。

クライアント側のプログラムはシステムアプリで用意されていますので、配信のためにアプリケーションが準備しなければならないのは、配信する子機プログラムと配信を制御するためのサーバーコードです。また、擬似ダウンロードプレイ子機の機能

をサポートするためのクラスも用意されています。通信処理などはライブラリ内部で行われますので、アプリケーションは状態を確認して必要な関数を呼び出すだけです。

以下の図は、3DSダウンロードプレイで行うサーバー側での処理の流れを、親機(サーバー)と子機(クライアント)の状態の遷移を軸に図示したものです。

図 3-2. 3DSダウンロードプレイで行う処理の流れ(サーバー側)



注意: 3DSダウンロードプレイを行うと、本体バージョンの低い開発実機上のクライアントはダウンロード完了後に FatalError になることがありますので、サーバーとクライアントの本体バージョンをそろえておく必要があります。実際の製品では、カードアプリならば無線通信によるクライアントのシステムアップデートが行われます。ダウンロードアプリならば本体更新開始直後に通信が中断します。

3.2.1. 初期化

サーバーコードで必要な関数は nn::dlp::Server クラスにまとめられています。DLP ライブラリ自体の初期化は必要なく、nn::dlp::Server クラスの Initialize() を呼び出してサーバーの初期化を行います。

コード 3-23. 初期化

```
static nn::Result nn::dlp::Server::Initialize(
                bool* pNotice, nn::Handle eventHandle, u8 maxClientNum,
                u8 childIndex, void* pBuffer, size_t bufferSize,
                size_t blockBufferSize = MIN_NETWORK_BLOCK_BUFFER_SIZE * 2,
                size_t blockBufferNum = MIN_NETWORK_BLOCK_BUFFER_NUM);
static size_t nn::dlp::Server::GetBufferSize(u8 maxClientNum,
                size_t blockBufferSize = MIN_NETWORK_BLOCK_BUFFER_SIZE * 2,
```

```
                size_t blockBufferNum = MIN_NETWORK_BLOCK_BUFFER_NUM);
```

pNotice には、ダウンロードプレイの警告メッセージを表示しなければならないかどうかが格納されます。true が返されたときは、ガイドラインに従ってメッセージを表示してください。メッセージは、ダウンロードアプリがサーバーになったときに、システムバージョンの低いクライアントからの接続が警告なく切断されることへの警告メッセージです。なお、今後の更新でシステムがダウンロードアプリによるシステムアップデートに対応した場合、メッセージの表示は必要なくなり *pNotice* に true が返されなくなります。システムアップデートの対応状況に応じてメッセージ表示の有無を切り替えられるよう、必ず *pNotice* の結果に応じてメッセージを表示してください。また、カードアプリでは *pNotice* に true が返されることはありません。しかし、*pNotice* に応じたメッセージ表示を組み込んでおくと、カードアプリをダウンロードアプリとしてリリースする際にソースコードの変更が不要になります。

eventHandle には、3DSダウンロードプレイからの通知を待つ nn::os::Event クラスのハンドルを渡します。ハンドルを渡すイベントはアプリケーションで初期化する必要があります。

maxClientNum には、サーバーに接続可能なクライアントの最大数を 1 ～ MAX_CLIENT_NUM の範囲で指定します。

childIndex には、配信する子機プログラムのインデックス(rsf ファイルの ChildIndex)を指定してください。クライアントに表示されるアイコンやタイトルなどは、この引数で指定された子機プログラムのヘッダ情報から取得されます。

pBuffer と *bufferSize* には作業バッファとそのサイズ、*blockBufferSize* と *blockBufferNum* にはブロックバッファ(子機プログラムの読み出しバッファ)のサイズと個数を渡します。作業バッファはアプリケーションで**デバイスメモリ以外から確保**し、先頭アドレスが nn::dlp::Server::BUFFER_ALIGNMENT (4096 Byte) アライメントでサイズが GetBufferSize() に *maxClientNum*、*blockBufferSize*、*blockBufferNum* を渡して取得したサイズ以上の nn::dlp::Server::BUFFER_UNITSIZE (4096) の倍数でなければなりません。引数に範囲外の値を指定した場合、GetBufferSize() は不定値を返します。

ServerWithName クラスは Server クラスの Initialize() にユーザー名の指定機能を追加したものです。このクラスを利用してユーザー名を設定する場合は、必ず UGC のガイドラインに従って NG ワードのチェックを行ってください。チェックの結果、NG ワードが含まれている場合は Initialize() に渡す構造体の isNgUserName を true にし、userName には伏せ文字などの加工を行っていないユーザー名をそのまま設定します。ServerWithName クラスの Initialize() 以外の動作は Server クラスと同じです。

Initialize() で返される可能性のある返り値とその原因を以下に示します。

表 3-8. Initialize() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultAlreadyOccupiedWirelessDevice	無線通信モジュールがほかの無線処理で占有されています。インフラストラクチャ通信などを終了させてください。
ResultOutOfRange	範囲外の値を引数に指定しています。
ResultInvalidPointer	不正なポインタを引数に指定しています。
ResultInvalidHandle	不正なハンドルを引数に指定しています。
ResultInternalError	ライブラリ内で回復不能なエラーが発生しました。
ResultFailedToAccessMedia	指定されたインデックスの子機プログラムが存在しないか、カードが抜けている可能性があります。
ResultChildTooLarge	子機プログラムが配信可能なサイズを超えています。
ResultInvalidRegion	親機と子機プログラムのリージョンが異なっています。

ResultWirelessOff	無線通信ができない状態(スリープ中または無線オフモード)です。

3.2.2. 状態の確認

サーバー側の状態を確認する方法には、初期化で渡したイベントがシグナル状態になったときに GetEventDesc() でイベントの詳細を取得する方法と、定期的に GetState() で状態を取得する方法があります。

コード 3-24. 状態の確認

```
static nn::Result nn::dlp::Server::GetEventDesc(
                                  nn::dlp::EventDesc* pEventDesc);
static nn::Result nn::dlp::Server::GetState(nn::dlp::ServerState* pState);
```

GetEventDesc() で取得することのできるイベントの詳細は 32 個までキューに保持され、いっぱいになると古いものから捨てられます。イベントがない場合はすぐに処理を戻し、返り値に nn::dlp::ResultNoData が返されます。

イベントのシグナル状態を監視しなければサーバーとして機能できないわけではありません。定期的に GetState() で状態を確認し、適宜処理を行う方がサーバーコードの実装は簡単になるでしょう。

> **補足:** サーバーの実装には、GetState() による状態のポーリングを強く推奨します。セッション参加の受付中やダウンロード中に、スリープに移行したり無線オフモードに切り替えられたりして、通信状態を維持できなくなったときは必ずエラー状態になるため、エラーへの対処が容易になります。

GetState() で取得することのできるサーバーの状態は下表のとおりです。

表 3-9. サーバーの状態

定義	説明
SERVER_STATE_INVALID	まだ初期化を行っていない状態です。
SERVER_STATE_INITIALIZED	初期化完了後の状態です。
SERVER_STATE_OPENED_SESSIONS	セッションへの参加を受け付けている状態です。クライアントの募集はこの状態でのみ行えます。
SERVER_STATE_DISTRIBUTING	配信を行っている状態です。
SERVER_STATE_COMPLETE_DISTRIBUTION	配信が完了した状態です。
SERVER_STATE_REBOOTING_CLIENTS	配信後にサーバーがクライアントをリブートしている状態です。すべてのクライアントが切断されたことを確認してから 3DSダウンロードプレイを終了してください。
SERVER_STATE_ERROR	エラーが発生し、再初期化が必要な状態です。この状態に遷移するのは、Initialize() から Finalize() までにスリープ状態への移行や無線オフモードへの切り替えが発生したときや、配信後のリブート以外で配信中にクライアントが切断したとき、カードが途中で抜けてしまったときです。

GetEventDesc()、GetState() で返される可能性のある返り値とその原因を以下に示します。

表 3-10. GetEventDesc()、GetState() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultNoData	取得できるデータがありません。
ResultInvalidPointer	不正なポインタを引数に指定しています。

3.2.3. セッションの開始と停止

3DSダウンロードプレイの処理で、クライアントからの配信要求を受け付け可能になる状態から、配信を完了してクライアントをリブートするまでをセッションと呼びます。

セッションの開始と停止は OpenSessions() と CloseSessions() で行うことができます。

コード 3-25. セッションの開始と停止

```
static nn::Result nn::dlp::Server::OpenSessions(
                                bool isManualAccept = false, u8 channel = 0);
static nn::Result nn::dlp::Server::CloseSessions();
```

isManualAccept に true を渡してセッションを開始した場合は、セッションに参加しようとしているクライアントへの接続の許可と拒否をアプリケーションで行うことができます。引数を省略した場合や false を渡して開始した場合は、自動的に接続が許可されます。

channel には、通信で使用するチャンネルを 0, 1, 6, 11 のいずれかから指定します。0 を指定した場合はチャンネルを自動で選択します。通常は何も指定する必要はありません。

> **注意:** 製品版の本体での実行時には、チャンネルが常に自動で選択されます。

セッションの途中でクライアントが切断されるなど、サーバーの状態遷移が止まってしまったときは、CloseSessions() を呼び出してセッションを停止してください。また、配信後にリブートされたクライアントが必ずしもサーバーに再接続するとは限りませんので、サーバーコードはユーザーが任意のタイミングでセッションを停止できるように実装しなければなりません。

OpenSessions()、CloseSessions() で返される可能性のある返り値とその原因を以下に示します。

表 3-11. OpenSessions()、CloseSessions() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultInvalidState	不適切な状態で関数を呼び出しました。
ResultOutOfRange	範囲外の値を引数に指定しています。
ResultWirelessOff	無線通信ができない状態(スリープ中または無線オフモード)です。

3.2.4. 接続状況の確認

セッションを開始したあとは、エラーが発生していない状態ならば、GetConnectingClients() で接続されているクライアントの数とそのノード ID を確認することができます。また、取得したノード ID からクライアントの情報や状態を GetClientoInfo() と GetClientState() で確認することができます。

コード 3-26. 接続状況の確認

```
static nn::Result nn::dlp::Server::GetConnectingClients(
                        u16* pNum, u16 clients[], u16 size);
static nn::Result nn::dlp::Server::GetClientInfo(
                        nn::dlp::NodeInfo* pClientInfo, u16 nodeId);
static nn::Result nn::dlp::Server::GetClientState(
                        nn::dlp::ClientState* pClientState, u16 nodeId);
static nn::Result nn::dlp::Server::GetClientState(
                        nn::dlp::ClientState* pClientState, size_t* pTotalNum,
                        size_t* pDownloadedNum, u16 nodeId);
```

GetConnectingClients() の引数 *pNum* にはセッションに接続されているクライアントの数が、*pClients* にはクライアントのノード ID の一覧が、それぞれ格納されます。*pNum* に渡すのが u16 型の変数へのポインタ、*clients* に渡すのが u16 型の配列であることに注意してください。*size* には *clients* に指定した配列のサイズを渡します。配列のサイズは初期化時に指定した *maxClientNum* 以上でなければなりません。

GetClientInfo() または GetClientState() の引数 *nodeId* には、情報を取得したいクライアントのノード ID を指定してください。存在しないクライアントやサーバーのノード ID を指定した場合は nn::dlp::ResultNoData が返されます。

GetClientInfo() の引数 *pClientoInfo* で指定された nn::dlp::NodeInfo 構造体に格納される情報は、「3.1.9.3. ノード情報の取得」で取得するノード情報と同じです。ユーザーの名前の表示などに利用することができます。

GetClientState() の引数 *pClientState* で指定された nn::dlp::ClientState 型のポインタには、クライアントの状態が格納されます。*pTotalNum* には子機プログラムの総パケット数が、*pDownloadNum* にはダウンロード済みのパケット数がそれぞれ格納されます。*pTotalNum* には 0 が格納されることがありますので、進捗の表示で除数に使用する場合は注意が必要です。

pClientState にはクライアントの状態が格納されています。

表 3-12. クライアントの状態（サーバーから確認できる状態のみ）

定義	説明
CLIENT_STATE_WAITING_INVITE	サーバーからセッション参加を許可され、セッションに招待されるのを待っている状態です。
CLIENT_STATE_WAITING_ACCEPT	ネットワークに接続されたクライアントが、サーバーからのセッション参加許可を待っている状態です。
CLIENT_STATE_JOINED_SESSION	セッションに参加している状態です。
CLIENT_STATE_DOWNLOADING	配信が開始され、ダウンロードを行っている状態です。配信開始直後などで、サーバーからデータを受け取っていない場合も含みます。
CLIENT_STATE_DOWNLOAD_COMPLETE	ダウンロードが完了した状態です。

GetConnectingClients()、GetClientInfo()、GetClientState() で返される可能性のある返り値とその原因

を以下に示します。

表 3-13. GetConnectingClients()、GetClientInfo()、GetClientState() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultInvalidState	不適切な状態で関数を呼び出しました。
ResultInvalidPointer	不正なポインタを引数に指定しています。
ResultNoData	指定されたノード ID を持つクライアントは接続されていません。

3.2.5. 接続の許可と拒否

セッションを開始してから配信を開始するまでの間、サーバーはクライアントからのセッションへの参加要求を受け付けます。引数 *isManualAccept* に true を渡してセッションを開始していた場合、アプリケーションでクライアントに対するセッションへの接続許可を制御することができます。

コード 3-27. 接続の許可と拒否

```
static nn::Result nn::dlp::Server::AcceptClient(u16 nodeId);
static nn::Result nn::dlp::Server::DisconnectClient(u16 nodeId);
```

AcceptClient() で接続を許可し、DisconnectClient() で接続を拒否することができます。クライアントの状態が CLIENT_STATE_WAITING_ACCEPT でなければ、これらの関数を呼び出して接続の許可や拒否を行うことができません。

AcceptClient()、DisconnectClient() で返される可能性のある返り値とその原因を以下に示します。

表 3-14. AcceptClient()、DisconnectClient() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultInvalidState	不適切な状態で関数を呼び出しました。
ResultNoData	指定されたノード ID を持つクライアントは接続されていません。
ResultWirelessOff	無線通信ができない状態（スリープ中または無線オフモード）です。

3.2.6. 配信の開始

配信先のクライアントを確定したあとは、StartDistribute() を呼び出してセッションへの参加を締め切り、クライアントへの配信を開始することができます。

コード 3-28. 配信の開始

```
static nn::Result nn::dlp::Server::StartDistribute();
```

配信を開始し、サーバーの状態が SERVER_STATE_PREPARING_FOR_TITLE_DISTRIBUTION に遷移します。

サーバーの状態が SERVER_STATE_COMPLETE_DISTRIBUTION に遷移して配信が完了するまで、クライアントの接続状況を確認して進捗などを表示しながら待機するだけです。

> **注意:** 配信が開始されても、状態が CLIENT_STATE_JOINED_SESSION のままになっているクライアントはシステム更新が必要なクライアントです。その状態のままシステムアップデート(ダウンロード、リブート、再接続)され、クライアントの状態が CLIENT_STATE_DOWNLOADING に遷移するまで、サーバーはクライアントの再接続を待ち受けている状態になっています。しかし、ダウンロードのキャンセルや電源の切断、無線オフモードへの切り替え、通信品質の悪化など、サーバーとクライアントが通信できない状況になる可能性があり、すべてのクライアントが必ずしも再接続されるとは限りません。そのため、ユーザーの操作でセッションをいつでも停止(CloseSessions())できるように実装してください。

StartDistribute() で返される可能性のある返り値とその原因を以下に示します。

表 3-15. StartDistribute() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultInvalidState	不適切な状態で関数を呼び出したか、配信すべき子機が存在しません。
ResultWirelessOff	無線通信ができない状態(スリープ中または無線オフモード)です。
ResultFailedToAccessMedia	メディアにアクセスできませんでした。カードが抜けている可能性があります。

> **補足:** 配信中にサーバーのカード抜けなどによって接続が切断されても、クライアントが中断画面に遷移しないのは仕様です。また、アプリケーションでこの事象に対応する必要はありません。

3.2.7. クライアントのリブート

配信が完了し、サーバーの状態が SERVER_STATE_COMPLETE_DISTRIBUTION に遷移したときは、セッションに参加していたすべてのクライアントをリブートするために RebootAllClients() を呼び出してください。

コード 3-29. クライアントのリブート

```
static nn::Result nn::dlp::Server::RebootAllClients(
                        const char passpharase[] = NULL);
```

passphrase には、リブート後の子機とローカル通信を行うために必要となる、ネットワーク構築時のパスフレーズを指定します。指定可能な文字列の長さは NULL を含めて、最大 MAX_CHILD_UDS_PASSPHRASE_LENGTH 文字です。パスフレーズはダウンロードセッションごとに異なる値を指定し、容易に推測できるような文字列を指定しないでください。

配信完了後にこの関数を呼び出した場合は、すべてのクライアントのリブートを行い、サーバーの状態が SERVER_STATE_REBOOTING_CLIENTS に遷移します。以降はクライアントの接続状況を確認し、すべてのクライアントの

切断を確認してから終了処理を行ってください。

RebootAllClients() で返される可能性のある返り値とその原因を以下に示します。

表 3-16. RebootAllClients() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultInvalidState	不適切な状態で関数を呼び出しました。

3.2.8. 終了処理

すべてのクライアントのリブートとセッションからの切断を確認したあとは、Finalize() でサーバーの終了処理を行うことができます。また、蓋が閉じられたことによるスリープ状態への遷移時には Finalize() を呼び出さなければなりません。

コード 3-30. 終了処理

```
static nn::Result nn::dlp::Server::Finalize();
```

終了処理の完了で 3DSダウンロードプレイは終了します。

配信した子機プログラムとローカル通信を行う場合は、終了処理以降に UDS ライブラリでネットワークを構築してクライアントの接続を待ちます。しかし、必ずしもクライアントからの再接続が行われるとは限らないことには注意が必要です。

Finalize() で返される可能性のある返り値とその原因を以下に示します。

表 3-17. Finalize() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。

3.2.9. 子機プログラムについて

3DSダウンロードプレイで子機プログラムをダウンロードしたあと、クライアントはサーバーの指示でリブートします。子機プログラムでサーバーだった本体を親機とのローカル通信を行う場合の補助として、親機への再接続のための情報を nn::dlp::GetRebootInfo() で取得することができます。

コード 3-31. 再接続情報の取得

```
nn::Result nn::dlp::GetRebootInfo(nn::dlp::RebootInfo* pRebootInfo);
```

この関数は DLP ライブラリの初期化を行わずに呼び出すことができます。*pRebootInfo* には、再接続のための情報を格納する nn::dlp::RebootInfo 構造体へのポインタを渡してください。

コード 3-32. nn::dlp::RebootInfo 構造体の定義

```
typedef struct
{
    u8      bssid[6];
    char    passphrase[MAX_CHILD_UDS_PASSPHRASE_LENGTH];
```

```
    NN_PADDING1;
} nn::dlp::RebootInfo;
```

bssid と passphrase には親機が構築するネットワークの BSSID とパスフレーズが格納されています。

これらの情報をもとに、どのネットワークに接続するのかをアプリケーションが自動的に判断することもできます。

3.2.9.1. 組み込み方法

rsf ファイルで指定する、子機プログラムとサーバー側のアプリケーションのユニーク ID は同じでなければなりません。また、子機プログラムの Category は DlpChild でなければなりません。なお、作成する子機プログラムに CTR アイコンは必要ですが、CTR タイトルバナーは必要ありません。

アプリケーションのビルドに OMake を利用している場合は、サーバー側のアプリケーションの OMakefile のビルド変数 "CHILD_APPS[]" に子機プログラムの cia ファイルを指定してください。

OMake を利用していない場合は、コマンドラインツールの ctr_makeciaarchive で cfa ファイルを作成し、ctr_makerom の引数 "-content" で cci ファイルに含めてください。

3.2.9.2. 強制ダウンロード

通常、クライアントは自身の持っていない子機プログラムか、自身にインポートされているものより新しい子機プログラムでなければダウンロードしませんが、Config ツールの強制インポート設定を有効にすることでクライアントに子機プログラムのバージョンを無視して強制的にダウンロードさせることができます。

3.2.9.3. デバッグ方法

子機プログラムのデバッグは、ダウンロードプレイで子機プログラムの実行が始まったあとに、以下の手順で行うことができます。

- デバッガの ATTACHA コマンドで子機プログラムにアタッチする
- 実行が一時停止するので、LS コマンドで axf ファイルからデバッグ情報を読み込む
- 実行を再開する

3.2.9.4. 子機プログラムの判定

nn::dlp::IsChild() を呼び出して、実行中のアプリケーションが子機プログラムであるかどうかを判定することができます。

コード 3-33. 子機プログラムの判定

```
bool nn::dlp::IsChild();
```

この関数は nn::dlp::Server や nn::dlp::FakeClient を使用しているとき(Initialize() を呼び出してから Finalize() が完了するまで)は呼び出さないでください。

3.2.10. 擬似クライアント

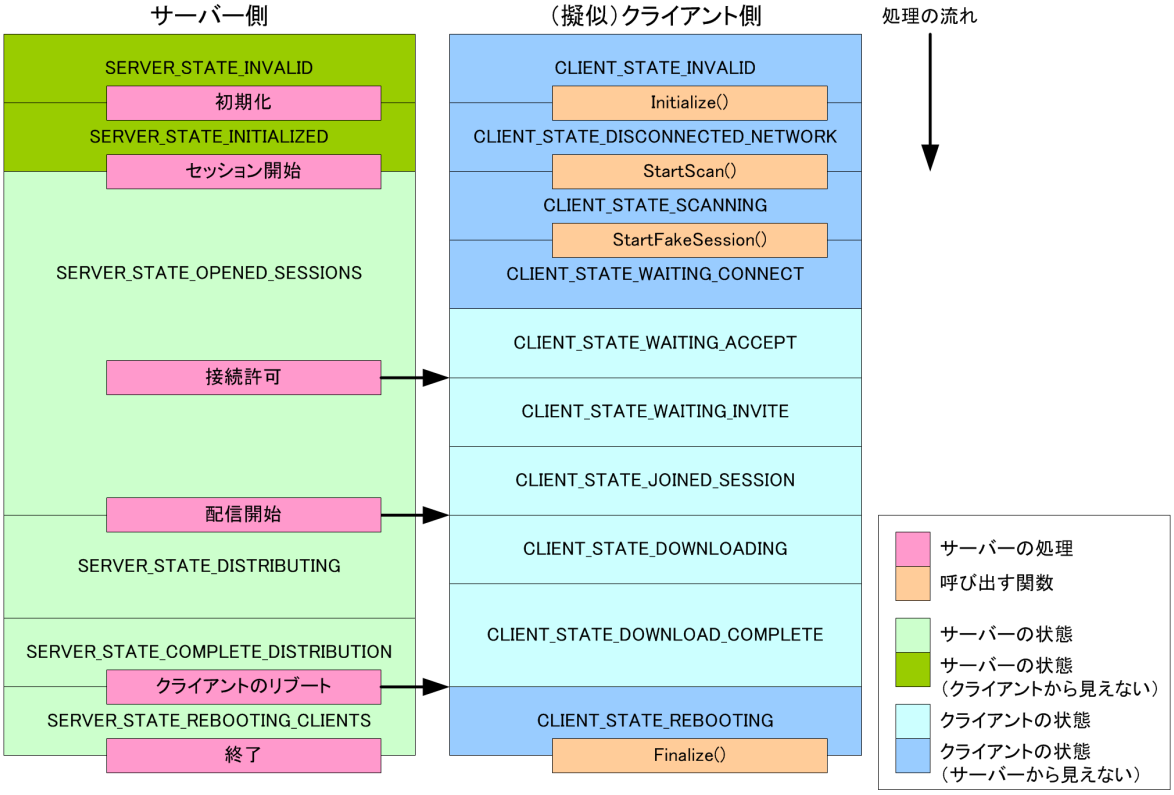
擬似クライアント(nn::dlp::FakeClient クラス)は、ダウンロードとリブートを行わないクライアントとして、アプリケーションからダウンロードプレイのセッションに参加するために用意されています。

擬似クライアントを利用すると、ダウンロードプレイで参加する 3DS と同じセッションに参加することができるため、アプリケーションと子機プログラムが混在するネットワークでの無線通信へと円滑に移行することができます。また、ソフトを持っていないユーザーとソフトを持っているユーザーをネットワークに混在させることができるという利点以外にも、セッションの管理などを DLP ライブラリにまかせることができるという利点があります。

> **注意:** 擬似クライアントを利用するタイトルをパッチや改訂版で修正する場合は、リマスターバージョンの異なる擬似クライアントが DLP サーバーに接続する場合について配慮する必要があります。
>
> 詳しくは「パッチマニュアル」の「疑似クライアント機能」を参照してください。

以下の図は、3DSダウンロードプレイで行う擬似クライアント側での処理の流れを、親機(サーバー)と子機(擬似クライアント)の状態の遷移を軸に図示したものです。

図 3-3. 3DSダウンロードプレイで行う処理の流れ(擬似クライアント側)



3.2.10.1. 初期化

擬似クライアントのコードで必要な関数は `nn::dlp::FakeClient` クラスにまとめられています。DLP ライブラリ自体の初期化は必要なく、`nn::dlp::FakeClient` クラスの `Initialize()` を呼び出して擬似クライアントの初期化を行います。

コード 3-34. 擬似クライアントの初期化

```
static nn::Result nn::dlp::FakeClient::Initialize(
u8 scanNum, nn::Handle eventHandle,
void* pBuffer, const size_t bufferSize);
static size_t nn::dlp::FakeClient::GetBufferSize(u8 scanNum);
```

scanNum には、1 回のスキャン要求で取得可能なタイトルの最大数を 1 ～ `MAX_SCAN_NUM` の範囲で指定します。

eventHandle には、3DSダウンロードプレイからの通知を待つ `nn::os::Event` クラスのハンドルを渡します。ハンドルを渡すイベントはアプリケーションで初期化する必要があります。

pBuffer と *bufferSize* には作業バッファとそのサイズを渡します。作業バッファはアプリケーションで**デバイスメモリ以外から確保**し、先頭アドレスが `nn::dlp::FakeClient::BUFFER_ALIGNMENT` (4096 Byte) アライメントでサイズが

GetBufferSize() に *scanNum* を渡して取得したサイズ以上の nn::dlp::FakeClient::BUFFER_UNITSIZE(4096) の倍数でなければなりません。引数に範囲外の値を指定した場合、GetBufferSize() は不定値を返します。

FakeClientWithName クラスは FakeClient クラスの Initialize() にユーザー名の指定機能を追加したものです。このクラスを利用してユーザー名を設定する場合は、必ず UGC のガイドラインに従って NG ワードのチェックを行ってください。チェックの結果、NG ワードが含まれている場合は Initialize() に渡す構造体の isNgUserName を true にし、userName には伏せ文字などの加工を行っていないユーザー名をそのまま設定します。FakeClientWithName クラスの Initialize() 以外の動作は FakeClient クラスと同じです。

Initialize() で返される可能性のある返り値とその原因を以下に示します。

表 3-18. Initialize() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultAlreadyOccupiedWirelessDevice	無線通信モジュールがほかの無線処理で占有されています。インフラストラクチャ通信などを終了させてください。
ResultOutOfRange	範囲外の値を引数に指定しています。
ResultInvalidPointer	不正なポインタを引数に指定しています。
ResultInvalidHandle	不正なハンドルを引数に指定しています。
ResultInternalError	ライブラリ内で回復不能なエラーが発生しました。
ResultWirelessOff	無線通信ができない状態(スリープ中または無線オフモード)です。

3.2.10.2. 状態の確認

擬似クライアント側の状態を確認する方法には、初期化で渡したイベントがシグナル状態になったときに GetEventDesc() でイベントの詳細を取得する方法と、定期的に GetMyStatus() で状態を取得する方法があります。

コード 3-35. 擬似クライアントの状態の確認

```
static nn::Result nn::dlp::FakeClient::GetEventDesc(
                        nn::dlp::EventDesc* pEventDesc);
static nn::Result nn::dlp::FakeClient::GetMyStatus(
                        nn::dlp::ClientStatus* pStatus);
```

GetEventDesc() で取得することのできるイベントの詳細は 32 個までキューに保持され、いっぱいになると古いものから捨てられます。イベントがない場合はすぐに処理を戻し、返り値には nn::dlp::ResultNoData が返されます。

イベントのシグナル状態を監視しなければ擬似クライアントとして機能できないわけではありません。定期的(ゲームフレームごとなど)に GetMyStatus() で状態を確認し、適宜処理を行う方がコードの実装は簡単になるでしょう。

> **補足:** 擬似クライアントの実装には、GetMyStatus() による状態のポーリングを強く推奨します。サーバーのスキャン中やダウンロード中に、スリープに移行したり無線オフモードに切り替えられたりして、通信状態を維持できなくなったときは必ずエラー状態になるため、エラーへの対処が容易になります。

GetMyStatus() で取得することのできる nn::dlp::ClientStatus は以下のように定義されています。

コード 3-36. nn::dlp::ClientStatus の定義

```
typedef struct
{
    u16             nodeId;
    SessionType     sessionType;
    ClientState     state;
    size_t          totalNum;
    size_t          downloadedNum;
} nn::dlp::ClientStatus;
```

state メンバに擬似クライアントの状態が格納されます。格納される値の定義は下表のとおりです。

表 3-19. 擬似クライアントの状態（擬似クライアントで確認できる状態のみ）

定義	説明
CLIENT_STATE_INVALID	まだ初期化を行っていない状態です。
CLIENT_STATE_DISCONNECTED_NETWORK	初期化完了後、またはサーバーから切断された状態です。
CLIENT_STATE_SCANNING	サーバーのスキャン中です。
CLIENT_STATE_WAITING_CONNECT	ネットワークへの接続を待っている状態です。
CLIENT_STATE_WAITING_INVITE	セッションへの招待を待っている状態です。
CLIENT_STATE_JOINED_SESSION	セッションに参加している状態です。
CLIENT_STATE_DOWNLOADING	配信が開始され、ダウンロードを行っている状態です。擬似クライアントでは、サーバーからデータを受け取りません。
CLIENT_STATE_DOWNLOAD_COMPLETE	ダウンロードが完了した状態です。
CLIENT_STATE_RECONNECTING_NETWORK	擬似クライアントでは、この状態に遷移することはありません。
CLIENT_STATE_REBOOTING	サーバーからのリブート要求を受信した状態です。
CLIENT_STATE_ERROR	エラーが発生し、再初期化が必要な状態です。 Initialize() から Finalize() までにスリープ状態への移行や無線オフモードへの切り替えが発生したときや、配信中にサーバーから切断されたり、配信中にほかのクライアントが切断したときにも、この状態に遷移します。

state メンバ以外のメンバに格納される値は、擬似クライアントの実装には必要ありません。

3.2.10.3. サーバーのスキャン

サーバー（タイトル）の一覧を取得するために、ダウンロードプレイのセッションを開始しているサーバーをスキャンします。

コード 3-37. サーバーのスキャン

```
static nn::Result nn::dlp::FakeClient::StartScan(
                u32 uniqueId, u8 childIndex, const u8* pMac = NULL);
static nn::Result nn::dlp::FakeClient::StopScan();
```

StartScan() を呼び出して、サーバーのスキャンを開始します。*uniqueId*、*childIndex*、*pMac* を指定することで、スキャン対象のタイトルやサーバーを絞り込むことができます。

> **注意:** 異なるアプリケーションのサーバーを選択すると、そのサーバーの動作に悪影響を及ぼす可能性があります。擬似クライアントは *uniqueId* と *childIndex* を必ず指定し、特定のタイトルを配信しているサーバーだけをスキャンの対象にしてください。

サーバーのスキャンは StopScan() で停止するまで続けられ、スキャンの結果は再度 StartScan() を呼び出すまで、最大で Initialize() の 引数 *scanNum* に指定した数が保持されます。

StartScan() で返される可能性のある返り値とその原因を以下に示します。

表 3-20. StartScan() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultInvalidState	不適切な状態(CLIENT_STATE_DISCONNECTED_NETWORK 以外)で関数を呼び出しました。

3.2.10.4. スキャン結果の取得

スキャン結果は GetTitleInfo() を呼び出して、タイトル情報(nn::dlp::TitleInfo)の一覧として取得します。また、取得したタイトル情報をもとに、GetServerInfo() でサーバー情報(nn::dlp::ServerInfo)を取得することができます。これらの関数は擬似クライアントの状態が CLIENT_STATE_INVALID や CLIENT_STATE_ERROR で呼び出すと nn::dlp::ResultInvalidState を返します。

コード 3-38. スキャン結果の取得

```
static nn::Result nn::dlp::FakeClient::GetTitleInfo(
        nn::dlp::TitleInfo* pTitleInfo, bool isReset = false);
static nn::Result nn::dlp::FakeClient::GetTitleInfo(
        nn::dlp::TitleInfo* pTitleInfo,
        const u8* pMac, u32 uniqueId, u8 childIndex);
static nn::Result nn::dlp::FakeClient::GetServerInfo(
        nn::dlp::ServerInfo* pServerInfo, const u8* pMac);
static nn::Result nn::dlp::FakeClient::DeleteScanInfo(const u8* pMac);
```

GetTitleInfo() のオーバーロードのうち、引数に *isReset* を持つものはタイトル情報一覧の取得に使用するためのものです。通常は、まだ GetTitleInfo() で取得していないタイトル情報を取得するように、*isReset* には false を指定します。返り値のインスタンスで IsSuccess() が true を返したときは、新たなタイトル情報を取得しています。未取得のタイトル情報がない場合や、これ以上タイトル情報を保持できない場合は、nn::dlp::ResultNoData が返されます。*isReset* に true を指定すれば、取得済みのタイトル情報を含め、タイトル情報の一覧を先頭から再取得することができます。StartScane() でスキャンを開始してから StopScan() でスキャンを中止するまでサーバーのスキャンは続けられ、最大で Initialize() で指定した *scanNum* 個のタイトル情報を保持することができます。タイトル情報の一覧は StartScan() を再度呼び出したときにクリアされます。

引数に *pMac*、*uniqueId*、*childIndex* を持つオーバーロードは、サーバーの MAC アドレス、ユニーク ID、子機プログラムのインデックスが分かっている場合に使用するためのものです。

nn::dlp::TitleInfo は、以下のように定義されています。

コード 3-39. nn::dlp::TitleInfo の定義

```
typedef struct
{
    u32                         uniqueId;
    u8                          childIndex;
    NN_PADDING3;
    u8                          mac[6];
    u16                         programVersion;
    bit8                        ratingInfo[RATING_INFO_SIZE];
    wchar_t                     shortTitleName[SHORT_TITLE_NAME_LENGTH];
    wchar_t                     longTitleName[LONG_TITLE_NAME_LENGTH];
    nn::dlp::IconInfo           icon;
    u32                         importSize;
    nn::cfg::CfgRegionCode      region;
    bool                        ulcd;
    NN_PADDING2;
} nn::dlp::TitleInfo;
```

uniqueId、childIndex、mac の 3 つはタイトル情報を特定するために利用します。

GetServerInfo() は、サーバーの情報の取得に使用します。*pMac* には、サーバー情報を取得したいサーバーの MAC アドレスを指定します。指定された MAC アドレスの機器がない、もしくはダウンロードプレイのサーバーとして動作していない場合は nn::dlp::ResultNoData が返されます。

サーバー情報にはリンクレベルや接続されているクライアントの一覧などの情報が含まれており、以下のように定義されています。

コード 3-40. nn::dlp::ServerInfo の定義

```
typedef struct
{
    u8                      mac[6];
    u8                      channel;
    nn::uds::LinkLevel      linkLevel;
    u8                      maxNodeNum;
    u8                      nodeNum;
    bit16                   dlpVersion;
    NN_PADDING4;
    NodeInfo                nodeInfo[MAX_NODE_NUM];
    nn::os::Tick            lastUpdateTick;
} nn::dlp::ServerInfo;
```

状態がスキャン中(CLIENT_STATE_SCANNING)であれば、リンクレベル(linkLevel)、サーバーを含んだ接続ノード数(nodeNum)、ノードの詳細情報(nodeInfo) には、その時点での情報が反映されています。それ以外の状態では変化しませんので、「3.2.10.6. 副次的な情報の取得」で紹介する関数を利用してください。

ダウンロードプレイのサーバーのノード ID には、ローカル通信での Master と同じように、必ず 1 が割り当てられています。サーバー情報のノード情報(nodeInfo)のうち、ノード ID(nodeId)が 1 であるノード情報のユーザー名(userName.userName)を表示するなど、スキャン対象を絞った状態でも複数のサーバーが一覧に表示される可能性を考慮してください。ただし、ユーザー名の情報(userName)の isNgUserName が true の場合は、ユーザー名に NG ワードが含まれていますので、ガイドラインに従って処理しなければなりません。

スキャン結果から特定のタイトル情報を削除したいときは DeleteScanInfo() を呼び出します。*pMac* に指定された MAC アドレスのサーバーのスキャン結果が削除されます。一定時間サーバーの情報が更新されなかった場合などに使用し

てください。指定された MAC アドレスのサーバーが存在しなかった場合は nn::dlp::ResultNoData を返します。また、状態が CLIENT_STATE_DISCONNECTED_NETWORK または CLIENT_STATE_SCANNING 以外のときに呼び出された場合は、nn::dlp::ResultInvalidState を返します。

3.2.10.5. セッションへの参加

サーバー(タイトル情報)の一覧から接続するサーバーを選択し、StartFakeSession() でセッションに参加します。擬似クライアントは、システムアプリのダウンロードプレイでセッションに参加してきたクライアントと同じようにサーバーからは見えますが、子機プログラムのダウンロードは行われません。

> **補足:** セッションに参加する前に、必ず StopScan() でサーバーのスキャンを停止しておいてください。

コード 3-41. セッションへの参加

```
static nn::Result nn::dlp::FakeClient::StartFakeSession(
                        const u8* pMac, u32 uniqueId, u8 childIndex);
static nn::Result nn::dlp::FakeClient::StopFakeSession();
```

引数 *pMac*、*uniqueId*、*childIndex* に指定する値は、GetTitleInfo() で取得したタイトル情報から得ることができます。

StartFakeSession() で返される可能性のある返り値とその原因を以下に示します。

表 3-21. StartFakeSession() で返される可能性のある返り値

返り値	原因
IsSuccess() が true を返す	処理に成功しました。
ResultInvalidState	不適切な状態(CLIENT_STATE_DISCONNECTED_NETWORK 以外)で関数を呼び出しました。
ResultNoData	*pMac*、*uniqueId*、*childIndex* で指定されたタイトルはありません。
ResultWirelessOff	無線通信ができない状態(スリープ中または無線オフモード)です。
ResultNotFoundServer	*pMac* で指定されたサーバーはありません。
ResultServerIsFull	すでに接続クライアント数が最大に達していたため、サーバーに接続できませんでした。接続クライアント数が減少しない限り、接続できません。
ResultDeniedFromServer	サーバーが子機プログラムの配信中のため、接続を拒否されました。
ResultConnectionTimeout	一定時間以内に接続が成立しませんでした。電波状況が悪い場合や、通信負荷が高い場合に返されます。
ResultInternalError	ライブラリ内で不整合によるエラーが発生しました。

セッションの途中で状態の遷移が止まってしまったときは、StopFakeSession() を呼び出してセッションを停止してください。この関数で返されるインスタンスは必ず IsSuccess() が true を返します。擬似クライアントもサーバーと同様に、ユーザーが任意のタイミングでセッションを停止できるように実装しなければなりません。

3.2.10.6. 副次的な情報の取得

擬似クライアントの実装には必要ではない、副次的な情報を取得するための関数が用意されています。

コード 3-42. 副次的な情報の取得

```
static nn::Result nn::dlp::FakeClient::GetConnectingNodes(
                        u8* pNum, u16* pNodeIds, u16 size);
static nn::Result nn::dlp::FakeClient::GetNodeInfo(
                        nn::dlp::NodeInfo* pNodeInfo, u16 nodeId);
static nn::Result nn::dlp::FakeClient::GetLinkLevel(
                        nn::uds::LinkLevel* pLinkLevel);
```

GetConnectingNodes() は、セッションに接続中のノードの一覧を取得することができます。

GetNodeInfo() は、セッションに接続中のノードの詳細情報を取得することができます。*pNodeInfo* に返される nn::dlp::NodeInfo はローカル通信の nn::uds::NodeInfomation と同じです。ノードの詳細情報については、「3.1.9.3. ノード情報の取得」を参照してください。

GetLinkLevel() は、サーバーとの間のリンクレベル(通信品質)を取得することができます。*pLinkLevel* に返される値については、「3.1.9.2. リンクレベルの取得」を参照してください。

3.2.10.7. ダウンロードの終了

擬似クライアントの状態が CLIENT_STATE_DOWNLOAD_COMPLETE に遷移したことは、このクライアントのダウンロードが完了していることを示しているだけで、ほかのクライアントはダウンロード中である可能性があります。ダウンロードプレイのセッション全体としてはまだ完了していませんので、サーバーに接続しているすべてのクライアントがダウンロードを完了するまで待機してください。

サーバーに接続していたすべてのクライアントがダウンロードを完了すると、サーバーはすべてのクライアントに対してリブート要求を送信します。リブート要求を受信すると、擬似クライアントの状態は CLIENT_STATE_REBOOTING へと遷移します。サーバーから指定されたローカル通信のパスフレーズは、この段階でのみ GetPassphrase() で取得することができます。

パスフレーズを取得し終えたら、Finalize() を呼び出して擬似クライアントの終了処理を行ってください。

コード 3-43. パスフレーズの取得と終了処理

```
static nn::Result nn::dlp::FakeClient::GetPassphrase(char* pPassphrase);
static nn::Result nn::dlp::FakeClient::Finalize();
```

GetPassphrase() の *pPassphrase* には、MAX_CHILD_UDS_PASSPHRASE_LENGTH バイト以上のバッファを指定してください。CLIENT_STATE_REBOOTING 以外の状態では ResultInvalidState が返されます。

Finalize() は必ず成功しますので、返されるインスタンスの IsSuccess() は常に true を返します。

そのまま親機(サーバー)とのローカル通信を行う場合は、セッションの開始時に選択したサーバーの MAC アドレスと GetPassphrase() で取得したパスフレーズを利用してください。

スリープ状態への移行時や HOMEメニューを起動しなければならないときは、必ず Finalize() を呼び出して、終了処理を行ってください。

3.3. 自動接続

AC(自動接続)ライブラリは、本体設定のネットワーク設定による無線 LAN のアクセスポイントやニンテンドー Wi-Fi USB コネクタ、ニンテンドーゾーン、Wi-Fi ステーション、公共無線 LAN アクセスポイントを経由してのインターネット接続を自動的に行うライブラリです。3DS の無線通信モジュールに直接アクセスするライブラリは用意されていませんので、3DS から外部のサーバーにアクセスする際には AC ライブラリを利用してインターネット接続(もしくは LAN 接続)を確立しなければなりません。

AC ライブラリは本体設定のネットワーク設定に従って無線 LAN のアクセスポイントなどにアクセスします。そのため、事前に本体設定で有効な無線 LAN アクセスポイントへの接続設定を記録しておかなければなりません。

> **補足:** 本体設定を行うメニューの代わりにネットワーク設定の変更を行う、NetworkSetting ツールが用意されています。また、AC ライブラリでは、デバッグ用途でのみ利用可能な関数として、ネットワーク設定 1 を変更する `nn::ac::DebugSetNetworkSetting1()`、自動接続で確立する接続がインターネットへの接続まで必要なのか、それともアクセスポイントへの接続までででよいのかを指定する `nn::ac::DebugSetNetworkArea()`、接続対象とするアクセスポイントの種類を `ApType` 列挙子(表 3-22. nn::ac::ApType 列挙子)の論理和で指定する `nn::ac::DebugSetApType()` が用意されています。
>
> ただし、これらの関数はデバッグモードに設定(Config ツールで「Debug Setting」の「Debug Mode」を「enable」に設定)されているときにのみ動作します。

AC ライブラリを利用してインターネット接続を確立させると、デーモンマネージャがインフラストラクチャ通信モードで無線通信モジュールを独占し、すれちがい通信などのバックグラウンド通信が行われるのを阻害する可能性があります。

AC ライブラリの関数は処理結果を `nn::Result` クラスのインスタンスで返します。処理が正常に行われたときは、`IsSuccess()` で `true` が返されます。

3.3.1. 初期化

AC ライブラリの初期化は `nn::ac::Initialize()` の呼び出しで行われます。

コード 3-44. AC ライブラリの初期化

```
nn::Result nn::ac::Initialize();
bool nn::ac::IsInitialized();
```

`nn::ac::ResultAlreadyInitialized` が返されたときは、すでにアプリケーションで初期化を行っていることを示します。エラーではありませんが、ライブラリは初期化回数の参照カウンタを保持していますので、`Initialize()` を呼び出した回数と同じ回数 `Finalize()` を呼び出さなければなりません。`nn::ac::IsInitialized()` を呼び出すと、すでに初期化が行われているかどうかを判断することができます。

3.3.2. 自動接続

自動接続処理は `nn::ac::CreateDefaultConfig()` で作成された接続条件に従って行われます。

コード 3-45. 接続条件の作成

```
nn::Result nn::ac::CreateDefaultConfig(nn::ac::Config* config);
```

config に渡された `nn::ac::Config` 構造体に接続条件が格納されます。構造体はアプリケーションで確保しなければなりません。この引数に `NULL` を指定した場合は `nn::ac::ResultInvalidData` が返されます。

作成された接続条件を `nn::ac::Connect()` または `nn::ac::ConnectAsync()` の引数 *config* に渡して自動接

続処理を開始します。

コード 3-46. 自動接続

```
nn::Result nn::ac::Connect(nn::ac::Config& config);
nn::Result nn::ac::ConnectAsync(nn::ac::Config& config, nn::os::Event* event);
nn::Result nn::ac::CancelConnectAsync();
nn::Result nn::ac::GetConnectResult();
bool nn::ac::IsConnected();
```

> **注意:** 自動接続処理中に EULA 同意のチェックが行われるため、事前に FS ライブラリの初期化を行う必要があります。

nn::ac::ConnectAsync() は nn::ac::Connect() の非同期版です。非同期版では、処理が成功したかどうかに関わらず、自動接続処理が終了した時点で *event* に渡したイベントがシグナル状態になります。イベントがシグナル状態になったあとは、必ず nn::ac::GetConnectResult() で処理結果を取得してください。処理結果を取得するまでローカル通信の開始に失敗し、すれちがい通信が行われない状態になります。自動接続処理を途中でキャンセルする場合は nn::ac::CancelConnectAsync() を呼び出してください。処理がキャンセルされるかキャンセルされる前に処理が成功したときは、nn::ac::ConnectAsync() の *event* に渡されたイベントがシグナル状態になります。処理がキャンセルされたときは、処理結果として nn::ac::ResultCanceled が返されます。

> **注意:** バックグラウンド通信の接続要求が処理されているときに nn::ac::CancelConnectAsync() を呼び出すと、アプリケーションの接続要求を取り下げて nn::ac::ResultSuccess を返します。

> **補足:** 自動接続の接続先は、ユーザー設定がニンテンドーゾーンやホットスポットよりも優先されます。ユーザー設定内の接続先優先順位は不定ですので、接続先を確実に指定するにはユーザー設定を一つにしてください。デバッグモードでは nn::ac::DebugSetApType() で接続先を指定できます。

自動接続処理が成功すると、インターネット接続が確立したときは nn::ac::ResultWanConnected が、無線 LAN 接続が確立したときは nn::ac::ResultLanConnected が処理結果として返されます。以降、ソケット通信などの無線通信モジュールを利用したネットワーク経由の通信を行うことができるようになります。接続可能なアクセスポイントを発見できなかったときは nn::ac::ResultNotFoundAccessPoint が返されます。

> **補足:** すでにバックグラウンド通信によって接続が確立していた場合、すぐに成功を返すことがあります。逆に接続が確立していない場合は処理に時間がかかることがあり、インターネット接続を確立できないアクセスポイントに接続した場合は通常よりも処理に時間がかかることがあります。

処理結果に nn::ac::ResultWifiOff が返されたときは、無線オフモードになっているために無線通信モジュールを利用することができない状態です。処理結果に nn::ac::ResultNotAgreeEula が返されたときは、EULA 同意チェックに失敗した場合とアプリケーションにアイコンファイルが設定されていない場合とがあります。そのほかの処理結果については関数リファレンスを参照してください。

nn::ac::IsConnected() を呼び出して、アプリケーションが行った自動接続処理によって確立した接続で、アクセスポイントに接続中かどうかを確認することができます。すでにアクセスポイントなどとの接続がバックグラウンド通信で確立していることが判明していても、必ずアプリケーションで自動接続処理を行ってください。アプリケーションで接続処理を行っていない場合、バックグラウンドで接続していたプロセスが終了したときに、意図せず接続が切断される可能性があります。

3.3.3. 接続状態の取得

自動接続処理で接続を確立したアクセスポイントの種類や、接続中のアクセスポイントの電波強度を、以下の関数で取得することができます。

コード 3-47. 接続状態の取得

```
nn::Result nn::ac::GetConnectingApType(nn::ac::ApType* apType);
nn::Result nn::ac::GetConnectingNintendoZoneBeaconSubset(
                                nn::ac::NintendoZoneBeaconSubset* beacon);
nn::Result nn::ac::GetConnectingHotspotSubset(nn::ac::HotspotSubset* hotspot);
nn::Result nn::ac::GetLinkLevel(nn::ac::LinkLevel* linkLevel);
nn::ac::LinkLevel nn::ac::GetLinkLevel();
```

nn::ac::GetConnectingApType() は接続しているアクセスポイントの種類を *apType* に返します。接続されていない状態で呼び出した場合は、返り値に nn::ac::ResultNotConnecting が返されます。アクセスポイントの種類は nn::ac::ApType 列挙子に、以下のように定義されています。アクセスポイントの種類は、ファームウェアのアップデートが行われた際に追加される可能性がありますので、記載されているもの以外を取得した場合についてもアプリケーションの実装で考慮する必要があります。

表 3-22. nn::ac::ApType 列挙子

列挙子	アクセスポイント
AP_TYPE_NONE	なし
AP_TYPE_USER_SETTING_1	ネットワーク設定 1
AP_TYPE_USER_SETTING_2	ネットワーク設定 2
AP_TYPE_USER_SETTING_3	ネットワーク設定 3
AP_TYPE_NINTENDO_WIFI_USB_CONNECTOR	ニンテンドー Wi-Fi USB コネクタ
AP_TYPE_NINTENDO_ZONE	ニンテンドーゾーン
AP_TYPE_WIFI_STATION	Wi-Fi ステーション
AP_TYPE_FREESPOT	フリースポット(AP_TYPE_HOTSPOT に統合されました)
AP_TYPE_HOTSPOT	ホットスポット
AP_TYPE_ALL	上記すべて(nn::ac::DebugSetApType() のみ)

nn::ac::GetConnectingNintendoZoneBeaconSubset() は接続中のニンテンドーゾーンのゾーンビーコンを *beacon* に格納します。nn::ac::GetConnectingHotspotSubset() は接続中のホットスポットの情報を *hotspot* に格納します。どちらの関数も、対応するアクセスポイント以外に接続しているときに呼び出した場合には nn::ac::ResultInvalidLocation を返します。

接続中のアクセスポイントの電波強度は、nn::ac::GetLinkLevel() で取得することができます。返り値または引数 *pLinkLevel* で取得できる電波強度は nn::ac::LinkLevel 列挙子に、以下のように定義されています。電波強度を取得する処理でエラーが発生したときは LINK_LEVEL_0 を返すため、エラー発生時に *pLinkLevel* の値を書き換える必要はありません。

表 3-23. nn::ac::LinkLevel 列挙子

列挙子	電波強度
LINK_LEVEL_0	通信品質が非常に悪い、もしくは通信が成立していない
LINK_LEVEL_1	通信品質が悪い
LINK_LEVEL_2	通信品質があまりよくない
LINK_LEVEL_3	通信品質がよい

ニンテンドー3DSステーションへの接続中は、機材間の距離によって AP_TYPE_WIFI_STATION が返る場合と AP_TYPE_NINTENDO_ZONE が返る場合があります。

いつの間に通信拠点へ接続中は、nn::ac::AP_TYPE_WIFI_STATION、nn::ac::AP_TYPE_NINTENDO_ZONE、nn::ac::AP_TYPE_HOTSPOT のいずれかが返ります。いつの間に通信拠点への接続中であることを検出したい場合はこれらの論理和で判断 してください。

3.3.4. 切断

自動接続処理で確立した接続は、アプリケーションで明示的に切断する場合と、電波状況の悪化などの外因によって切断される場合とがあります。

コード 3-48. 接続の切断

```
nn::Result nn::ac::Close();
nn::Result nn::ac::CloseAsync(nn::os::Event* event);
nn::Result nn::ac::GetCloseResult();
nn::Result nn::ac::RegisterDisconnectEvent(nn::os::Event* event);
```

nn::ac::Close() または nn::ac::CloseAsync() を呼び出すことで、確立していた接続をアプリケーションで明示的に切断することができます。nn::ac::CloseAsync() は nn::ac::Close() の非同期版です。処理が成功したかどうかに関わらず、切断処理が終了した時点で *event* に渡したイベントがシグナル状態になります。イベントがシグナル状態になったあと、nn::ac::GetCloseResult() で処理結果を取得してください。

切断されたかどうかについては、処理結果で返されるインスタンスの IsSuccess() が true を返すかどうかでのみ判断してください。バックグラウンドで動作しているデーモンが自動接続で接続を確立していた場合、切断後の状態が未接続(nn::ac::STATUS_IDLE)に遷移するとは限りません。

電波状況の悪化やアクセスポイントの圏外になるなどの理由で、確立していた接続が切断される場合があります。接続が切断されたことは、nn::ac::RegisterDisconnectEvent() の *event* に渡したイベントがシグナル状態になることによってアプリケーションに通知されます。自動接続処理で接続を確立したあとは、この関数で渡したイベントからの通知を待ち受けるスレッドを作成して外因による切断を検知してください。なお、イベントがシグナル状態になった時点でアクセスポイントとの接続が切断されていることは保証されていますので、ライブラリの終了処理前にアプリケーションで特別な処理を行う必要はありません。

3.3.5. 終了

ネットワーク経由の接続が不要になるなど、AC ライブラリの使用を終了するときは nn::ac::Finalize() を呼び出して終了処理を行ってください。

コード 3-49. AC ライブラリの終了

```
nn::Result nn::ac::Finalize();
```

初期化が行われていない状態で呼び出した場合は、nn::ac::ResultNotInitialized(エラーではありません)が返されます。

3.4. 信頼性のあるローカル通信(RDT 通信)

CTR-SDK では、信頼性のあるローカル通信を実現する RDT ライブラリを用意しています。

RDT ライブラリは、UDS ライブラリの上位ライブラリとして位置づけられており、以下の特徴があります。

- 1 対 1 で接続した相手にデータを欠落させることなく届ける
- 送信したバイト列は、その順序を保った形で受信できることが保証される
- 送信側から受信側への一方向通信(通信路を 2 つ用意すれば、双方向通信の実現は可能ですが、メモリなどのリソース消費は 2 倍になります)

RDT ライブラリの設計は、サイズの小さなデータを定期的に送信する用途には向いていませんが、ある程度サイズの大きなデータを送信する用途に向いています。

RDT ライブラリで使用する用語は以下のとおりです。

表 3-24. RDT ライブラリで使用する用語

用語	説明
Sender	RDT 通信で、データを送信する側のクラス
Receiver	RDT 通信で、データを受信する側のクラス

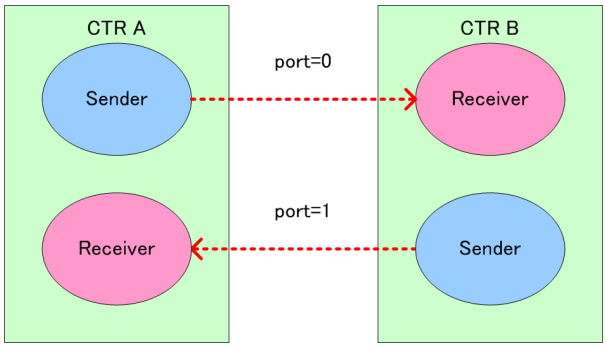
3.4.1. 初期化処理

RDT ライブラリは、UDS 通信を利用して信頼性のある通信を実現します。そのため、RDT ライブラリを使用するためには、事前に UDS 通信によるネットワーク接続が確立している必要があります。つまり、Master 側は nn::uds::CreateNetwork() の、Client 側は nn::uds::ConnectNetwork() の実行が完了していなければなりません。

ネットワーク接続を確立させた後に、Sender と Receiver の初期化を行います。それぞれ、nn::rdt::Sender クラスと nn::rdt::Receiver クラスのインスタンスを生成し、メンバ関数の Initialize() を呼び出してください。
Master/Client に関係なく、データの送信方向だけを考慮して、生成するインスタンスが Sender なのか Receiver なのかを決めてください。

通信路を 2 つ用意して双方向通信を行う場合は、Sender と Receiver で異なるポート番号を使用してください。

図 3-4. 双方向通信



3.4.1.1. Sender の初期化

`Initialize()` を呼び出して、Sender の初期化を行います。初期化が成功すると、Sender は内部で暗黙的に `nn::uds::EndpointDescriptor` を 2 つ生成します。関数内で `nn::uds::Attach()` を呼び出し、暗黙的に受信バッファを確保しますので、UDS ライブラリの初期化時に指定した受信バッファのサイズが十分に大きくなければ `nn::uds::RestultOutOfMemory` を返して初期化に失敗します。

コード 3-50. Sender の初期化関数

```
nn::Result nn::rdt::Sender::Initialize(const nn::rdt::SenderConfig &config);
```

config には、初期化パラメータをまとめた構造体(`nn::rdt::SenderConfig`)を指定します。構造体は以下のように定義されています。

コード 3-51. SenderConfig 構造体

```
struct nn::rdt::SenderConfig
{
    void *pWorkBuf;
    void *pSendBuf;
    u16   sendBufSize;
    u16   nodeId;
    u8    port;
    u8    padding[3];
};
```

pWorkBuf には、Sender 用に確保したワークメモリの先頭アドレスを指定します。ワークメモリは `nn::rdt::Sender::SENDER_WORKBUF_ALIGNMENT` (8 Byte) アライメント、`nn::rdt::Sender::SENDER_WORKBUF_SIZE` (32,768 Byte) 以上のバッファでなければなりません。

pSendBuf と *sendBufSize* には、送信用バッファの先頭アドレスとサイズを指定します。

nodeId と *port* には、通信相手(送信先)のノード ID とポート番号を指定します。ノード ID とポート番号は UDS ライブラリでの指定と同じです。

初期化で渡したワークメモリと送信用バッファは、`Finalize()` を呼び出すまで解放しないでください。

3.4.1.2. Receiver の初期化

`Initialize()` を呼び出して、Receiver の初期化を行います。初期化が成功すると、Receiver は内部で暗黙的に `nn::uds::EndpointDescriptor` を 2 つ生成します。関数内で `nn::uds::Attach()` を呼び出し、暗黙的に受信バッファを確保しますので、UDS ライブラリの初期化時に指定した受信バッファのサイズが十分に大きくなければ `nn::uds::RestultOutOfMemory` を返して初期化に失敗します。

コード 3-52. Receiver の初期化関数

```
nn::Result nn::rdt::Receiver::Initialize(
                        const nn::rdt::ReceiverConfig &config);
```

config には、初期化パラメータをまとめた構造体(nn::rdt::ReceiverConfig)を指定します。構造体は以下のように定義されています。

コード 3-53. ReceiverConfig 構造体

```
struct nn::rdt::ReceiverConfig
{
    void *pWorkBuf;
    void *pRecvBuf;
    u16  recvBufSize;
    u16  nodeId;
    u8   port;
    u8   padding[3];
};
```

pWorkBuf には、Receiver 用に確保したワークメモリの先頭アドレスを指定します。ワークメモリは nn::rdt::Receiver::RECEIVER_WORKBUF_ALIGNMENT (8 Byte) アライメント、nn::rdt::Receiver::RECEIVER_WORKBUF_SIZE (128 Byte) 以上のバッファでなければなりません。

pRecvBuf と *recvBufSize* には、受信用バッファの先頭アドレスとサイズを指定します。

nodeId と *port* には、通信相手(送信元)のノード ID とポート番号を指定します。ノード ID とポート番号は UDS ライブラリでの指定と同じです。

初期化で渡したワークメモリと受信用バッファは、Finalize() を呼び出すまで解放しないでください。

3.4.2. 状態の取得

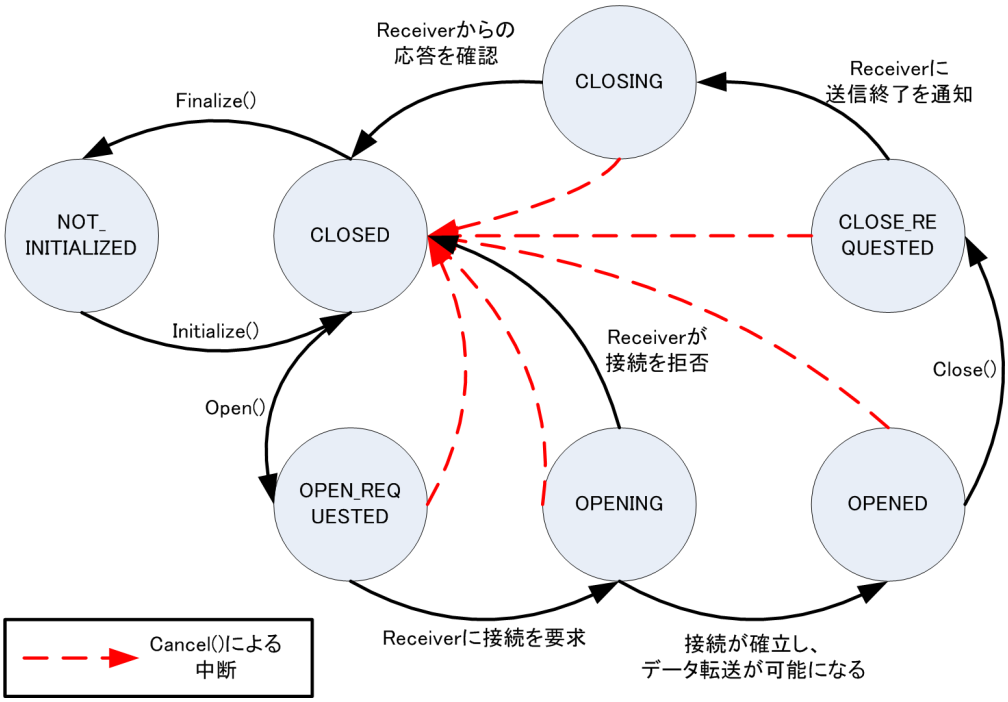
Sender と Receiver には、それぞれ内部状態が存在します。状態を取得するには、双方のクラスに用意されているメンバ関数 GetStatus() を呼び出してください。

コード 3-54. Sender/Receiver の状態の取得

```
enum nn::rdt::SenderState nn::rdt::Sender::GetStatus(void) const;
enum nn::rdt::ReceiverState nn::rdt::Receiver::GetStatus(void) const;
```

Sender と Receiver それぞれの、典型的な状態の遷移を以下に示します。

図 3-5. Sender の状態遷移

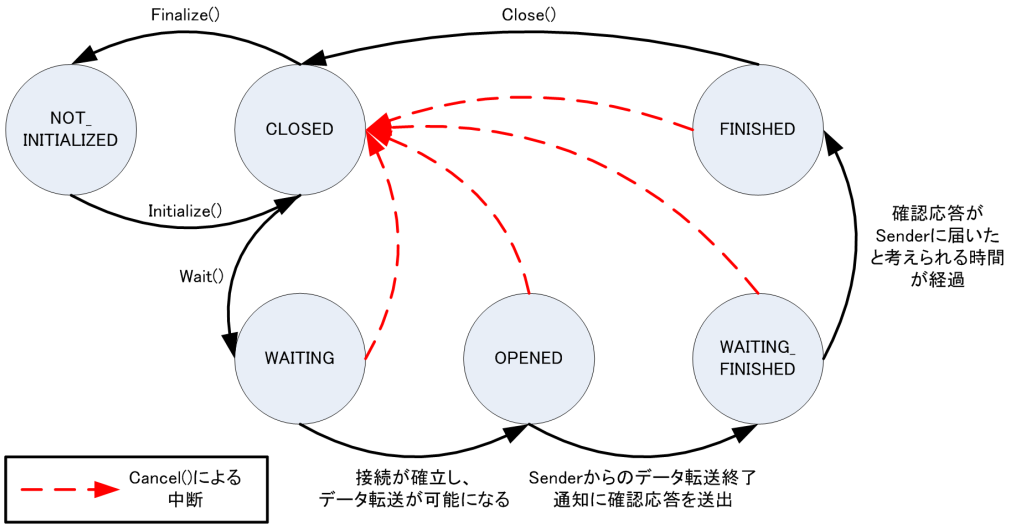


Sender の状態は nn::rdt::SenderState で定義されています。上の状態遷移図では、それぞれの状態を定義の先頭にある "SENDER_STATE_" を省略して表記しています。

表 3-25. Sender の状態(nn::rdt::SenderState の定義)

定義	説明
SENDER_STATE_NOT_INITIALIZED	インスタンス生成直後など、初期化が行われていない状態です。
SENDER_STATE_CLOSED	初期化後などに遷移する、インスタンスの初期状態です。
SENDER_STATE_OPEN_REQUESTED	接続を開始した直後に遷移する状態です。
SENDER_STATE_OPENING	通信相手に接続要求を送出し、反応を待っている状態です。
SENDER_STATE_OPENED	接続が確立し、データ送信が可能になった状態です。
SENDER_STATE_CLOSE_REQUESTED	Close() で接続の終了を宣言した直後に遷移する状態です。
SENDER_STATE_CLOSING	送信を完了し、相手にデータ転送終了通知を送出した状態です。

図 3-6. Receiver の状態遷移



Receiver の状態は nn::rdt::ReceiverState で定義されています。上の状態遷移図では、それぞれの状態を定義の先頭にある "RECEIVER_STATE_" を省略して表記しています。

表 3-26. Receiver の状態(nn::rdt::ReceiverState の定義)

定義	説明
RECEIVER_STATE_NOT_INITIALIZED	インスタンス生成直後など、初期化が行われていない状態です。
RECEIVER_STATE_CLOSED	初期化後などに遷移する、インスタンスの初期状態です。
RECEIVER_STATE_WAITING	通信相手からの接続要求を待っている状態です。
RECEIVER_STATE_OPENED	接続が確立し、データ受信が可能になった状態です。
RECEIVER_STATE_WAITING_FINISHED	データ転送終了通知を受け、確認応答を返した状態です。
RECEIVER_STATE_FINISHED	確認応答が相手に届いたと考えられる状態です。

後述する Cancel() などによって、通信の中断が指示されると CLOSED への遷移が発生します。

RDT の関数は、それぞれ呼び出すことのできる状態が決められています。現在の状態に対して不適切な関数が呼び出された場合は、即座にエラー(nn::rdt::ResultUntimelyFunctionCall)が返されます。

3.4.3. 通信処理の進行

Sender と Receiver の双方に用意されているメンバ関数 Process() は、RDT ライブラリの通信処理を進行させるために呼び出します。データの送受信や確認応答などの送受信処理は Process() 内部で実行されますので、インスタンスの初期化に成功したあとのアプリケーションは、少なくともフレームごと(60 フレーム換算で約 16.667 ミリ秒ごと)に Process() を呼び出すように設計してください。

コード 3-55. 通信処理の進行

```
nn::Result nn::rdt::Sender::Process(void);
 nn::Result nn::rdt::Receiver::Process(void);
```

> **補足:** Sender::Process() が nn::uds::ResultBufferIsFull を返す場合がありますが、このエラーは無視してもかまいません。到達が確認されていないデータは自動的に再送されます。

3.4.4. 接続の開始

Sender のメンバ関数 Open() を呼び出すことで、Sender は初期化パラメータで指定した相手のノード ID、ポート番号で動作している Receiver への接続を試みます。このとき、Receiver の状態が WAITING(Receiver のメンバ関数 Wait() の実行後)であれば接続要求は受諾され、Sender と Receiver はともにデータの送受信が可能な状態である OPENED に遷移します。

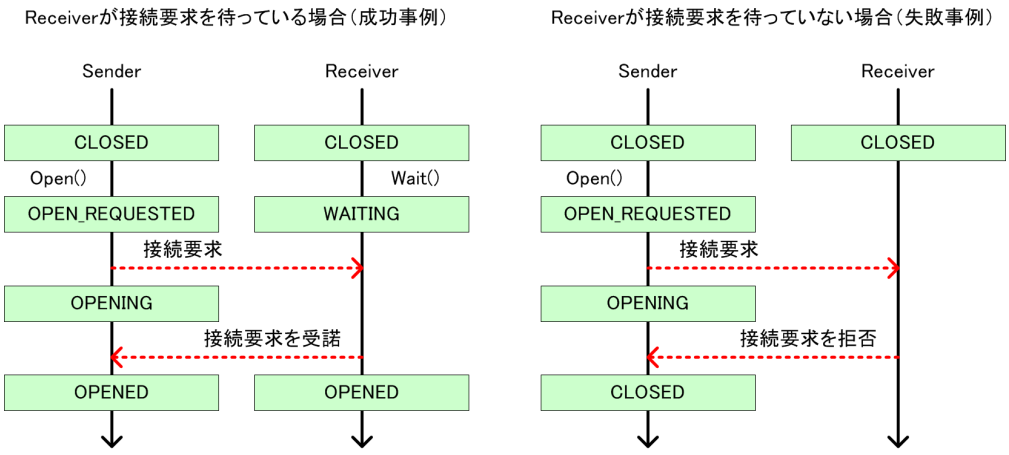
もし Receiver の状態が CLOSED であるときに Sender から接続を要求された場合、Receiver は接続を拒否するためにリセット信号を返し、これを受信した Sender の状態は CLOSED に遷移します。Open() 後に Sender の状態が CLOSED へ遷移しても、Receiver が Wait() を実行する前である可能性を考えて、数回接続を試みてください。

コード 3-56. 接続の開始

```
nn::Result nn::rdt::Sender::Open(void);
nn::Result nn::rdt::Receiver::Wait(void);
```

以下に、接続が開始されるまでの Sender と Receiver の状態遷移を図にしたものを示します。

図 3-7. 接続開始時の状態遷移



3.4.5. データの送受信

Sender と Receiver の状態がともに OPENED に遷移しているときは、データの送受信が可能です。

送信には nn::rdt::Sender::Send() を、受信には nn::rdt::Receiver::Receive() を呼び出します。Send() は、データを Sender 内部の送信バッファにデータを書き込む処理を、Receive() は Receiver 内部の受信バッファからデータを取り出す処理を実行するだけです。データが実際に無線で送受信される処理は、Process() を呼び出したときに行われます。

なお、RDT 通信で送受信されるデータには「バイト境界」のようなものは存在しません。そのため、仮に Sender がデータを 1024 バイトずつ送信したとしても、そのデータが Receiver に 1024 バイトずつ到着するとは限りません。

> **補足:** 送信側が nn::rdt::Sender::Send() で送信バッファに書き込んだ順序通りに、受信側では nn::rdt::Receiver::Receive() でデータを読み取ることができます。

コード 3-57. データの送受信

```
nn::Result nn::rdt::Sender::Send(const void* pBuf, size_t bufSize);
nn::Result nn::rdt::Receiver::Receive(
                        void* pBuf, size_t* pRecvSize, size_t bufSize);
```

3.4.6. 中断処理

ユーザーの指示により通信を中断する場合や UDS ライブラリでネットワーク切断を検知した場合は、Sender/Receiver の双方に用意されているメンバ関数 Cancel() を呼び出して、RDT 通信を中断してください。Cancel() は、状態を強制的に CLOSED に遷移させ、接続相手に向けてリセット信号を送出します。リセット信号を受信した接続相手の状態が CLOSED に遷移し、通信は中断されます。

コード 3-58. 中断処理

```
void nn::rdt::Sender::Cancel(void);
void nn::rdt::Receiver::Cancel(void);
```

3.4.7. 接続の終了

Sender がすべてのデータ送信し終えたときは、nn::rdt::Sender::Close() を呼び出してください。Close() は、これ以上送信すべきデータがないことを Sender に通知します。Close() が呼び出されると、状態は CLOSE_REQUESTED に遷移します。この状態に遷移したあとに呼び出された Process() の処理でクローズ指示が Receiver に送信され、Sender の状態は CLOSING に遷移します。

クローズ指示が Receiver に通知されると、Receiver の状態は WAITING_FINISHED に遷移します。この状態に遷移したあとに呼び出された Process() の処理でクローズ指示の確認応答が Sender に送信され、Sender に届いたと判断できるだけの時間が経過したときに Receiver の状態は FINISHED に遷移します。受信側は、Receive() で取得できるデータがなくなったことを確認してから nn::rdt::Receiver::Close() を呼び出してください。Receiver の状態が CLOSED に遷移し、接続が閉じられます。

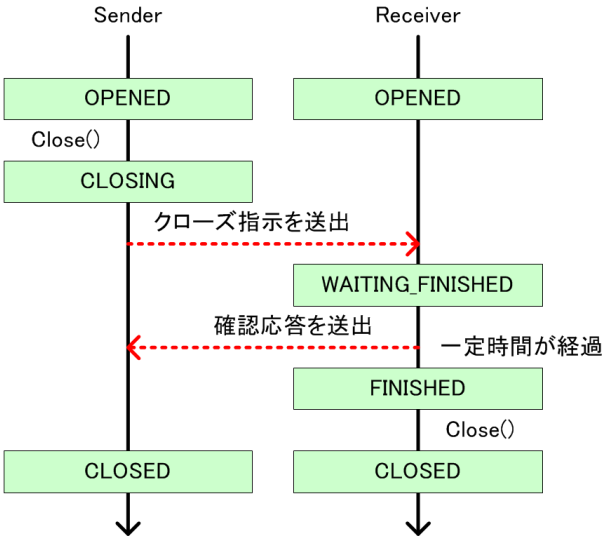
通信を中断する目的で Close() を呼び出さないでください。nn::rdt::Sender::Close() は、送信すべきデータをすべて正常に送信したことを指示するために用意されています。また、nn::rdt::Receiver::Close() は、受信すべきデータをすべて正常に受信できたことを確認してから呼び出されることを想定しています。

コード 3-59. 接続の終了

```
nn::Result nn::rdt::Sender::Close(void);
nn::Result nn::rdt::Receiver::Close(void);
```

以下の図は接続の終了までの状態の遷移を示したものです。

図 3-8. 接続終了までの状態遷移



3.4.8. 終了処理

RDT ライブラリの利用を終える際には、Sender と Receiver の双方に用意されているメンバ関数 `Finalize()` を呼び出して、終了処理を行ってください。`Finalize()` により、初期化処理の際にインスタンスが暗黙的に確保していた `nn::uds::EndpointDescriptor` が解放され、初期化処理で渡していたメモリを確保しておく必要もなくなります。

コード 3-60. 終了処理

```
void nn::rdt::Sender::Finalize(void);
void nn::rdt::Receiver::Finalize(void);
```

4. バックグラウンド通信

アプリケーションの裏で自動的に行われる無線通信（バックグラウンド通信）をアプリケーションで利用するために、以下のライブラリが用意されています。

- すれちがい通信（CEC）
- いつの間に通信（BOSS）
- プレゼンス機能（FRIENDS）

この章では、それぞれのライブラリを使用したアプリケーションの開発に必要な情報とプログラミング手順について説明します。

バックグラウンドで動くデーモンによるアプリケーションへの影響

デーモンとはバックグラウンド動作している常駐モジュールのことで、3DS にはネットワークの処理を行う複数のデーモンが存在します。これらのデーモンを統括する特別なデーモンとして ndm（Network Daemon Manager）があり、SDK には ndm を制御するための NDM ライブラリが用意されています。

アプリケーションの動作する CPU コアと異なる CPU コアでデーモンは動作しているため、デーモンがアプリケーションの CPU 時間を直接奪うようなことはありません。しかし、ネットワーク（無線通信）や NAND へのアクセスは競合するため、デーモンの動作がアプリケーションのパフォーマンスに影響を与える場合があります。そのため、ネットワークや NAND へのアクセスパフォーマンスの低下が致命的である場合は、NDM ライブラリでデーモンの動作をアプリケーションで停止することができます。

> **補足：** 明示的に停止を解除しなければ、アプリケーションの動作中はいつの間に通信が停止した状態になります。なお、スリープ中やタスクの即時実行は停止の影響を受けません。

デーモンは自由に停止することができる一方で、デーモンにはそれぞれ役割があり停止すると様々な制限が生じます。したがって、デーモンを停止する場合は、停止することによってどのようなデメリットが生じるのかを十分に理解していなければなりません。

以下に、現時点で判明しているデーモンによるアプリへの影響を示します。

表 4-1. 現時点で判明しているデーモンによるアプリへの影響

デーモン（nn::ndm::DaemonName）	ネットワークへの影響	NAND への影響
nn::ndm::DN_CEC	なし（インフラストラクチャ通信を使用しません）	中（すれちがい成立時に 10 KByte ～ 10 MByte 程度の NAND アクセスが発生）
nn::ndm::DN_BOSS	高（タスク実行時に HTTP 通信が発生）	高（タスク実行時に NAND 書き込みが発生）
nn::ndm::DN_NIM	高（タスク実行時に HTTP 通信が発生）	高（タスク実行時に NAND 書き込みが発生）
nn::ndm::DN_FRIENDS	低（断続的に UDP 通信が発生）	低

> **注意：** デーモンを制御する場合は、関数リファレンスに記載されている情報にも必ず目を通してください。

4.1. すれちがい通信

3DS のすれちがい通信は、ニンテンドーDS での同名の機能を強化し、より使いやすくしたものです。

バックグラウンドで通信を行いますので、起動中のアプリケーションや状態に関わらず、送信データを用意しておくだけでデータの送受信を行うことができます。ただし、アプリケーション独自の通信処理を実装することができないため、特殊な条件を必要とするような送受信方法を実現することはできない可能性があります。

> **注意:** すれちがい通信を動作させるためには EULA(End User License Agreement)の設定が必要です。CEC ライブラリを使用する場合は、Config ツールの「EULA Setting」で、「Agree Version」に 1.0 以上を設定してください。
>
> 開発用の本体(デバッガ、開発実機)と製品版の本体とでは、すれちがい通信は行われません。

4.1.1. 動作の概要

すれちがい通信は、すれちがいデーモン(バックグラウンドで動作する常駐プロセス)によって実際の通信処理を行っています。3DS 本体に用意されている領域にすれちがいボックスを作成し、そこに送信するメッセージとしてのすれちがいデータを登録することで、すれちがい通信が可能な状態になると、すれちがいデーモンが自動的に通信相手とすれちがいデータの送受信を行います。すれちがいデータは暗号化されて送信されますので、アプリケーションでデータの秘匿性を考慮する必要はありません。

すれちがいボックスは 12 個用意されており、原則的に 1 つのアプリケーションで 1 つのすれちがいボックスを占有することになります。すれちがい通信を行うためにすれちがいボックスを専有してデータを登録するときには、必ずユーザーの確認を取ってから行ってください。そのとき、12 個のすれちがいボックスがすべて埋まっていた場合は、「これ以上すれちがい通信をセットできません。本体設定のすれちがい通信管理で他のソフトの設定を消去してください」と表示し、本体設定でいらないすれちがい通信設定を消去する必要があることをユーザーに説明してください。不要なすれちがい通信設定をユーザーが消去するまで、すれちがいボックスを新たに作成することができません。

> **注意:** 同じタイトルの複数のゲームカードで 1 台の 3DS を共有している場合のすれちがい通信は、すべて同じすれちがいボックスに対して処理されることに注意が必要です。

すれちがいボックスをどのアプリケーションが使用しているかは本体設定の「すれちがい通信管理」の画面で確認することができ、すれちがいボックス単位で、すれちがい通信設定の消去を行うことができます。消去されたすれちがい通信設定のおしらせはおしらせリストに表示されなくなります。本体設定ですれちがいボックスが消去される可能性を考慮し、アプリケーションはボックス内のデータが保持されていなくても動作するように実装しなければなりません。

すれちがいボックス内には送信ボックスと受信ボックスがあり、それぞれ、送信するすれちがいデータを登録する領域、受信したすれちがいデータを保存する領域として確保されます。

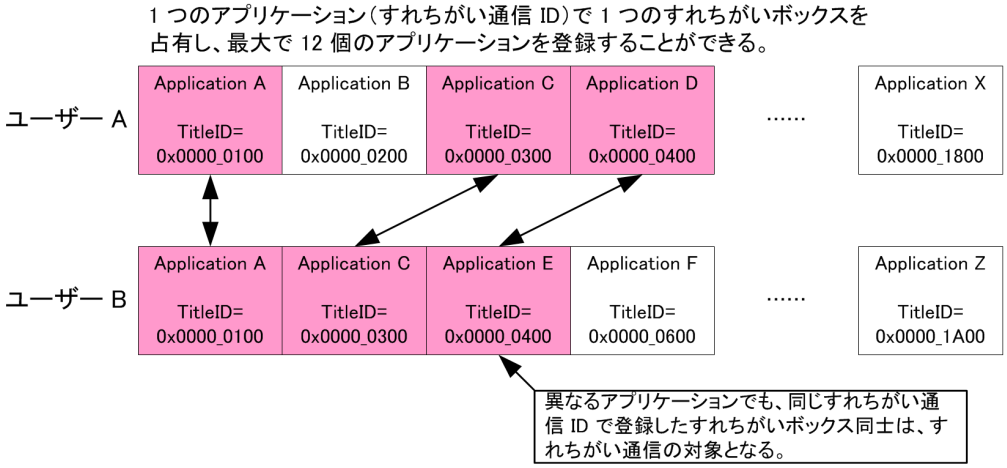
すれちがい通信可能な状態の本体同士が近づき、互いに相手を見つけると、最初に双方のシステム領域内にあるすれちがいボックスを走査します。そこで、通信相手にも同じアプリケーション(すれちがい通信 ID)で作成したすれちがいボックスがあれば、すれちがいデータの送受信が必要な分だけの通信が開始されます。すれちがい通信 ID が同じであれば、本体のリージョンが異なっていても通信が行われます。複数のすれちがいボックスからすれちがいデータが送信される場合は、そのサイズが小さいものから順に送信されます。すべてのすれちがいデータの送受信が完了するまで、双方が通信可能範囲に存在しているとは限りません。通信可能範囲に存在した時間が短い場合、最初の方に送信されるサイズの小さなデータだけが通信に成功する可能性があります。そのため、通信の成功率を上げるには、すれちがいデータのサイズをなるべく小さくすることが有効となります。また、1 つのすれちがいボックスからは、グループ化されていない限り、1 つのすれちがいデータが送信されます。

設定によっては受信のみを行うすれちがいデータを登録することもできますが、受信のみのデータ同士ではすれちがい通信が行われないことに注意してください。これは送信のみを行うすれちがいデータでも同じです。特に、最初に登録するデータが受信のみまたは送信のみで固定されている場合、すれちがい通信がいつまで経っても行われなくなってしまいます。

通信した相手のすれちがい通信用アドレス(8 時間ごとに再生成される)がアドレスフィルタに登録されるため、同じ相手とのすれちがい通信は平均 4 時間、最大 8 時間経過するまで行われません。アドレスフィルタは、アドレスフィルタをリセットしてすれちがい通信を開始する関数(デバッグ用途のみ)でクリアすることができます。製品版ではアプリケーションでアドレスフィルタの制御を行うことができませんので、すれちがい通信は、その場で設定したデータを目の前のユーザーに渡す用途には向いていません。

> **補足:** 接続設定で登録されているアクセスポイントや周囲のニンテンドーゾーンに接続している場合、すれちがい通信が行われません。アプリケーションの動作中にすれちがい通信が行われることを確認するときは、接続設定を解除し、周囲にニンテンドーゾーン等がない環境にしてください。
>
> ただし、アクセスポイントやニンテンドーゾーンに接続している場合でも、すれちがい通信専用モード(「4.1.3.7. すれちがい通信専用モード(デバッグ用途のみ)」)に切り替えたり、スリープするとすれちがい通信は行われます。

図 4-1. すれちがいボックスの走査



4.1.1.1. すれちがい通信に関する諸条件

すれちがい通信の開始や通信相手との接続、すれちがいデータの送信、受信データの破棄が行われる条件、おしらせランプが点灯する条件を示します。

すれちがい通信を開始する条件

以下の条件がすべて満たされていなければ、通信相手の検索が行われず、すれちがい通信は開始されません。

- 無線オンモードである
- ペアレンタルコントロールで、すれちがい通信が制限されていない
- ユーザーが EULA に同意している
- すれちがいボックスが存在し、送信可能回数が 1 以上のすれちがいデータが存在する

ニンテンドーゾーンの検索もすれちがいデーモンで行われるため、上記条件が満たされていなくても、デーモンの状態がスキャン中になることがあります。

通信相手と接続する条件

すれちがい通信の開始が可能な状態の本体同士が近付いたときに、以下の条件がすべて満たされていなければ、通信相手との接続が行われません。

- 通信相手にも、同じすれちがい通信 ID のすれちがいボックスが存在する
- 処理される順番にあるすれちがいデータ同士が、通信可能な送受信モードの組み合わせになっている
- 通信相手のすれちがい通信用アドレスが、アドレスフィルタ(通信した相手のすれちがい通信用アドレスのリスト)に登録されていない
- 通信相手が受信拒否リストに登録されていない

すれちがい通信用アドレスは 8 時間ごとに再生成されます。本体に保存されるアドレスフィルタには、最大 255 個のすれちがい通信用アドレスが登録(256 個目からは古いものに上書き)されます。すれちがいボックスに関係なく、登録されているすれちがい通信用アドレスの相手とのすれちがい通信は行われなくなります。そのため、新たにすれちがいボックスを作成しても、同じ相手とのすれちがい通信は平均 4 時間、最大 8 時間経過するまで行われません。

すれちがいデータを送信する条件

通信相手との接続が行われたあと、事前にデータのやり取りを行い、以下の条件がすべて満たされていなければ、すれちがいデータを相手に送信しません。

- 通信相手がフレンドであり、送信対象の設定にフレンドのみを対象とするフラグが含まれている。または、通信相手がフレンドではなく、送信対象の設定にフレンド以外を対象とするフラグが含まれている
- すれちがいデータを送信した場合に、通信相手の受信ボックスのデータサイズと個数の上限を超えない

送受信モードが「受信のみ」または「送受信」である場合、上記の条件を満たしていなくても、通信相手が上記の条件を満たしていればすれちがいデータの受信が開始されます。

条件を満たしていないために通信相手との送受信が開始されない場合は、この時点でアドレスフィルタにすれちがい通信用アドレスが登録されます。ただし、事前に行っているデータのやり取りに失敗している場合は登録されません。

送信または受信が開始された場合、すれちがいデータの送受信に成功したときにアドレスフィルタにすれちがい通信用アドレスが登録されます。送受信に失敗したときは登録されません。

受信したすれちがいデータを破棄する条件

以下の条件のいずれかを満たしている場合、受信したすれちがいデータは破棄されます。

- すれちがいデータの署名が正しくない
- 同じメッセージ ID のすれちがいデータが受信ボックスに存在する

おしらせランプが点灯する条件

いずれかのすれちがいボックスにデータを受信し、かつそのすれちがいデータが破棄されなかった場合におしらせランプが緑色に点灯します。

4.1.2. 初期化と終了

CEC ライブラリはすれちがい通信をアプリケーションで利用するためのライブラリです。ライブラリの初期化と終了は、`nn::cec::Initialize()` と `nn::cec::Finalize()` で行います。

コード 4-1. 初期化と終了

```
static nn::Result nn::cec::Initialize(nn::fnd::IAllocator& cecAllocFunc);
nn::Result nn::cec::Finalize();
```

すれちがい通信でライブラリが使用するメモリの管理は、アプリケーション独自のメモリアロケータで行わなければなりませ

ん。ライブラリは *cecAllocFunc* に渡されたアロケータから、すれちがいデータの最大サイズの 2 倍程度のメモリをデータ処理などのために確保します。

nn::cec::Initialize() の呼び出しが成功したあとに、再度 nn::cec::Initialize() を呼び出した場合は、初期化済みであることを示すエラー(nn::cec::MakeResultAlreadyInitialized())が返されます。

4.1.3. すれちがいボックス

すれちがいボックスへのアクセスは nn::cec::MessageBox クラスを介して行います。

4.1.3.1. アクセスの開始(オープンと作成)

すれちがいボックスへのアクセスを開始する前に、すれちがいボックスを作成可能か nn::cec::CTR::MessageBox::CanCreateMessageBox() で確認できます。作成可能な場合は nn::cec::MessageBox::CreateMessageBox() ですれちがいボックスを作成できます。すれちがいボックスは作成に成功した時点でオープンされています。目的のゲームタイトルのすれちがいボックスがすでに存在している場合は nn::cec::MessageBox::OpenMessageBox() でオープンしてアクセスを開始します。すれちがいボックスの登録数が上限に達している場合は作成することができません。

すれちがいボックスをオープンすると、すれちがいデーモンが停止状態になります。停止状態はすれちがいボックスをクローズするまで継続します。すれちがいボックスをオープンしたまま HOMEメニューを表示すると、デーモンは停止された状態ですので、すれちがい通信が発生しません。すれちがいボックスをオープンしたままスリープ状態に移行した場合はデーモンが動作状態になり、スリープ状態から復帰したときに停止状態になります。スリープ中にすれちがい通信が行われてすれちがいボックスの内容が更新されたとしても、オープンしていた MessageBox クラスには反映されず、最新の情報を取得するには再度オープンする必要があります。すれちがいボックスへのアクセス結果が巻き戻され、すれちがいボックスの作成やすれちがいデータの登録・削除といった処理がなかったことになる可能性がありますので、HOMEメニューの表示やスリープ状態へ移行する前に、オープンしていたすれちがいボックスはクローズまたはコミットすることを推奨します。

> **補足:** すれちがいボックスの作成やすれちがいデータの登録や削除は、クローズまたはコミットされるまで反映されません。そのため、すれちがいボックスへのアクセスは、オープンからクローズまでをひとまとめにして行うことを推奨します。

> **注意:** すれちがい通信が行われたかどうかをリアルタイムで検知する場合は、GetCecEvent()(「4.1.6. 通信発生の通知」参照)を利用してください。すれちがいボックスのオープンとクローズを定期的に繰り返すと、オープンの度に通信が中止されるため、バックグラウンドでの通信が行われなくなります。

コード 4-2. すれちがいボックスのオープン、作成

```
nn::cec::TitleId nn::cec::MakeCecTitleId(bit32 id, bit8 variation = 0x0);
nn::Result nn::cec::MessageBox::CanCreateMessageBox(const TitleId cecTitleId);
nn::Result nn::cec::MessageBox::OpenMessageBox(const TitleId cecTitleId,
                                               const u32 privateId);
nn::Result nn::cec::MessageBox::CreateMessageBox(
        const TitleId cecTitleId, const u32 privateId, const char* hmacKey,
        const void* icon, size_t iconSize, const wchar_t* name, size_t nameSize,
        size_t inboxSizeMax = CEC_INBOX_SIZE_DEFAULT,
        size_t outboxSizeMax = CEC_OUTBOX_SIZE_DEFAULT,
        size_t inboxMessNumMax = CEC_INBOX_MESSNUM_DEFAULT,
        size_t outboxMessNumMax = CEC_OUTBOX_MESSNUM_DEFAULT,
```

```
        size_t messageSizeMax = CEC_MESSSIZEMAX_DEFAULT);
```

cecTitleId には、任天堂から指定されたゲームタイトルを識別するための値(すれちがい通信 ID)を指定します。すれちがい通信 ID は業務部から割り当てられたユニーク ID(20 ビット)を引数に、nn::cec::MakeCecTitleId() を呼び出して生成された ID です。複数のタイトルで共通のすれちがい通信を行う場合は、そのうちのいずれかのユニーク ID を代表値として指定し、1 つのすれちがいボックスを共有することになります。

privateId は、すれちがいボックスへのアクセスに必要なキーとなる値です。この値は開発者が自由に決めることができます。この値の生成方法によっては、特定のセーブデータだけがアクセスできるようにしたり、特定のゲームカードだけがアクセスできるようにしたりすることが可能です。

上記の 2 つの引数の組み合わせが作成時のものと同じでなければ、すれちがいボックスはオープンすることができません。指定されたすれちがい通信 ID で作成されたすれちがいボックスが存在しない場合はデータなしのエラーが、すれちがいボックスは存在するが *privateId* が異なる場合はアクセス権なしのエラーが返されます。

hmacKey には、改竄防止のために使用される 32 バイトの文字列を指定します。文字列は開発者が自由に決めることができます。すれちがい通信の送信側、受信側ともに同じ文字列でなければなりません。つまり、同じすれちがい通信 ID ですれちがい通信を行うアプリケーションは同じ文字列を指定する必要があります。

icon と *iconSize* にはそれぞれ、すれちがい通信管理画面で表示されるアイコンのデータとデータサイズを指定します。アイコンのデータは、サイズが 48×48 ピクセル、ピクセルフォーマットが RGB565 の PICA ネイティブフォーマットのテクスチャでなければなりません。通常はそのゲームタイトルのアイコンを使用しますが、シリーズ共通の場合はシリーズであることが判別できるアイコンを使用することができます。

name と *nameSize* にはそれぞれ、すれちがい通信管理画面で表示されるタイトル名とその長さ(バイトサイズ)を指定します。タイトル名は、最大 64 文字(終端文字を含む)の文字列で指定します。ただし、実際に表示されるのは最大幅の内蔵フォント(日米欧リージョンは "%"、そのほかのリージョンはひらがな、漢字、ハングルなど)で 17 文字分の幅です。なお、タイトル名の言語は本体の言語設定に合わせても構いませんし、アプリケーション自身の言語設定に合わせても構いません。

inboxSizeMax と *outboxSizeMax* にはそれぞれ、受信ボックスと送信ボックスの容量を指定します。2 つのボックスの合計サイズは 1 MByte 以内でなければなりません。すれちがいデータを伝播させる場合を除いて、受信ボックスに大きな容量を割り当てるのが基本です。引数の指定を省略したときは、デフォルトの 512 KByte を指定したことになります。

inboxMessNumMax と *outboxMessNumMax* には、受信ボックスと送信ボックスに保存するすれちがいデータの数をそれぞれ指定します。ボックスの容量を一番大きなすれちがいデータのサイズで割った値を指定してください。引数の指定を省略したときは、デフォルトの 99 を指定したことになります。

messageSizeMax には、すれちがいデータの最大サイズを指定します。指定された値よりも大きなサイズのすれちがいデータは保存することができません。通常は引数の指定を省略し、デフォルトの 100 KByte を指定してください。ただし、現在の CTR-SDK では *messageSizeMax* の指定に関わらず、すれちがいデータの最大サイズが 100 KByte 固定となります。

nn::cec::MessageBox::OpenMessageBox() が返す可能性のあるエラーとその対処を下表に示します。

表 4-2. OpenMessageBox() が返す可能性のあるエラーとその対処

エラー	原因と対処
nn::cec::ResultNoData()	すれちがいボックスが存在しませんので、すれちがいボックスを作成してから、再度オープンしてください。
nn::cec::ResultNotAuthorized()	*privateId* が一致しないためオープンできません。すれちがいボックスを再作成する場合は、削除する必要があります。

nn::cec::ResultStateBusy()	複数のスレッドからアクセスしなければ返されないエラーです。時間を置いてリトライすることでオープンすることができるようになります。
上記以外	このエラーが返ってきた場合は失敗として扱い、この関数を使わずにアプリケーションを進行させる必要があります。アプリケーションの進行にこの関数が必須である場合は、FATAL エラーとして表示しても構いません。

nn::cec::MessageBox::CreateMessageBox() が返す可能性のあるエラーとその対処を下表に示します。

表 4-3. CreateMessageBox() が返す可能性のあるエラーとその対処

エラー	原因と対処
nn::cec::MakeResultBoxAlreadyExists()	同じ *cecTitleId* と *privateId* で作成されたすれちがいボックスがすでに存在しています。 nn::cec::MessageBox::OpenMessageBox() で nn::cec::ResultNoData() が返されている場合は、*cecTitleId* の variation のみが異なるすれちがいボックスが存在しています。
nn::cec::MakeResultBoxNumFull()	すでに 12 個のすれちがいボックスが作成されています。新規に作成することができませんので、すれちがい通信管理画面で不要なすれちがいボックスを削除するように、ユーザーを案内してください。
nn::cec::ResultStateBusy()	複数のスレッドからアクセスしなければ返されないエラーです。時間を置いてリトライすることで作成することができるようになります。
nn::cec::ResultInvalidArgument()	引数の指定に間違いがあります。
nn::cec::ResultTooLarge() nn::cec::ResultInvalidData()	アイコン・名称の指定に間違いがあります。
上記以外	このエラーが返ってきた場合は失敗として扱い、この関数を使わずにアプリケーションを進行させる必要があります。アプリケーションの進行にこの関数が必須である場合は、FATAL エラーとして表示しても構いません。

4.1.3.2. 付随データの設定

すれちがい通信管理画面で表示されるときに使用されるアイコンとタイトル名を変更することができます。すれちがいボックスをオープンし、変更するアイコンやタイトル名をすれちがいボックスの付随データとして設定してください。

コード 4-3. 付随データの設定

```
nn::Result nn::cec::MessageBox::SetMessageBoxData(
                        u32 datatype, const void* data, size_t dataSize);
```

datatype には、付随データの種類を指定します。アイコンのデータならば nn::cec::BOXDATA_TYPE_ICON を、タイトル名ならば nn::cec::BOXDATA_TYPE_NAME_1 を指定してください。*data* と *dataSize* には、付随データの先頭アドレスとそのサイズを指定します。

アイコンデータは、サイズが 48×48 ピクセル、ピクセルフォーマットが RGB565 の PICA ネイティブフォーマットのテクスチャでなければなりません。通常はそのゲームタイトルのアイコンを使用しますが、シリーズ共通の場合はシリーズであることが判別できるアイコンを使用することができます。

タイトル名は、最大 64 文字(終端文字を含む)の文字列で指定します。実際に表示されるのは最大幅の内蔵フォント(日米欧リージョンは "%"、そのほかのリージョンはひらがな、漢字、ハングルなど)で 17 文字分の幅です。なお、タイトル名の言語は本体の言語設定に合わせても構いませんし、アプリケーション自身の言語設定に合わせても構いません。

4.1.3.3. 送信ボックス / 受信ボックスの情報

作成時に指定した容量やメッセージの最大保存数、現時点でのボックスの使用量、保存されているすれちがいデータの数などの情報を取得することができます。

コード 4-4. 送信ボックス / 受信ボックスの情報の取得

```
u32 nn::cec::MessageBox::GetBoxSizeMax(nn::cec::CecBoxType boxType) const;
u32 nn::cec::MessageBox::GetBoxSize(nn::cec::CecBoxType boxType) const;
u32 nn::cec::MessageBox::GetBoxMessageNumMax(nn::cec::CecBoxType boxType) const;
u32 nn::cec::MessageBox::GetBoxMessageNum(nn::cec::CecBoxType boxType) const;
```

これらの関数のように、送信ボックスか受信ボックスかを指定する必要のあるものは、nn::cec::CecBoxType 列挙子を引数にとります。CEC_BOXTYPE_INBOX が受信ボックスの、CEC_BOXTYPE_OUTBOX が送信ボックスの指定に使われます。

GetBoxSizeMax() と GetBoxSize() では、指定したボックスの容量と使用量を取得することができます。

GetBoxMessageNumMax() と GetBoxMessageNum() では、指定したボックスに保存することのできるすれちがいデータの数と現時点で保存されているすれちがいデータの数を取得することができます。

これらの情報のうち、現時点で保存されているすれちがいデータの数でループを組み、nn::cec::MessageBox クラスの GetMessageId() でインデックスに対応するメッセージ ID を取得することで、ボックス内のデータを走査することができます。

コード 4-5. インデックス指定によるメッセージ ID の取得

```
nn::Result nn::cec::MessageBox::GetMessageId(MessageId* messId,
                        const CecBoxType boxType, const u32 messIndex);
```

取得したメッセージ ID を引数に ReadMessage() を呼び出してすれちがいデータを取得し、nn::cec::Message クラスを介してアクセスすることができるようになります。

コード 4-6. すれちがいデータの取得

```
u32 nn::cec::MessageBox::GetMessageSize(const CecBoxType boxType,
                                        const u32 messIndex) const;
nn::Result nn::cec::MessageBox::ReadMessage(
                nn::cec::Message& cecMessage, void* buf, const size_t bufLen,
                const CecBoxType boxType, const MessageId& messageId);
```

cecMessage には、nn::cec::Message クラスのインスタンスを渡してください。

buf と *bufLen* には、GetMessageSize() で取得したサイズのバッファを指定しなければなりません。これは、一度バイナリ配列として読み込んでからインスタンスに情報を設定するためです。

nn::cec::Message クラスを介した、すれちがいデータへのアクセスについては「4.1.4.3. 情報の取得」を参照してください。

4.1.3.4. 削除

すれちがいボックスは 12 個しかない有限のリソースですので、すれちがいデータを 1 つも登録していない状態やユーザーがすれちがい通信を行わないときはすれちがいボックスを削除し、ほかのアプリケーションがすれちがいボックスを作成できるようにしてください。

コード 4-7. すれちがいボックスの削除

```
nn::Result nn::cec::MessageBox::DeleteMessageBox();
nn::Result nn::cec::MessageBox::DeleteMessageBox(const TitleId cecTitleId);
```

cecTitleId には、削除するすれちがいボックスのすれちがい通信 ID を指定します。引数の指定のない関数では、そのクラスでオープンしているすれちがいボックスが削除されます。

> **注意:** アプリケーションで作成したすれちがいボックス以外を削除しないでください。
>
> すれちがい通信 ID を指定したすれちがいボックスの削除は、すれちがいボックスの破損などが原因でオープンできなくなったときにのみ利用してください。

4.1.3.5. アクセスの終了（クローズとコミット）

すれちがいボックスへのアクセスが終了したあとは、必ずデーモンを動作状態に戻さなければなりません。動作状態に戻すには、nn::cec::MessageBox::CloseMessageBox() を呼び出して、オープンしていたすれちがいボックスをクローズします。

コード 4-8. すれちがいボックスのクローズ

```
void nn::cec::MessageBox::CloseMessageBox();
```

すれちがいボックスへの変更は、コミットが行われるまで保存領域に書き込まれません。そのため、コミットを行う前にアプリケーションを終了すると、前回コミットされた時点のデータに巻き戻ってしまう可能性があります。コミットを行うには nn::cec::MessageBox::CommitMessageBox() を呼び出しますが、すれちがいボックスのクローズ処理の中でもコミットが行われています。

コード 4-9. すれちがいボックスのコミット

```
nn::Result nn::cec::MessageBox::CommitMessageBox();
```

4.1.3.6. すれちがいデーモンとの関係

アプリケーションですれちがいボックスをオープンすると、すれちがいデーモンは停止状態になります。このとき、デーモンが処理を行っている途中であった場合は処理がキャンセルされ、受信したデータを破棄してしまう可能性があります。そのため、すれちがいボックスをオープンする前に nn::cec::CecControl::GetCecState() でデーモンが処理中（DAEMON_STATE_BUSY）や通信中（DAEMON_STATE_COMMUNICATING）でないことを確認することで、安全にすれちがいボックスへのアクセスを行うことができます。

4.1.3.7. すれちがい通信専用モード（デバッグ用途のみ）

デバッグ用途での使用に限定されていますが、以下の関数でバックグラウンド通信をすれちがい通信専用モードに切り替えることができます。

コード 4-10. すれちがい通信専用モードへの切り替え

```
static nn::Result nn::cec::CecControl::EnterExclusiveState();
static nn::Result nn::cec::CecControl::LeaveExclusiveState();
```

すでにバックグラウンド通信が排他的に使用されているときやローカル通信を行っているときは、すれちがい通信専用モードに切り替える際にエラーとなります。

4.1.4. すれちがいデータ

すれちがいデータへのアクセスは nn::cec::Message クラスを介して行います。

4.1.4.1. 新規作成

すれちがいデータを新規に作成するには、nn::cec::Message クラスのインスタンスを用意し、メンバ関数の NewMessage() を呼び出さなければなりません。

コード 4-11. すれちがいデータの新規作成

```
nn::Result nn::cec::Message::NewMessage(TitleId cecTitleId, u32 groupId,
                        u8 messageTypeFlag, u8 sendMode,
                        u8 sendCount, u8 propagationCount);
nn::Result nn::cec::Message::NewMessage(TitleId cecTitleId, u32 groupId,
                        u8 messageTypeFlag, u8 sendMode,
                        u8 sendCount, u8 propagationCount,
                        const void* icon, size_t iconSize,
                        const wchar_t* infoTextData, size_t infoTextSize);
```

cecTitleId には、すれちがいボックスをオープンしたときに指定したすれちがい通信 ID を指定します。

groupId には、すれちがいデータをグループ化する場合に、そのグループ番号を指定します。グループ化する場合は、同じグループで登録されたすれちがいデータの合計サイズが、1 回の通信で送信可能な最大サイズ(100 KByte)を超えないように注意しなければなりません。グループ番号に 0 を指定した場合はグループ化が行われず、グループ番号が 0 のすれちがいデータがほかに存在していても個別に送受信が行われます。

messageTypeFlag には、送信対象をフラグの論理和で指定します。フラグの組み合わせで、すれちがいデータを送信する相手をフレンドかフレンドでないかに限定、もしくは両方を指定することができます。この設定によって、フレンドとの通信とフレンドではない相手との通信で異なるすれちがいデータを送信することができ、グループ内で設定の異なるすれちがいデータを登録すると、フレンドには追加でコメントを送信するなどの制御が可能です。このフラグの指定は送信が行われるときにのみ考慮されます。フレンドであるかどうかを受信の条件にすることはできません。

表 4-4. 送信対象を設定するフラグ

フラグ	説明
下記 2 つのフラグの論理和を指定	フレンドであるかどうかを区別せずに送信する場合に設定します。 通常はこの設定を使用し、誰にでも送信するように設定します。
MESSAGE_TYPEFLAG_NON_FRIEND	通信相手がフレンドではない場合にのみ送信します。 同じグループのすれちがいデータで、フレンドとフレンド以外に対して異なるコメントを送信する場合などに設定します。
MESSAGE_TYPEFLAG_FRIEND	通信相手がフレンドである場合にのみ送信します。 誰にでも送信するすれちがいデータと同じグループに含めて、フレンドに対して送信したときだけコメントを追加する場合などに設定します。

注意: MESSAGE_TYPEFLAG_FRIEND のみを設定する場合は、すれちがい通信中継での処理の違いがアプリケーションの仕様上問題ないかを確認してください。詳細については、「9. 付録:すれちがい通信中継」を参照してください。

sendMode には、すれちがいデータの送受信を制御する送受信モードを指定します。通信相手の送受信モードとの組み合わせによって、通信時にすれちがいデータの送信と受信が行われるのかが決定します。

表 4-5. 送受信モード

送受信モード	説明
CEC_SENDMODE_RECV	受信のみ。通信相手が「送信のみ」、「送受信」ならば受信します。 同じ設定の相手とはすれちがい通信が行われないため、設定を推奨していません。 ※ すれちがいデータは送信ボックスに登録します。
CEC_SENDMODE_SEND	送信のみ。通信相手が「受信のみ」、「送受信」ならば送信します。 同じ設定の相手とはすれちがい通信が行われないため、設定を推奨していません。
CEC_SENDMODE_SENDRECV	送受信。通信相手が「受信のみ」ならば送信、「送信のみ」ならば受信、「送受信」ならば送信と受信を行います。 自作データの配布やゲーム内ハウスへの招待など、一方通行（送信だけ、もしくは受信だけ）の通信でもよい場合に設定します。
CEC_SENDMODE_EXCHANGE	交換。通信相手が同じ「交換」ならば、送信と受信を行います。双方で送信と受信が正しく行われなければ、すれちがい通信が成立しません。 ゲーム内ペットの交流やすれちがいでの自動対戦など、通信したデータ同士を特定する必要がある場合に設定します。

表 4-6. 送受信モードの組み合わせによって行われる送受信

A（縦） B（横）	受信のみ	送信のみ	送受信	交換
受信のみ	×	A ← B	A ← B	×
送信のみ	A → B	×	A → B	×
送受信	A → B	A ← B	A ⇔ B	×
交換	×	×	×	A ⇔ B

相手にすれちがいデータが問題なく送信されたかどうかは保証されません。送受信モードが異なっていてもグループ化されるため、同じグループに属するデータの送受信モードをなるべく統一し、送受信モードが「受信のみ」のデータを含めないようにしてください。

> **注意：** 送受信モードの「交換」は、同じ「交換」に設定された相手とのみ通信が発生します。同じすれちがいボックスに、「交換」とそのほかの設定（「送信のみ」、「受信のみ」、「送受信」）が混在すると、すれちがい通信が発生する確率が下がります。

sendCount には、すれちがいデータの送信可能回数を指定します。nn::cec::MESSAGE_SENDCOUNT_UNLIMITED を指定した場合は送信回数を制限しません。送信が行われると送信可能回数が 1 減少し、0 になると送信されなくなります。受信側が受け取ったデータを受信ボックスに登録したかどうかに関係なく、送信可能回数はデータを送信した時点で減少します。データを送信したときに減少しますので、送信モードが「送受信」の場合でも、相手の送信モードが「送信のみ」ならば減少しません。また、受信可能回数の限界に達しているなど、通信相手のボックスに空きがないときにはデータが送信されず、送信可能回数は減少しません。ただし、自分の受信ボックスに空きがあれば、相手からデータを受信する可能性はあります。送受信モードが「受信のみ」（CEC_SENDMODE_RECV）のときは受信回数の制限として機能し、すれちがいデータを受信するたびに 1 減少して 0 になるまでデータを受信します。

同じすれちがいデータを受信したときは、新しい方のすれちがいデータが破棄されます。すれちがいデータが破棄された場合でも、送信した側の *sendCount* は 1 減少します。

送信されたデータが相手に到達するかどうかは保証されていません。また、すれちがい通信が成立した直後に、すれちがいデータの保存や送信可能回数の減少処理の途中で電源が切断されたりすると、送信可能回数の残りと受信したデータの個数との整合性が取れなくなる場合があります。例えば、送信が完了するのとほぼ同時に通信が切れた場合、タイミングによっては下記のようなケースが発生することがあります。

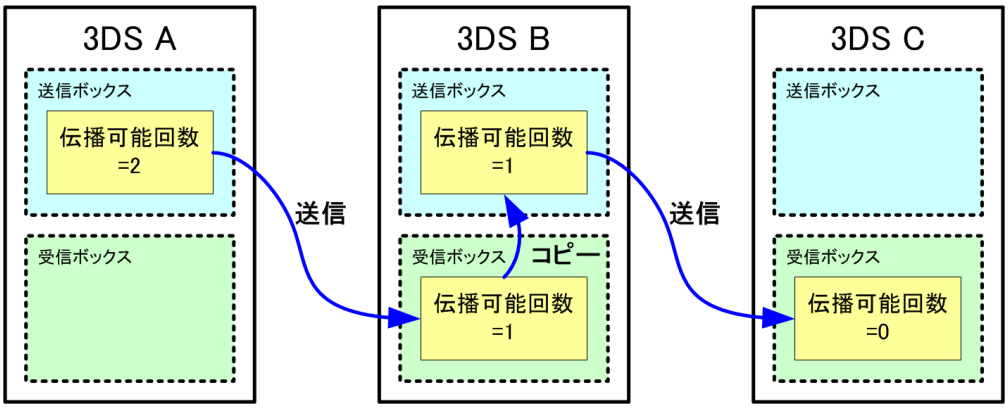
- 送信側は送信に失敗、受信側は受信に成功する。この場合、送信側の送信可能回数が減っていないのにすれちがいデータが受信されている。
- 送信側は送信に成功、受信側は受信に失敗する。この場合、送信側の送信可能回数が減っているのにすれちがいデータは受信されていない。

> **補足:** アドレスフィルタは、その本体で送信または受信が1件でも成功したときに登録されます。
>
> すれちがい通信した二つの端末で通信が途切れたために片方は送信は成功・受信は失敗、もう片方は受信は成功・送信は失敗した場合、送受信されなかったデータがまだほかに残っていてもアドレスフィルタはお互いに登録されます。
>
> この場合、他のタイトルデータのすれちがい通信が行われていても、期待するタイトルデータのすれちがい通信が行われないままアドレスフィルタにより通信が制限され、その端末同士でのすれちがい通信が再開可能になるまで時間がかかるということが起こりえます。

propagationCount には、すれちがいデータの伝播可能回数を指定します。伝播可能回数は受信時に 1 減少し、1 以上ならば受信したデータを送信ボックスにコピーします。つまり、2 以上を指定した場合にすれちがいデータの伝播が行われることになります。受信したデータが受信ボックスだけでなく送信ボックスにも保存されることや、伝播経路によっては自分が送信したすれちがいデータを受信する可能性があることに注意してください。データの爆発的な広がりを防ぐため、*sendCount* または *propagationCount* のどちらかには必ず 1 を設定しなければならないように、ライブラリで制限されています。

伝播可能回数の設定されたすれちがいデータを受け取ったときに、受信ボックスに同じデータが存在している場合は新しいデータが破棄されるため伝播は行われません。受信ボックスに同じデータが存在せず、データを受信した場合でも送信ボックスの容量が足りなければ伝播は行われません。ただし、以前に同じデータを受け取って送信ボックスにコピーしていた場合は、新しいデータの伝播可能回数と送信可能回数(必ず 1)に更新されるため伝播は継続します。

図 4-2. 伝播可能回数に 2 を設定したときの伝播の様子



`nn::cec::Message::NewMessage()` では、主に送受信の条件を設定しました。ほかにも、おしらせに表示される情報(アイコン、説明文)や送信するデータを設定しなければなりません。この 2 つの情報は、引数 *icon*, *iconSize*, *infoTextData*, *infoTextSize* を持つ `nn::cec::Message::NewMessage()` のオーバーロードならば、新規作成時に設定することができます。*infoTextData* に設定する説明文の言語は本体の言語設定に合わせても構いませんし、アプリケーション自身の言語設定に合わせても構いません。ただし、設定可能なテキストは 1 言語分だけであるため、送

信者と受信者で言語設定が異なる場合に、受信者側の設定と異なる言語の説明文が表示される可能性があります。

おしらせに表示される情報は拡張ヘッダ情報として、nn::cec::Message::SetExHeader() で設定します。

コード 4-12. 拡張ヘッダ情報の設定

```
nn::Result nn::cec::Message::SetExHeader(
        const u32 exhType, const size_t exhLen, const void* exhBody);
```

exhType には、拡張ヘッダ情報の種類を指定します。

表 4-7. 拡張ヘッダの種類

種類	説明
MESSAGE_EXHEADER_TYPE_ICON	アイコン。サイズが 40×40 ピクセル、ピクセルフォーマットが RGB565 の PICA ネイティブフォーマットのテクスチャ。
MESSAGE_EXHEADER_TYPE_INFO	説明文。最大幅の内蔵フォント(日米欧リージョンは ″%″、そのほかのリージョンはひらがな、漢字、ハングルなど)16 文字分の幅で 2 行まで表示。改行と終端文字を含めて最大 128 文字。文字コードは UTF16-LE。

exhBody と *exhLen* には、拡張ヘッダ情報のデータとそのサイズを指定します。

アイコンのデータは nn::cec::Message::SetIcon() の呼び出しで、説明文のデータは nn::cec::Message::SetInfoText() の呼び出しでも設定することができます。

すれちがい通信で送信したいデータは nn::cec::Message::SetMessageBody() で設定します。

コード 4-13. 送信データの設定

```
nn::Result nn::cec::Message::SetMessageBody(
                        const void* dataBody, const size_t size);
```

dataBody と *size* には、送信したいデータのデータとそのサイズを指定します。サイズは nn::cec::Message::MESSAGE_BODY_UNITSIZE (4) の倍数でなければなりません。

送信条件などのヘッダ情報、拡張ヘッダ情報、送信データの合計は、すれちがいボックスの作成時に指定したすれちがいデータの最大サイズ(デフォルトは 100 KByte)以内でなければなりません。送信データの最大サイズは nn::cec::Message::MESSAGE_BODY_SIZE_MAX(96 KByte)です。ただし、すれちがいデータのサイズが大きくなると、すれちがいボックス間での送信順で後回しになる、送信にかかる時間が長くなるなど、通信の成功率が下がる可能性があります。

受信のみのすれちがいデータには、拡張ヘッダ情報と送信データを設定する必要はありません。

すれちがいデータにはほかにも、nn::cec::Message::SetTag() で任意の 16 ビットのデータを設定することができます。すれちがいデータを読み込まずにヘッダ情報へのアクセスで取得できますので、データ種別の判定など、自由に使用してください。

4.1.4.2. 送信ボックスへの登録

すれちがいデータをすれちがい通信で送受信するためには、nn::cec::MessageBox クラスのメンバ関数 WriteMessage() で送信ボックスに保存する必要があります。

コード 4-14. ボックスへのすれちがいデータの保存

```
nn::Result nn::cec::MessageBox::WriteMessage(const nn::cec::Message& cecMessage,
                                             const CecBoxType boxType, MessageId& messageIdOut);
```

cecMessage には、作成したすれちがいデータを示す nn::cec::Message クラスのインスタンスを指定します。

boxType には通常、送信ボックス(CEC_BOXTYPE_OUTBOX)を指定します。アプリケーションで受信ボックスを指定するような操作が必要になることはまずありません。

messageIdOut には、この関数の呼び出しですれちがいデータが保存されたときに、すれちがいデータを識別するためのメッセージ ID が書き込まれます。

すれちがいデータの保存時に EULA の同意チェックが行われますので、事前に FS ライブラリの初期化が必要です。すれちがいデータの保存時に、ユーザーが利用規約に同意していない場合やアプリケーションにアイコンファイルが設定されていない場合は EULA 非同意(ResultNotAgreeEula)のエラーが、ペアレンタルコントロールですれちがい通信が制限されている場合はペアレンタルコントロールによる制限(ResultParentalControlCec)のエラーが返されます。

> **補足:** すれちがいデータの最大サイズは 100 KByte 固定となりました。そのため、データのサイズが 100 KByte を超えなければ、サイズ超過(ResultMessTooLarge)のエラーは返されません。

送信ボックスへの登録順と処理の順番について

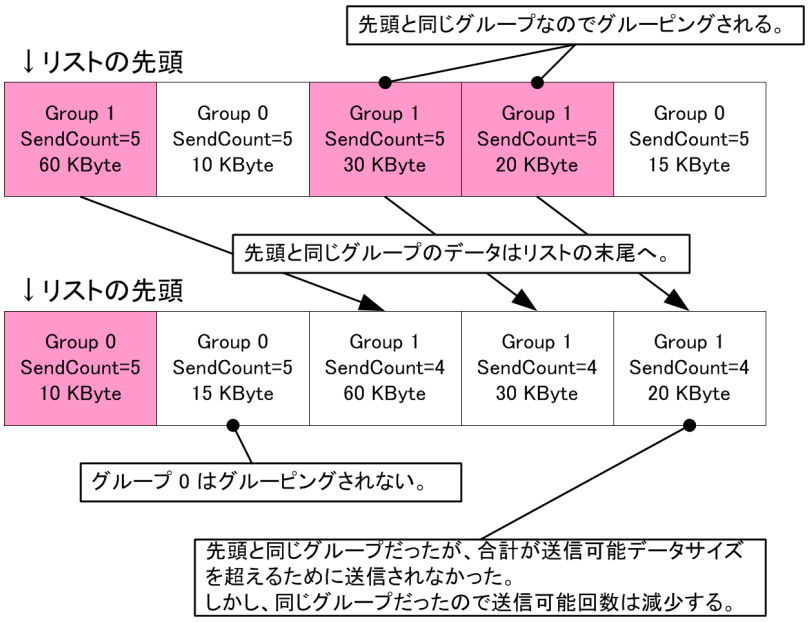
すれちがいボックス内では、送信ボックスに登録された順番でリストが作成されます。基本的に、このリストの先頭のデータが処理されますが、先頭のデータの送信可能回数が 0 になっていた場合は、リストの順番で次にくるデータが処理の対象となります。処理の対象となったデータの送受信モードにより、実際に送受信が行われるかどうかが決定されます。そのため、送受信モードが「受信のみ」に設定されているデータがグループに含まれていると、通信相手にそのデータが送信されてしまいます。同じ送受信モードのデータだけでグループ化されるように、データの置き方を考慮してください。

処理対象のデータのグループ ID が 0 以外に設定されていると、リスト内に同じグループのデータがあればリストの末尾に到達するか、データの合計サイズが 1 回の送信で送信可能な最大サイズになるまでのデータが一緒に処理されます。先頭と同じグループのデータはリストの末尾に回され、リストの先頭のデータが入れ替わります。このとき、処理されなかったデータを含めて、処理されたデータと同じグループのデータすべての送信可能回数が 1 減少します。

処理対象のデータのグループ ID が 0 に設定されている場合はそのデータのみが処理されます。ほかにグループ ID が 0 に設定されているデータが存在していてもグループとはみなされません。処理されたデータのみがリストの末尾に回され、送信可能回数が 1 減少します。

新たにすれちがいデータを登録した場合は、リストの末尾に追加されます。また、受信したデータの中に減少後の伝播可能回数が 1 以上のデータが存在した場合は、そのデータがリストの末尾(送信ボックス)に追加され、以降のすれちがい通信時に別の本体から来たデータを送信することになります。

図 4-3. グループの設定と処理の順番



4.1.4.3. 情報の取得

受信ボックスを走査して取得した受信データにアクセスするために、nn::cec::Message クラスを介して必要な情報を取得します。

コード 4-15. 情報の取得

```
u32 nn::cec::Message::GetBodySize() const;
u32 nn::cec::Message::GetMessageBody(void* dataBody, size_t size) const;
MessageId nn::cec::Message::GetMessageId_Pair(MessageId* messIdPair) const;
u16 nn::cec::Message::GetTag() const;
```

すれちがいデータに設定された送信データを取得するには、メンバ関数の GetMessageBody() を呼び出します。

dataBody と *size* には、送信データを格納するバッファとそのサイズを指定します。必要なバッファのサイズは GetBodySize() で取得してください。

すれちがいデータの送受信モードに「交換」(CEC_SENDMODE_EXCHANGE)が設定されていた場合は、GetMessageId_Pair() で交換相手となった送信データのメッセージ ID を取得してアクセスしてください。

GetTag() はすれちがいデータに設定されたタグを取得することができます。

4.1.4.4. 削除

送受信されたすれちがいデータは自動的に削除されることはありません。送信ボックス、受信ボックスともに保存可能な容量やデータ数には限りがありますので、送信可能回数が 0 になってしまった送信データや、読み込んでアプリケーションに反映し終わった受信データなど、不要になったすれちがいデータはアプリケーションで削除しなければなりません。すれちがいデータの削除は nn::cec::MessageBox クラスのメンバ関数 DeleteMessage() で行います。すれちがいボックス内のすべてのすれちがいデータを削除する場合は、nn::cec::MessageBox クラスのメンバ関数 DeleteAllMessages() を呼び出してください。

コード 4-16. すれちがいデータの削除

```
nn::Result nn::cec::MessageBox::DeleteMessage(
                const CecBoxType boxType, const MessageId& messageId);
nn::Result nn::cec::MessageBox::DeleteAllMessages(const CecBoxType boxType);
```

削除するすれちがいデータのメッセージ ID と、保存されているボックスの種類を *messageId* と *boxType* で指定してください。

すれちがいボックス内に 2 つ以上のすれちがいデータが存在し、そのすべてを削除する場合、DeleteMessage() で個別に削除するよりも、DeleteAllMessages() で削除する方が処理にかかる時間が短くなります。

4.1.4.5. 情報の更新

すれちがいボックス内のすれちがいデータの情報は nn::cec::Message クラスを介して更新することができます。

nn::cec::MessageBox クラスのメンバ関数 ReadMessage() で更新対象のすれちがいデータを読み込み、取得できた nn::cec::Message クラスのインスタンスから更新する属性値を設定した後、そのインスタンスを nn::cec::MessageBox クラスのメンバ関数 WriteMessage() ですれちがいボックスに書き込むことで情報が更新されます。

書き込み後に nn::cec::MessageBox クラスのメンバ関数 CommitMessageBox() または CloseMessageBox() を実行することで、更新した情報がすれちがいボックス内に確実にコミットされます。

以下に挙げるようなメンバ関数を使用することで、すれちがいデータの属性値が更新できます。

コード 4-17. すれちがいデータの設定

```
nn::Result nn::cec::Message::SetGroupID(u32 groupId);
nn::Result nn::cec::Message::SetMessageTypeFlag(MessageTypeFlag messTypeFlag);
nn::Result nn::cec::Message::SetSendMode(SendMode sendMode);
void nn::cec::Message::SetTag(u16 tag);
```

> **補足:** すれちがいボックス内から取得した nn::cec::Message オブジェクトの一部属性(拡張ヘッダ情報など)は更新しようとしても編集不可(ResultNotAuthorized)のエラーが返されます。
>
> 詳細は各属性の設定関数リファレンスを参照してください。

4.1.5. ヘッダ情報へのアクセス

nn::cec::MessageBox クラスでは、すれちがいデータのヘッダ情報にだけアクセスし、実際にデータを開かなくても、ある程度の情報を引き出すことができます。ただし、送信データや拡張ヘッダに設定されている情報にはアクセスすることができません。

すれちがいデータの情報のうち、すれちがいデータの総サイズなどを以下のメンバ関数で取得することができます。

コード 4-18. すれちがいデータへのアクセス

```
u32 nn::cec::MessageBox::GetMessageSize(
                        const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageBodySize(
                        const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageGroupId(
                        const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageSessionId(
                        const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessageTypeFlag(
                        const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessageSendMode(
                        const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessageSendCount(
                        const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessagePropagationCount(
                        const CecBoxType boxType, const u32 messIndex) const;
bit16 nn::cec::MessageBox::GetMessageTag(
                        const CecBoxType boxType, const u32 messIndex) const;
nn::fnd::DateTimeParameters nn::cec::MessageBox::GetMessageSendDate(
                        const CecBoxType boxType, const u32 messIndex) const;
nn::fnd::DateTimeParameters nn::cec::MessageBox::GetMessageRecvDate(
                        const CecBoxType boxType, const u32 messIndex) const;
nn::fnd::DateTimeParameters nn::cec::MessageBox::GetMessageCreateDate(
                        const CecBoxType boxType, const u32 messIndex) const;
MessageId nn::cec::MessageBox::GetMessageIdPair(
                        const CecBoxType boxType, const u32 messIndex) const;
nn::Result nn::cec::MessageBox::GetMessageIdPair(
        MessageId* messId, const CecBoxType boxType, const u32 messIndex) const;
MessageId nn::cec::MessageBox::GetMessageId(
                        const CecBoxType boxType, const u32 messIndex) const;
nn::Result nn::cec::MessageBox::GetMessageId(
        MessageId* messId, const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageIndex(
                        CecBoxType boxType, const MessageId& messId) const;
u32 nn::cec::MessageBox::GetMessageIndex(CecBoxType boxType, u8* messId) const;
```

4.1.6. 通信発生の通知

すれちがいデータの通信が行われたことを示す通知をイベントで受け取ることができます。また、最新の受信データについての情報を取得することもできます。

コード 4-19. 通信発生の通知を受け取るイベント、受信データの取得

```
nn::Result nn::cec::GetCecEvent(nn::os::Event* event);
nn::Result nn::cec::GetCecInfoBuffer(
                        u32 cecTitleId, u8 pCecInfoBuffer[], size_t size);
```

event に指定する nn::os::Event クラスのインスタンスには、初期化されていないものを指定してください。このイベントは、イベントを取得したアプリケーションが登録したすれちがいボックスにすれちがいデータを受信したときだけでなく、ほかのアプリケーションが登録したすれちがいボックスに受信したときや、すれちがいデータの送信が行われたときにもシグナル状態になることに注意してください。このイベントによる通知は、アプリケーションの動作中にバックグラウンドで行われたすれ

ちがい通信で、すれちがいデータを受信したことを即座に知りたい場合に利用します。

最新の受信データの情報は nn::cec::GetCecInfoBuffer() で取得することができます。この関数は、直前に行われたすれちがい通信についての情報を *pCecInfoBuffer* に指定されたバッファに格納します。通常、上記のイベントですれちがい通信が行われたことを検知したあと、自分のすれちがいボックスにすれちがいデータが受信されたかどうかを調べるために、*cecTitleId* を指定してこの関数を呼び出します。また、この関数は、すれちがい通信のデーモンプロセスを停止状態にしなくても呼び出すことができます。

格納される情報は nn::cec::CecNotificationData 構造体ですので、*pCecInfoBuffer* と *size* には構造体を格納できるだけのバッファとサイズを指定してください。

nn::cec::CecNotificationData 構造体の num メンバには構造体に格納された情報の数が、count メンバには本体が起動されてから発生したすれちがい通信の回数が格納されています。param メンバには、先頭から num 番目までの nn::cec::CecNotificationParam 構造体にすれちがいデータについての情報が格納されています。

nn::cec::CecNotificationParam 構造体の recvDate メンバには、すれちがいデータを受信した日時が格納されています。cecTitleId メンバにはすれちがいボックスのすれちがい通信 ID が、messageId メンバにはすれちがいデータのメッセージ ID が格納されています。

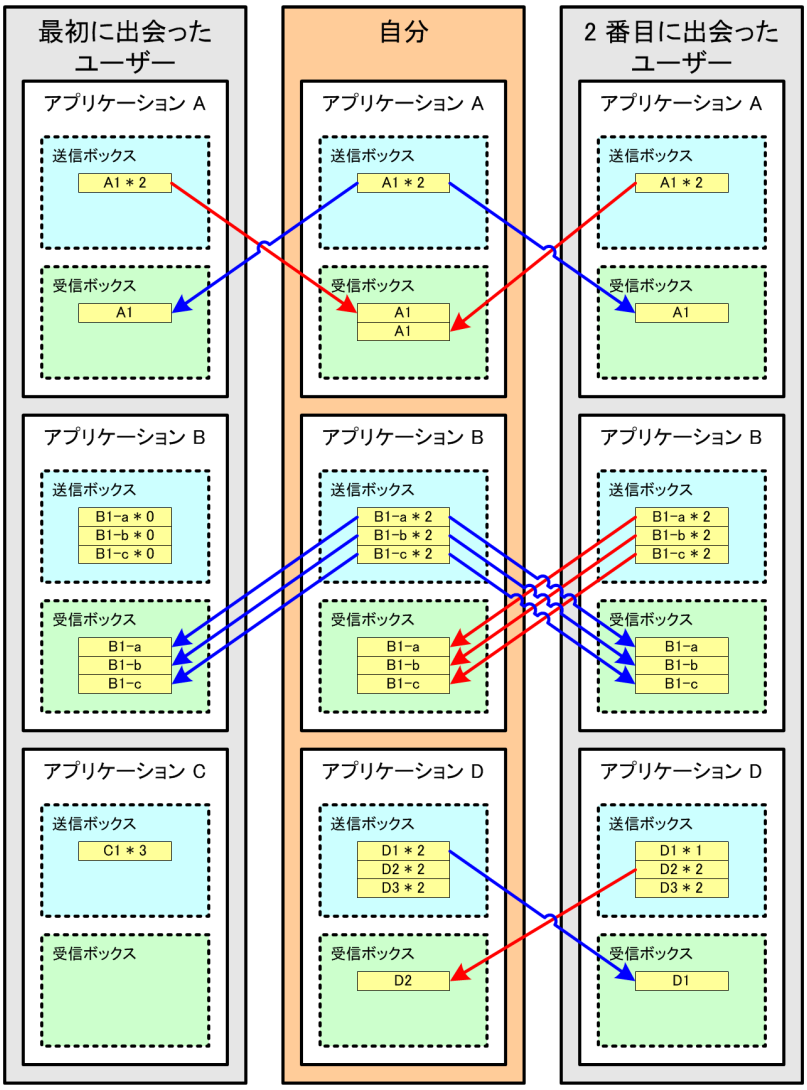
> **補足:** CecNotificationParam 構造体の messageId メンバには、そのすれちがいボックスに受信した最新のすれちがいデータのメッセージ ID が格納されます。ただし、グルーピングされたすれちがいデータを受け取った場合でも、1 件分の情報しか格納されません。

実際に受信したすれちがいデータにアクセスするには、すれちがいボックスのオープンなどの処理を行う必要があります。

4.1.7. すれちがい通信での送受信の例

すれちがいボックスに複数のアプリケーションからすれちがいデータを登録している状態で、ほかのユーザーと 2 回すれちがったときの送受信の例を以下に示します。

図 4-4. すれちがいデータの送受信の例



青い線が送信、赤い線が受信を示し、黄色いタグがすれちがいデータを示します。すれちがいデータの送信モードはすべて送受信に設定されているとします。すれちがいデータの表記では、先頭の英字に続く数字がグループを示し、ハイフンに続く英字が同じグループ内の異なるデータを、アスタリスクに続く数字が送信可能回数を示します。送信可能回数は、自分が登録時の値、出会ったユーザーがすれちがった時点での値です。

アプリケーション A は、すれちがったユーザーの両方が登録していたので、送信と受信が 2 回行われました。

アプリケーション B は、すれちがったユーザーの両方が登録していましたが、最初にすれちがったユーザーは送信可能回数がすでに 0 になっていたため、自分からの送信だけが行われました。2 番目に出会ったユーザーとは、双方の送信可能回数が 0 ではなかったので、送信と受信を行いました。

アプリケーション C は、最初に出会ったユーザーだけが登録していたので、送信も受信も行われませんでした。

アプリケーション D は、2 番目に出会ったユーザーと送信と受信を行いましたが、自分が送信するデータの順番が "D1" で、相手が送信するデータの順番が "D2" だったため、互いに異なるデータを受信しました。

4.1.8. 効果的な設定

3DS のすれちがい通信はバックグラウンド処理で動作するため、動作中のアプリケーション以外にもすれちがいの機会が与えられることになります。しかし、すれちがい通信は複雑な設定項目が多く、効果的にすれちがいデータの送受信が行われ

るようにするには工夫が必要です。ここでは、すれちがい通信の機会が訪れたときにデータの送受信につながるような設定や、設定項目の有効な利用方法について説明します。

また、ライブラリには多様な目的のための機能が用意されています。開発者は、それらの機能を利用すると、データの送受信がどの程度の頻度でどのくらいのユーザー間で発生するかを常に意識する必要があります。

4.1.8.1. 推奨設定

ユーザーがすれちがい通信でやり取りしたいデータを選択し、自ら登録するような一般的なケースを取り上げます。この場合、ユーザーにとってすれちがい通信の結果が分かりやすく、送受信がもっとも活発に行われるような設定として、以下の設定を推奨します。

(1) 登録するデータ数は 1 つだけ。

ユーザーにとっても分かりやすく、すれちがい通信の仕様が持つ、多くの制約を回避することができます。
見かけ上複数のデータを登録させたい場合は、1 つのデータに統合して登録してください。また、グループ化を利用する方法もあります。グループ化せずに複数のデータを登録すると、一度のすれちがい通信で 1 つだけしか送受信されなくなってしまいます。

(2) 送受信モードは「送受信」。

「送受信」は互いの受信ボックスの状態に関わらず送受信が発生するため、機会損失がありません。「交換」は「送受信」とほぼ同じ状況で送受信が発生しますが、片方の受信ボックスに空きがない場合など、両方が通信に成功しないとすれちがい通信が成立しません。つまり、受信ボックスに空きがあったユーザーはデータを受け取る機会を損失してしまいます。
「送信のみ」と「受信のみ」は、同じ送受信モードの相手とすれ違ってもすれちがい通信が発生しません。ユーザーがどちらかを選択できるようにした場合でも、全体的に見たときにどちらかに大きく偏る可能性があり、結果として送受信の機会を損失してしまいます。

(3) 送信可能回数は無制限。

常にすれちがい通信を行おうとするため、機会損失がありません。また、データの登録を解除するまで有効になることは、ユーザーにとっても分かりやすい設定方法です。
無制限にしなかった場合は、いつの間にか送信可能回数が 0 になり、気付かないうちに配布を終了してしまいます。終了したことはユーザーに分かりにくく、送信可能回数が 0 のユーザーとすれ違ったユーザーはデータを受け取る機会を損失してしまいます。

(4) 送信対象はフレンドと非フレンドの両方。

送信対象を限定しないため、機会損失がありません。
非フレンドに限定すると、顔を合わせる機会が多いと思われる、フレンド関係にあるユーザーとの送受信が行われなくなってしまいます。逆にフレンドに限定すると、たくさんの一般のユーザーとのすれちがい通信が行われなくなってしまいます。

4.1.8.2. 複数のデータを登録する

一人の相手に対して複数のデータを一度に送りたい場合は、グループ化よりもデータを 1 つにパックすることをお勧めします。データをまとめることで、グループ化では無駄となるヘッダ情報を削減でき、全体のデータ量を抑えることができます。

複数のデータを登録する際に一番気をつけなければならないことは、3DS のすれちがい通信では一度に 1 つのデータしか送信されず、先頭のデータが送信されるまで後続のデータが送信されることがないということです。つまり、送受信される機会の極端に低いデータが先頭にある場合は、後続のデータに順番が回りにくくなってしまいます。

また、同じ相手とは最大で 8 時間、平均 4 時間経つまで再度すれちがい通信をしないため、同じ相手には先頭から 2 つ目

以降のデータが連続して届きません。ユーザーが複数データを登録した場合、どちらが相手に届くかはどちらがそのとき先頭にあるかで決まるため、偶然の要素が大きくなります。そのため、ユーザーにとっては分かりにくくなる恐れがあります。

4.1.8.3. 送信対象の設定を利用する

すれちがいデータを送信する相手を限定する送信対象の設定ですが、グループ化と組み合わせることで、フレンドとそうでないユーザーとに異なるデータを送信することができます。

具体的な設定例としては、グループの先頭データは送信対象を限定しない共通部分とし、後続のデータにフレンド用と非フレンド用の 2 つのデータを加えます。すれちがい通信は送受信の判断を先頭のデータで行いますので、先頭に来るデータにはなるべく機会損失の少ない設定を行ってください。

4.1.8.4. 伝播可能回数の設定を利用する

伝播可能回数を複数回に設定したすれちがいデータは、遠くのユーザーが登録したものを、複数のユーザーを介して受信する可能性があるという面白みがあります。しかし、伝播の途中でデータが削除されたり、送信ボックスに空きがなかったり、すでに同じデータを受信していたりすると伝播が途絶えてしまいます。また、送信ボックスにほかユーザーのデータが登録されることで、送信の順番によってはユーザーが登録したデータがすぐには送信されない可能性があります。

伝播するデータは、運営者などの限られたユーザーから特別に配信したり、アプリケーションが低い確率で登録したりするスペシャルデータを特定の地域でのみ流通させたい場合などに利用することができます。

ユーザーが登録するすれちがいデータに紛れて配信する場合は特別な設定は必要ありません。グループ化すれば、ユーザーの登録したデータと一緒に送信することもできます。

ユーザーがすれちがいデータを登録せずに、伝播されてくるデータを待ち受ける場合、待ち受ける側はダミーデータを登録しなければならないことに注意してください。たとえば運営者からの特別配信を行う場合、運営者は「送信のみ、伝播可能回数 2 以上、送信可能回数 1」で設定し、待ち受ける側は「受信のみ、伝播可能回数 1、送信可能回数 1」に設定することで、運営者とすれちがったユーザーからほかのユーザーに伝播するデータを登録することができます。

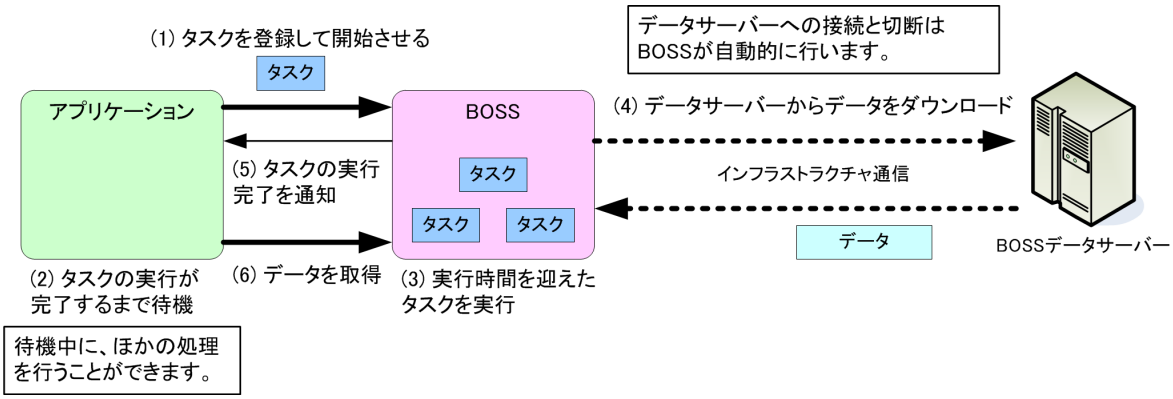
4.2. いつの間に通信

いつの間に通信は BOSS（Background Online Service System の略）と呼ばれる機能を利用して行われる通信の総称です。アプリケーションは BOSS を利用することで、任天堂が提供するデータサーバー（以降、BOSSデータサーバーと呼びます）との通信処理をバックグラウンドで行わせることができます。

アプリケーションから BOSS を利用するには、行わせたい通信処理に関する情報のセットである「タスク」を BOSS に登録し、そのタスクを開始させます。タスクの設定には実行間隔や消尽回数（タスクの残り実行回数）といった情報があり、BOSS はそれらの情報をもとに、タスクを定期的にバックグラウンドで実行します。

> **補足：** ここでいう「バックグラウンド」とは、スリープ中や HOMEメニューの表示中を指します。つまり、アプリケーションが実行中ではない状態です。

図 4-5. BOSS の機能を利用したデータのダウンロード



> **補足:** EULA(End User License Agreement)に同意していない場合、BOSS に登録されたタスクの実行は、EULA への同意によって実行が許可されるまで保留されます。

データをダウンロードするタスクについては、アプリケーションは新規データのダウンロード完了を、タスクの状態の確認やタスクの実行結果の確認、新着データイベントによる通知などで把握することができます。また、そのデータを BOSS 経由で読み込むことができます。
現在、以下のタスクが用意されています。

- Nintendo アーカイブダウンロードタスク(「4.2.2. Nintendo アーカイブダウンロードタスク(NADL タスク)」)
- データアップロードタスク(「4.2.4. データアップロードタスク」)
- DataStore アップロードタスク(「4.2.5. DataStore アップロードタスク」)
- DataStore ダウンロードタスク(「4.2.6. DataStore ダウンロードタスク」)

> **補足:** データアップロードタスク、DataStore アップロードタスク、DataStore ダウンロードタスクにつきましては、サーバー環境のセットアップが必要ですので、製品に利用する場合は弊社窓口までご相談ください。

タスクは基本的にバックグラウンドで実行されますが、登録済みのタスクをアプリケーションから任意のタイミングでただちに実行させることもできます。この場合、タスクによる通信処理はフォアグラウンドで実行されることになります。

また、これらのタスクをフォアグラウンドで実行(アプリケーションの実行中にタスクを登録し、ただちに実行)するために、専用のタスク登録関数(即時実行専用タスクとして登録する関数)も用意されています。

BOSS を利用した基本的なサービス例には、以下のものが考えられます。

- バックグラウンドで、定期的におしらせや追加データを配信する(NADL タスク)
- ニンテンドーゾーンを利用して、事業者ごとに異なるデータを配信する(NADL タスク)
- ユーザー間のデータのやり取りをバックグラウンドで行う(DataStore アップロード/ダウンロードタスク)

4.2.1. すべてのタスクで共通する処理

ここでは、タスクの種類に関わらず共通する処理について説明します。

タスクを利用するにあたって、アプリケーションが行うべき処理の内容を以下に示します。

- BOSS ライブラリの初期化(参照:「4.2.1.1. BOSS ライブラリの初期化」)
- タスクの登録準備(参照:「4.2.1.2. タスクのプロパティ設定」、「4.2.1.3. タスクの動作設定」)

- タスクの登録と実行(参照:「4.2.1.4. タスクの登録と実行」)
- タスクの完了待ち(参照:「4.2.1.5. タスクの情報」、「4.2.1.6. タスク一覧の取得」)
- タスクの実行結果に対応した処理
- アプリケーションによる処理(ダウンロードしたデータの処理など)
- タスクの登録情報の変更(参照:「4.2.1.7. タスクの変更」)
- タスクの登録解除(参照:「4.2.1.8. タスクの登録解除」)
- BOSS ライブラリの終了(参照:「4.2.1.9. BOSS ライブラリの終了」)
- BOSS ライブラリのエラーハンドリング(参照:「4.2.1.10. BOSS ライブラリのエラーハンドリング」)

基本的にアプリケーションでは、上記の内容を上から順に処理することになります。

4.2.1.1. BOSS ライブラリの初期化

BOSS を利用するためには、nn::boss::Initialize() で BOSS ライブラリの初期化を行う必要があります。

コード 4-20. BOSS ライブラリの初期化

```
nn::Result nn::boss::Initialize(void);
```

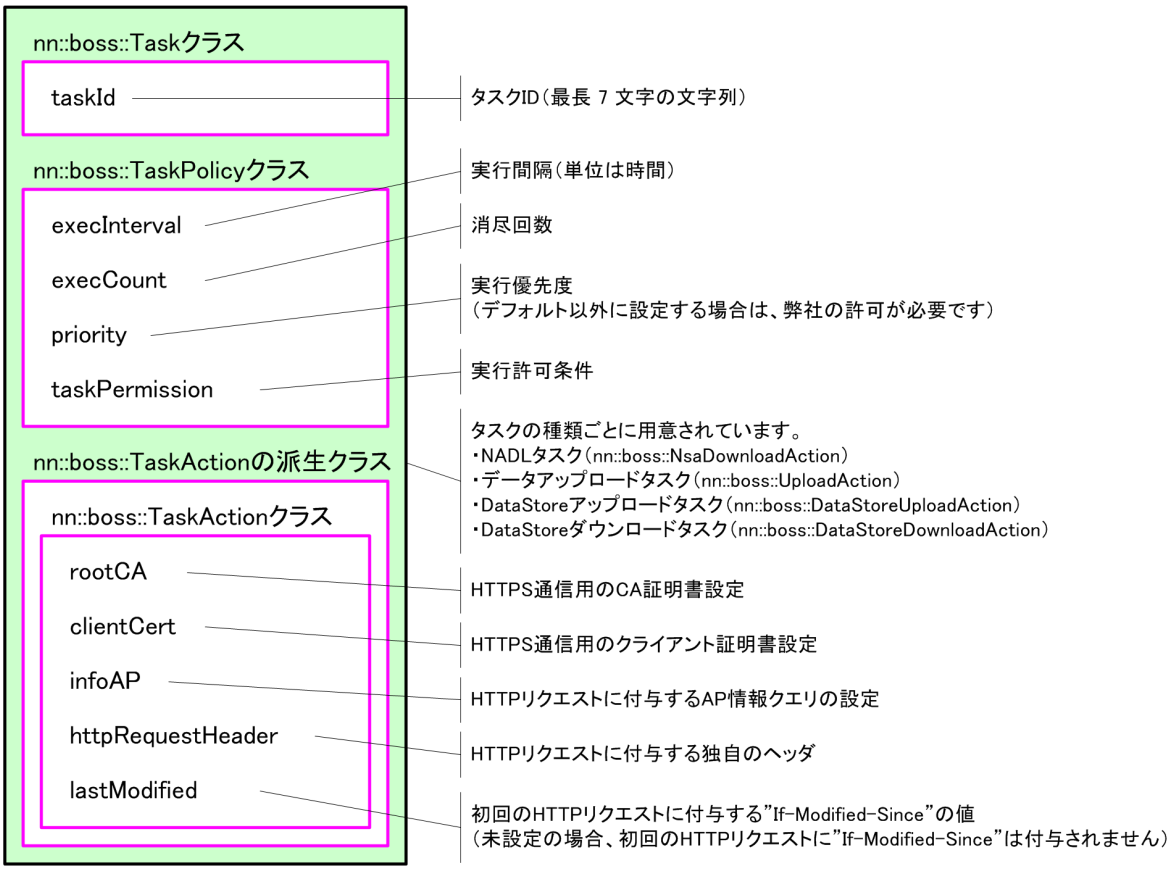
初期化成功前に BOSS ライブラリの関数を呼び出すと、nn::boss::ResultIpcNotSessionInitialized が返されます。

4.2.1.2. タスクのプロパティ設定

タスクのプロパティは 3 つのクラスに分けて設定を行います。設定を行った 3 つのクラスを引数に nn::boss::RegisterTask() を呼び出すことで、1 つのタスクが BOSS に登録されます。

- 基本情報(nn::boss::Task クラス)
- 実行方針(nn::boss::TaskPolicy クラス)
- 動作設定(nn::boss::TaskAction クラス)

図 4-6. タスクのプロパティ構成図



実際にアプリケーションが使用しても構わないプロパティ値には制約があります。

下表は、NADL タスクを登録する際の制約をまとめたものです。

表 4-8. NADL タスクを登録する際のプロパティ値の制約

プロパティ	設定値に対する制約
taskId	弊社に申請したタスク ID を利用してください。
execInterval	1 時間～168 時間（1 週間）の間で設定してください。 デバッグ用途で提供している、秒単位の実行間隔を設定できる関数を使用している場合はロットチェックを通りません。
execCount	1 回～100 回の間で設定してください。
priority	デフォルトの設定（PRIORITY_MEDIUM）のままにしてください。 この設定値以外を設定する場合は、事前に弊社窓口までご相談ください。
taskPermission	デフォルトの設定（TASK_PERMISSION_IN_PARENTAL_CONTROL）のままにしてください。 nn::boss::TaskPolicy クラスの SetProperty 関数で再設定するときには、必ず TASK_PERMISSION_IN_PARENTAL_CONTROL を含めてください。
infoAP	通常のタスクでは、AP 情報を付与する設定にはしないでください。 AP 情報を付与することで、ニンテンドーゾーンの場所や店舗に応じた個別のデータを配信することができます。AP 情報を付与する NADL タスクを利用する場合は、企画検討の早い段階で弊社窓口までご相談ください。

基本情報(nn::boss::Task クラス)

タスクの基本情報を設定する nn::boss::Task クラスは、タスクの識別子(タスク ID)をプロパティに持っています。タスク ID の設定は、nn::boss::Task クラスのインスタンスを生成したあとの、Initialize() による初期化時に行います。

コード 4-21. nn::boss::Task クラスの初期化

```
class nn::boss::Task
{
    nn::Result Initialize(const char* pTaskId);
}
```

タスク ID は半角英数字(0～9、A～Z、a～z)とアンダーバー(_)およびハイフン(-)で構成された最長 7 文字(終端文字を含めると 8 文字)の文字列です。アプリケーションのタスクを一意に表す ID ですので、アプリケーションで登録する各タスクで一意となる値を指定してください。また、タスク ID を途中で変更することはできません。

> **注意:** タスク ID には、必ず弊社に申請したタスク ID を利用してください。申請されていないタスク ID が設定された場合はロットチェックを通りません。

実行方針(nn::boss::TaskPolicy クラス)

タスクの実行方針を設定する nn::boss::TaskPolicy クラスは、タスクの実行間隔や消尽回数をプロパティに持っています。これらのプロパティの設定は、nn::boss::TaskPolicy クラスのインスタンスを生成したあとの、Initialize() による初期化時に行います。初期化後に個別に変更することも可能です。実行間隔と消尽回数の変更については、「4.2.1.7. タスクの変更」を参照してください。

コード 4-22. nn::boss::TaskPolicy クラスの初期化

```
class nn::boss::TaskPolicy
{
    nn::Result Initialize(u32 interval, u32 count);
}
```

interval に実行間隔、*count* に消尽回数を指定します。実行間隔は 1 時間単位で 1～168 時間(7 日)、消尽回数は 1～100 回です。消尽回数に UNLIMITED_COUNT は設定しないでください。消尽回数はタスクが実行されるたびに 1 減少し、0 になるとそのタスクは実行されなくなります。

> **補足:** タスクが実行され、実行終了状態 (nn::boss::Task::GetState() で得られる タスクの状態が nn::boss::TASK_DONE か nn::boss::TASK_ERROR) になると、消尽回数が 1 差し引かれます。タスクの実行中に中断しても減少しません。

> **注意:** デバッグ用途のみですが、InitializeWithSecInterval() で実行間隔を秒単位で設定することができます。ただし、実行間隔を秒単位で登録している場合はロットチェックを通りません。

これまでに紹介したプロパティを含め、すべてのプロパティは SetProperty() と GetProperty() で設定と取得を行うことができます。

コード 4-23. プロパティの設定、取得

```
class nn::boss::TaskPolicy
{
    nn::Result SetProperty(PropertyType type, void* pValue, size_t size);
    nn::Result GetProperty(PropertyType type, void* pValue, size_t size);
}
```

type に指定されたプロパティ識別子でアクセスするプロパティが決定します。*pValue* と *size* には、プロパティの値を格納する変数とそのバイトサイズ(プロパティの型により変化)を指定します。

表 4-9. nn::boss::TaskPolicy クラスで指定可能なプロパティ識別子

プロパティ識別子	型	プロパティ
TASK_EXEC_INTERVAL	u32	実行間隔(秒単位)
TASK_EXEC_COUNT	u32	消尽回数
TASK_PERMISSION	TaskPermission	※ 表外の説明を参照してください

プロパティ識別子の TASK_PERMISSION は、ペアレンタルコントロールで「他ユーザーとのインターネット通信」が制限されている場合の動作を制御するためにアクセスします。何も設定しない場合、ペアレンタルコントロールで「他ユーザーとのインターネット通信」が制限されていると、そのタスクは処理されません。

NADL タスクはユーザー間でデータ交換を行わないタスクですので、デフォルトで TASK_PERMISSION 識別子の値に TASK_PERMISSION_IN_PARENTAL_CONTROL を付加されています。そのためデフォルトのままであれば、ペアレンタルコントロールで「他ユーザーとのインターネット通信」が制限されていても、NADL タスクは処理されます。

> **注意:** NADL タスクでは、プロパティ識別子 TASK_PERMISSION が TASK_PERMISSION_IN_PARENTAL_CONTROL に設定されていない場合はロットチェックを通りません。プロパティ識別子 TASK_PERMISSION の設定を変更するときは、TASK_PERMISSION_IN_PARENTAL_CONTROL を含める必要があります。
>
> プロパティ識別子の TASK_PERMISSION では、EULA 非同意時の動作を設定することができますが、アプリケーションが登録するタスクでは EULA 非同意時の動作設定は行わないでください。動作設定を行わない場合、登録するタスクは EULA 非同意時には処理されないタスクとなります。

動作設定(nn::boss::TaskAction クラス)

タスクの動作に関するプロパティ設定は、タスクの種類に対応した nn::boss::TaskAction の派生クラスを介して行います。現在、以下のタスクが用意されています。

表 4-10. nn::boss::TaskAction の派生クラスと対応するタスクの種類

派生クラス	タスクの種類
nn::boss::NsaDownloadAction	Nintendo アーカイブダウンロードタスク(NADL タスク)
nn::boss::UploadAction	アップロードタスク
nn::boss::DataStoreUploadAction	DataStore アップロードタスク
nn::boss::DataStoreDownloadAction	DataStore ダウンロードタスク

それぞれのタスクの利用方法やプロパティ設定については、後述する各タスクの説明を参照してください。
アップロードタスクは弊社窓口にご相談いただいた際に、個別に提供しています。

4.2.1.3. タスクの動作設定

タスクの動作に関する設定を行う nn::boss::TaskAction クラスでは、サーバーへの接続時に付与する情報の設定や証明書の設定などを行うことができます。

コード 4-24. タスクの動作の設定関数

```
class nn::boss::TaskAction
{
    nn::Result SetApInfo(ApInfoType info);
    nn::Result SetCfgInfo(CfgInfoType info);
    nn::Result AddHeaderField(const char* pLabel, const char* pValue);
    nn::Result SetLastModifiedTime(const char* pLastModifiedTime);
    nn::Result SetRootCa(u32 inCaCertId);
    nn::Result SetClientCert(u32 inClientCertId);
}
```

nn::boss::TaskAction クラスのすべてのプロパティは、SetProperty() と GetProperty() で設定と取得を行うことができます。関数の定義は nn::boss::TaskPolicy クラスでの定義と同じです。

AP 情報の付与

SetApInfo() を呼び出して、「HTTP クエリーへの AP(アクセスポイント)情報付与」を指定することができます。この指定により、ニンテンドーゾーンのアクセスポイントを経由してインフラストラクチャ通信が行われている場合に、HTTP リクエストのクエリとして付与される、ニンテンドーゾーン AP から BOSSデータサーバーに送られる AP 情報を選択することができます。付与されるクエリは、ニンテンドーゾーン経由の接続時にのみタスクの処理に変化を加えたい場合に利用することができます。たとえば、「ニンテンドーゾーン接続時のみ異なるデータをダウンロードする」といった使い方が想定されます。

> **注意:** AP 情報が付与されているとロットチェックを通りません。AP 情報を付与したタスクを利用する場合は、企画検討の早い段階で事前に弊社窓口までご相談ください。

表 4-11. HTTP リクエストに付与される AP 情報

クエリ	付与される情報
ap	AP 単位で識別可能な情報
apgroup	AP のグループ(企業など)単位で識別可能な情報

付与する AP 情報は、以下の定義から 1 つを選択して指定します。

表 4-12. HTTP リクエストに付与する AP 情報の指定

定義	説明
APINFOTYPE_AP	ap を付与する。 ニンテンドーゾーン事業者別配信を申請した場合に指定してください。
APINFOTYPE_APGROUP	通常は指定しないでください。

本体設定情報の付与

SetCfgInfo() を呼び出して、「HTTP クエリーへの本体設定情報付与」を指定することができます。この指定により、HTTP リクエストのクエリに本体情報を付与することができます。たとえば、「独自サーバーに接続された本体の言語設定によって異なるデータをダウンロードする」といった使い方が想定されます。

> **補足:** この機能は独自サーバーを使った BOSS サービス専用の機能です。任天堂サーバーを用いた BOSS サービスでは、本体設定情報を付与しても効果はありません。
>
> 付与された情報を使う独自サーバーを検討されている場合は、弊社窓口までご相談ください。

付与する本体情報の指定には、以下の定義を使用します。なお、複数の値を論理和で設定することで、複数の情報を同時に付与することができます。

表 4-13. HTTP リクエストに付与する本体設定情報の指定

定義	説明
CFGINFOTYPE_NONE	本体設定情報を付与しない。
CFGINFOTYPE_COUNTRY	本体設定情報の国コードを付与する。
CFGINFOTYPE_LANGUAGE	本体設定情報の言語コードを付与する。

HTTP リクエストヘッダの追加

AddHeaderField() を呼び出して、独自の HTTP リクエストヘッダを接続に含めることができます。追加することのできる HTTP リクエストヘッダのラベルは最長 32 文字、値は最長 256 文字、個数は最大 3 つです。

初回実行時の最終更新時刻の指定

SetLastModifiedTime() を呼び出して、タスクの初回実行時の "If-Modified-Since" ヘッダフィールドに入る値を文字列で指定することができます。この指定を行わなかった場合は、初回実行時に "If-Modified-Since" ヘッダフィールドが付与されません。2 回目以降の実行時には、前回実行時に取得したデータの最終更新時刻(HTTP レスポンスの "Last-Modified" フィールドの値)が "If-Modified-Since" ヘッダフィールドとして付与されます。タスクを実行した結果、未更新(サーバーから "304 Not Modified" のステータスが返された場合)のためにデータを取得しなかった場合、タスクの消尽回数は 1 減少します。

そのほかにも、機器内蔵の証明書に関するプロパティの設定関数(nn::boss::TaskAction クラスから継承)が利用可能です。

コード 4-25. 機器内蔵の証明書に関するプロパティの設定関数

```
class nn::boss::TaskAction
{
    nn::Result SetRootCa(u32 inCaCertId);
    nn::Result SetClientCert(u32 inClientCertId);
}
```

機器に内蔵されている証明書のうちのどれを使用するのかは、SetRootCa() と SetClientCert() で設定します。ルート CA 証明書は 3 つまで、クライアント証明書は 1 つだけ設定することができます。

独自証明書の設定

タスク実行時の接続で独自のルート CA 証明書またはクライアント証明書を使用する場合、以下の関数で独自の証明書を設定することができます。独自の証明書を使用しない場合は呼び出す必要はありません。

コード 4-26. 独自証明書の設定

```
nn::Result nn::boss::RegisterPrivateRootCa(
                        const u8* pCertData, size_t certDataSize);
nn::Result nn::boss::RegisterPrivateClientCert(
                        const u8* pCertData, size_t certDataSize,
                        const u8* pPrivateKeyData, size_t privateKeyDataSize);
```

独自の証明書と機器に内蔵されている証明書は併用することができます。ルート CA 証明書、クライアント証明書ともに独自の証明書として設定できるのは 1 つだけです。重複して設定した場合は、最後に設定した証明書が有効になります。

TaskActionBase クラスの SetPrivateRootCa() または SetPrivateClientCert() を呼び出すことで、独自証明書を使用することができます。なお、ルート CA 証明書のみ内蔵の証明書と併用することができます。

> **補足:** 現在、任天堂が提供する BOSSデータサーバー以外に接続するタスクの登録は許可されていません。

4.2.1.4. タスクの登録と実行

4 つのクラスに設定されたタスクのプロパティをもとに、nn::boss::RegisterTask() でタスクを登録します。

コード 4-27. タスクの登録

```
nn::Result nn::boss::RegisterTask(nn::boss::Task* pTask,
                                  nn::boss::TaskPolicy* pPolicy,
                                  nn::boss::TaskAction* pAction,
                                  nn::boss::TaskOption* pOption=NULL,
                                  u8 taskStep=DEFAULT_STEP_ID);
```

> **補足:** *pOption*、*taskStep* は機能拡張用の引数のため、現状未対応です。指定しないでください。

登録されただけではタスクは実行されません。nn::boss::Task クラスの Start() または StartImmediate() で、BOSS に実行を指示する必要があります。

コード 4-28. タスクの実行、中止、完了待ち

```
class nn::boss::Task
{
    nn::Result Start(void);
    nn::Result StartBgImmediate(void);
    nn::Result StartImmediate(void);
    nn::Result Cancel(void);
    nn::Result WaitFinish(void);
    nn::Result WaitFinish(const nn::fnd::TimeSpan& timeout);
}
```

Start() で開始を指示されたタスクは、BOSS によるバックグラウンドでのタスク制御（スケジューリングと実行）の対象となります。バックグラウンドでタスクを実行する場合は、BOSS がインフラストラクチャ通信の接続と切断を自動的に行います。タス

クの実行中に通信が切断された場合、そのタスクはレジューム状態となります。レジューム状態のタスクは、次回のインフラストラクチャ通信の接続時に、切断された時点の状態から実行を再開します。

Start() で開始が指示されているタスクに対しては、スケジューリングや実行の中止を Cancel() で指示することができます。中止後に Start() を呼び出した場合は、再度スケジューリングや実行の対象となります。

StartImmediate() はフォアグラウンドでタスクを実行するために用意されています。StartImmediate() で開始を指示されたタスクはすぐ実行に移ります。フォアグラウンド通信によるタスク実行となりますので、インフラストラクチャ通信の接続と切断はアプリケーションで行わなければなりません(BOSS は実施しません)。StartImmediate() でタスクを実行した場合は Start() によるタスクのバックグラウンド実行とは異なり、タスクの実行中に通信が切断された場合はエラーが返され、レジューム状態にはなりません。

Start() で開始を指示していたタスクを、次回の実行時刻を待たずにすぐに実行したい場合にも、StartImmediate() を使用することができます。ただし、この場合でもフォアグラウンドでタスクを実行することになりますので、インフラストラクチャ通信の接続と切断はアプリケーションで行わなければなりません。

補足: タスクの実行中にほかのタスクが実行開始日時を迎えても、実行中のタスクが完了するまで実行されません。ただし、StartImmediate() でタスクを即時実行した場合、バックグラウンドで実行されていたタスクの実行は中止されますが、即時実行したタスクの完了後に再度実行されます。

注意: Start() で開始を指示していたタスクを StartImmediate() で実行する際、そのタスクがちょうど実行中であった場合、タスクはリトライ状態に遷移してしまいます。通常、アプリケーションの動作中に BOSS はタスクを実行することがないため問題になりませんが、アプリケーションの動作中でも BOSS が動作できるように許可(NDM ライブラリで設定可能)している場合は、この点への配慮が必要になります。そのようなアプリケーションは、以下のような対応を行ってください。

- そのタスクを Cancel() で中止してから、StartImmediate() で実行する。

バックグラウンドでのタスクの即時実行をデーモンに指示するには StartBgImmediate() を呼び出してください。この関数は、**タスクの初回スケジュール実行を即時に行う**ための関数です。そのため、すでに Start() や StartImmediate() によってスケジューリングの対象となっているタスクに対して呼び出しても、再び即時に実行されるようなことはありません。呼び出しのたびにタスクを即時に実行する場合は、StartImmediate() を使用してください。また、すでにスケジューリング対象となっているタスクに対し、再びバックグラウンドでのタスク即時実行を指示する場合は、そのタスクの実行を Cancel() で中止してから、StartBgImmediate() を呼び出してください。なお、StartImmediate() はほかのタスクの実行をキャンセルしますが、StartBgImmediate() ではほかのタスクが実行中であれば、そのタスクの実行が完了したあとにタスクが実行されます。

WaitFinish() はタスクの 1 回の実行が完了するのを待ちます。消尽回数が 1 回のタスクや、StartImmediate() で即時実行を指示したときなど、アプリケーションでタスクの完了を待つ場合に使用することができます。この関数を呼び出すと、タスクの実行が完了するタイミングまで制御が戻らないことに注意してください。WaitFinish() の実行には、専用のスレッドを用意するか、引数 *timeout* を持つオーバーロードを使用することを推奨します。オーバーロードでは、指定された時間内にタスクの 1 回の実行が完了しなかった場合に ResultWaitFinishTimeout を返し、アプリケーション側に制御が戻ります。

タスクの実行完了を待つ方法として、新しいデータの到着を利用することができます。新着データの確認には nn::boss::GetNewArrivalFlag()、nn::boss::RegisterNewArrivalEvent() を利用してください。これらの関数の詳細については「4.2.2.6. 新着フラグのチェック、データ新着イベントによる待ち受け 」を参照してください。

ほかにも、nn::boss::GetState() や nn::boss::GetStateDetail() を利用して、タスクの状態をポーリングする方法があります。

即時実行専用タスクの登録

即時実行のみ可能なタスクを登録する nn::boss::RegisterImmediateTask() が用意されています。**即時実行専用タスクは必ずこの関数で登録し、フォアグラウンドで実行してください。1 つのアプリケーションが登録できるタスク数はガイドラインで制限されていますが、この関数で登録する即時実行専用タスクはそのタスク数にはカウントされません。**

コード 4-29. 即時実行専用タスクの登録

```
nn::Result nn::boss::RegisterImmediateTask(
        nn::boss::Task* pTask, nn::boss::TaskAction* pAction,
        nn::boss::TaskPolicy* pPolicy=NULL, nn::boss::TaskOption* pOption=NULL,
        u8 taskStep=DEFAULT_STEP_ID);
```

pTask に指定するのは nn::boss:Task クラスではなく、そのサブクラスの nn::boss::FgOnlyTask クラスです。このクラスを使った場合、nn::boss::FG_ONLY_TASK_ID で定義された即時実行専用のタスク ID が自動的に指定されますので、タスク ID を指定する必要はありません。また、*pPolicy* で消尽回数や実行間隔を設定しても無効となります。

タスクの実行は nn::boss::FgOnlyTask::StartImmediate() で行わなければなりません。それ以外の関数でタスクを実行するとエラーが返されます。

登録できるタスクの最大数と自動登録解除

BOSS に登録することのできるタスクの数は最大で 127 個(即時実行専用タスクは含みません)です。この数はすべてのアプリケーション(内蔵アプリケーション含む)で登録されたタスクの総計の最大値です。なお、1 つのアプリケーションが登録できるタスクの最大数はガイドラインによって制限されています。登録されているタスクが最大数に達している状態で、アプリケーションが新たなタスクを登録しようとした場合、BOSS は「消尽回数が 0 のタスク」を自動的に登録解除して、新たなタスクの登録を受け容れます。登録しようとしているアプリケーションのタスクに限らず、すべてのタスクの中から該当するタスクを探して登録解除します。消尽回数が 0 のタスクが存在しない場合は、消尽回数と実行間隔の乗算結果がもっとも小さなタスクを登録解除し、新たなタスクの登録を受け付けます。

上記の自動登録解除により、アプリケーションを長期間起動しなかった場合は、以前登録したタスクが BOSS によって登録解除されている可能性があります。そのため、BOSS を利用するアプリケーションは、タスクが登録解除されているケースに対応する必要があります。たとえば、アプリケーション起動時にタスクが登録されているかを確認し、未登録の場合はユーザーにタスクを登録するかどうかを確認するような処理が必要となります。

4.2.1.5. タスクの情報

状態や実行結果などのタスクの情報は nn::boss::Task クラスのメンバ関数で取得することができます。

コード 4-30. タスクの情報の取得

```
class nn::boss::Task
{
    TaskServiceStatus GetServiceStatus(void);
    TaskStateCode     GetState(bool acknowledge=false,
                               u32* pCount=NULL, u8* pStepID=NULL);
    u32               GetHttpStatusCode(u32* pCount=NULL, u8* pStepID=NULL);
    TaskResultCode    GetResult(u32* pCount=NULL, u8* pStepID=NULL);
    nn::Result        GetStateDetail(TaskStatus* pStatus,
                                     bool acknowledge=false, u8* pStepID=NULL,
                                     u8 taskStep=CURRENT_STEP_ID);
    nn::Result        GetError(TaskError* pTaskError, u8* pStepID=NULL,
                               u8 taskStep=CURRENT_STEP_ID);
    nn::Result        GetInfo(TaskPolicy* pPolicy, TaskAction* pAction,
```

```
                                TaskOption* pOption, u8 taskStep=CURRENT_STEP_ID);
    nn::Result        GetActivePriority(TaskPriority* pPriority);
}
```

上記の関数のうち、引数に *pCount* がある関数は *pCount* に NULL ではなく u32 型変数へのポインタを渡すとタスクの消尽回数を返します。*pStepID* を現在利用することはできません。

> **注意:** *pCount* に返されるタスクの消尽回数と取得したタスクの情報は、ライブラリ内でそれぞれ個別に取得されています。そのため、わずかにタイムラグが存在し、まったく同一の情報であるとは限りません。

GetServiceStatus() を呼び出すことで、タスクの接続先サーバーのサービスの状態(「サービスが利用可能」、「サービスが終了している」など)を取得することができます。サービスの状態はタスクが実行されたときにサーバー側から送信されます。

SERVICE_AVAILABLE が返されたときは、サービスは稼働中です。SERVICE_TERMINATED が返されたときは、サービスは終了しています。タスクがまだ実行されていないときや、ネットワークエラーで接続に失敗したときは、SERVICE_UNKNOWN が返されます。関数の呼び出し自体に失敗したときは GET_SERVICE_STATUS_ERROR が返されます。

> **注意:** サービス終了の判定は、タスク実行時のサーバーからのレスポンスにサービス終了を示すフラグ(サービス終了フラグ)が含まれているかどうかで行われます。サービス終了フラグはアプリケーションの開発者が BOSSデータサーバーを使って付与設定を行うことになります。

GetState() を呼び出すことで、タスクの状態を取得することができます。関数の処理自体に失敗したときは GET_TASK_STATE_ERROR が返されます。タスクの実行結果が確実に取得できるように、タスクの実行が終わってから次のタスク実行時間になるまでの間(つまり状態が TASK_WAITING_TIMER の間)、タスクの前回実行時の結果を保持しています。そのため、実際の状態が TASK_WAITING_TIMER であっても、GetState() は前回実行時の結果(TASK_DONE/TASK_ERROR/TASK_RETRY)を返します。

保持された結果ではなく、実際の状態を取得できるようにするためには、引数の *acknowledge* に true を指定して GetState() を呼び出します。ただし、保持が解除されて実際の状態(TASK_WAITING_TIMER)が取得できるようになるのは、そのあとに GetState() でタスクの状態を取得したときです。

表 4-14. タスクの状態

定義	説明
TASK_STOPPED	停止中のため、タスクはスケジューリングされていません。
TASK_WAITING_TIMER	タスクの実行時刻を待っています。
TASK_WAITING	すでに実行時刻になっていますが、インフラストラクチャ通信の未接続や高い優先度を持った別のタスクの実行中などの理由で、実行を待っています。
TASK_RUNNING	タスクは実行中です。
TASK_PAUSED	タスクの実行を一時停止しています。
TASK_REGISTERED	タスクを登録した直後の状態(開始前)です。
TASK_DONE	タスクの実行が完了しました。
TASK_ERROR	タスクの実行でリトライ不可能なエラー(異常終了)が発生しました。

TASK_RETRY	前回の実行で通信が中断されたため、リトライ(レジューム)を待っています。

GetHttpStatusCode() を呼び出すことで、タスク実行時の HTTP ステータスコードを取得することができます。関数の処理自体に失敗したときは U32_CANNOT_GET_DATA が返されます。

GetResult() を呼び出すことで、タスクの実行結果を取得することができます。関数の処理自体に失敗したときは GET_TASK_RESULT_ERROR が返されます。返り値の定義は nn/boss/boss_Const.h の TaskResultCode 列挙子を参照してください。BOSS ライブラリの中で呼び出されている、ほかのライブラリの関数がエラーを返した場合の返り値は、TaskResultCode 列挙子に存在する値よりも大きい値(UNKNOWN_ERROR +「エラーが発生したモジュールのモジュール ID」。モジュール ID の定義については nn/Result.h を参照)になります。また、以下のような原因によって、タスク実行時のファイル操作でエラーが発生した場合は FS_UNKNOWN_ERROR が返されます。

- SD カードの空き容量が無い
- SD カードが Write プロテクトされている
- SD カードが入っていない。
- タスク登録時の SD カードとは別の SD カードが入っているなどで、対象タスクのデータを書き込んだり、読み込んだりするための BOSS ストレージ(後述)がない。
- SD カードが壊れている。

GetStateDetail() を呼び出すことで、タスクの状態を取得することができます。GetState() との違いは *pStatus* に返される nn::boss::TaskStatus クラスのインスタンスの GetProperty() を介して詳細な情報を取得できることです。

表 4-15. nn::boss::TaskStatus クラスで指定可能なプロパティ識別子

プロパティ識別子	型	プロパティ
TASK_STATE_CODE	TaskStateCode	タスクの状態(GetState() と同じ)
TASK_RESULT_CODE	TaskResultCode	実行結果(GetResult() と同じ)
TASK_COMM_ERROR_CODE	u32	通信エラーコード (GetHttpStatusCode() と同じ)
TASK_CURRENT_PRIORITY	u32	現在の実行優先度
TASK_EXECUTE_COUNT	u32	現在の消尽回数 (最新の値が反映されるまでに時間がかかりますので、nn::boss::TaskPolicy クラスで同名のプロパティを取得することを推奨します)
TASK_PENDING_TIME	u32	タスク実行の保留時間(秒)
TASK_START_TIME	s64	タスク開始日時(2000/1/1 基点。秒単位)
TASK_PROGRESS	u32	ダウンロード済みデータのサイズ
TASK_DATA_SIZE	u32	ダウンロードデータのサイズ(Content-Length)
TASK_SERVICE_STATUS	TaskServiceStatus	サービス利用可能ステータス
TASK_SERVICE_TERMINATED	bool	サービス終了フラグ
TASK_LAST_MODIFIED_TIME	char[MAX_LASTMODIFIED_LENGTH]	ダウンロードデータの最終更新日時 (HTTP レスポンスの "Last-Modified" フィールドの値)

GetError() を呼び出すことで、タスク実行時のエラーを取得することができます。GetCommErrorCode() や

GetResult() との違いは *pTaskError* に返される nn::boss::TaskError クラスの GetProperty() を介して詳細な情報を取得できることです。

表 4-16. nn::boss::TaskError クラスで指定可能なプロパティ識別子

プロパティ識別子	型	プロパティ
TASK_ERROR_RESULT_CODE	TaskResultCode	実行結果(エラーのみ。GetResult() と同じ)
TASK_ERROR_CODE	bit32	タスクの実行でエラーとなった原因
TASK_ERROR_MESSAGE	char[MAX_ERROR_MESSAGE]	タスク実行時の通信エラーメッセージ

GetActivePriority() を呼び出すことで、タスクの現在の実行優先度を取得することができます。

4.2.1.6. タスク一覧の取得

アプリケーションで登録しているタスクの一覧を取得するには nn::boss::GetTaskIdList() を呼び出してください。

コード 4-31. タスク一覧の取得

```
nn::Result nn::boss::GetTaskIdList(nn::boss::TaskIdList* pTaskIdList);
```

nn::boss::GetTaskIdList() の引数 *pTaskIdList* には、nn::boss::TaskIdList クラスのインスタンス(アプリケーションで生成)へのポインタを渡します。インスタンスのサイズは 1 KByte 程度ありますので、インスタンスの生成時にはスタックやヒープのサイズに配慮が必要です。nn::boss::TaskIdList クラスは以下のように定義されています。

コード 4-32. nn::boss::TaskIdList クラス

```
class nn::boss::TaskIdList
{
    explicit TaskIdList(void);
    virtual ~TaskIdList(void);
    u16 GetSize(void);
    char* GetTaskId(u16 index);
}
```

一覧に格納されたタスクの数は GetSize() で取得することができます。GetTaskId() では、一覧に格納されたタスクのタスク ID をインデックス指定で取得することができます。範囲外のインデックスを指定した場合は NULL が返されます。

> **補足:** 現在、nn::boss::GetStepIdList() を使用することはできません。

4.2.1.7. タスクの変更

nn::boss::ReconfigureTask() を呼び出してタスクのプロパティ設定を変更することができます。変更することができるのは nn::boss::TaskPolicy で設定可能なプロパティのみです。ステップ ID は指定しないでください。

コード 4-33. タスクの変更

```
nn::Result nn::boss::ReconfigureTask(nn::boss::Task* pTask,
                                     nn::boss::TaskPolicy* pPolicy,
                                     u8 taskStep=DEFAULT_STEP_ID);
```

消尽回数と実行間隔は nn::boss::Task クラスのメンバ関数でも変更することができます。

コード 4-34. nn::boss::Task クラスでのプロパティ設定

```
class nn::boss::Task
{
    nn::Result UpdateInterval(u32 interval);
    nn::Result UpdateIntervalWithSec(u32 intervalSec);
    nn::Result UpdateCount(u32 count);
    u32 GetInterval(void);
    u32 GetIntervalSec(void);
    u32 GetCount(void);
}
```

UpdateIntervalWithSec() では秒単位で実行間隔を指定することができますが、この関数はデバッグ用途でのみ利用可能です。

4.2.1.8. タスクの登録解除

nn::boss::UnregisterTask() を呼び出してタスクの登録を解除することができます。

コード 4-35. タスクの登録解除

```
nn::Result nn::boss::UnregisterTask(nn::boss::Task* pTask,
                                    u8 taskStep=DEFAULT_STEP_ID);
```

タスクの登録が解除されるときに、BOSS ストレージ内に作成された、そのタスク用の作業用ファイルが削除されます。BOSS ストレージが存在する SD カードが抜かれているなど、作業用ファイルの削除に失敗した場合はエラー(nn::boss::ResultFileAccess)が返されますが、タスクの登録解除には成功しています。作業用ファイルは拡張セーブデータが削除されるタイミングでも削除されますので、このエラーのハンドリングは必須ではありません。作業用ファイルは直接削除することはできませんが、再度登録した同じタスク ID のタスクの登録解除で作業用ファイルを削除することはできます。

> **補足:** 現在、引数 *taskStep* は使用されません。この関数を呼び出すときは、何も指定しないでください。

4.2.1.9. BOSS ライブラリの終了

BOSS の利用が終了した際には、必ず nn::boss::Finalize() を呼び出して BOSS ライブラリの終了処理を行ってください。

コード 4-36. BOSS ライブラリの終了

```
nn::Result nn::boss::Finalize(void);
```

4.2.1.10. BOSS ライブラリのエラーハンドリング

BOSS ライブラリのエラーハンドリング方法は大きく分けて4種類あります。

- プログラミングが原因のエラー
  BOSS ライブラリの使い方が原因となるアサート級のエラーです。アプリケーションで必ず修正してください。
  nn::boss::ResultInvalidTaskId 、nn::boss::ResultIpcNotSessionInitialized 、
  nn::boss::ResultStorageNotFound などがこれにあたります。
  上記は一例です。他の該当リザルトは API リファレンスで確認してください。
- API ごとに独自にハンドリングすべきエラー
  nn::boss::ResultNsDataListSizeShortage 、nn::boss::ResultNsDataListUpdated 、
  nn::boss::ResultTaskIdAlreadyExist などがこれにあたります。
  上記は一例です。他の該当リザルトや対応についての詳細は API リファレンスで確認してください。

- ファイルアクセスエラー
  ファイルシステムのエラーにより BOSS ライブラリの API が返す可能性があります。
  nn::boss::ResultFileAccess のみが該当します。
  nn::boss::TaskResultCode に nn::boss::FS_UNKNOWN_ERROR を指定して
  nn::boss::GetErrorCode() でエラーコードを取得し、エラー・EULA アプレットでエラーを表示してください。
- 上記以外のエラー
  予期せぬエラーです。
  nn::boss::TaskResultCode に nn::boss::SEVERE_ERROR を指定して
  nn::boss::GetErrorCode() でエラーコードを取得し、エラー・EULA アプレットでエラーを表示してください。

nn::boss::GetNsDataIdList() のエラーハンドリング例を次に示します。
プログラミングが原因のエラー、API ごとに独自にハンドリングすべきエラーについては、API ごとに異なるので必ず API リファレンスで確認してください。

コード 4-37. nn::boss::GetNsDataIdList() のエラーハンドリング例

```
static const u32 MAX_DATA_ID = 32;
u32 idBuf[MAX_DATA_ID];// シリアル ID 格納用バッファ。アプリケーションで用意します。大きなバッファを用意すれば、一括で多数のシリアル ID を取得できます。
nn::boss::NsDataIdList serialIdList(idBuf, MAX_DATA_ID);

do
{
    nn::Result result = nn::boss::GetNsDataIdList(nn::boss::DATA_TYPE_ALL,
&serialIdList);
    if(result.IsFailure())
    {
        if(result == nn::boss::ResultStorageNotFound() ||
                result == nn::boss::ResultInvalidNsDataIdList() ||
                result == nn::boss::ResultIpcNotSessionInitialized())
        {
            // プログラミングが原因のエラー( API ごとに異なるので詳細は API リファレンスを参照してください)

            // BOSS ストレージが登録されていない、NS データリスト情報のポインタが NULL、BOSS ライブラリが初期化されていないなど、アプリケーション側の実装ミスによるエラーです。
            // アサート級のエラーなので開発段階で必ず修正してください。
        }
        else if(result == nn::boss::ResultNsDataListSizeShortage())
        {
            // API ごとに独自にハンドリングすべきエラーのハンドリング( API ごとに異なるので詳細は API リファレンスを参照してください)

            // ID 一覧取得に成功しているが、NsDataIdList にすべての ID を格納できなかったケース。
            // 必要なら nn::boss::GetNsDataIdList をもう一度実行してください。
            // この例では do-while を抜けずに再度 GetNsDataIdList を実行するようにしています。
        }
        else if(result == nn::boss::ResultNsDataListUpdated())
        {
            // API ごとに独自にハンドリングすべきエラーのハンドリング( API ごとに異なるので詳細は API リファレンスを参照してください)

            // ID 一覧取得中に BOSS ストレージ内の NS データに追加や削除があったケース。
```

```
            // リストを初期化し、シリアル ID 一覧を初めから取り直してください。
            serialIdList.Initialize();
            result = nn::boss::ResultNsDataListSizeShortage();// do-while ループを抜けないように、result の値を更新。
            continue;
        }
        else if(result == nn::boss::ResultFileAccess())
        {
            // ファイルアクセスのエラーのハンドリング

            // ファイルアクセスエラーを示すエラーコード( nn::boss::FS_UNKNOWN_ERROR のエラーコード)を エラー・EULA アプレットでエラーを表示してください。
            u32 errCode = 0;
            nn::Result getErrCodeResult = nn::boss::GetErrorCode( &errCode, nn::boss::FS_UNKNOWN_ERROR);
            NN_UTIL_PANIC_IF_FAILED(getErrCodeResult);// boss::GetErrorCode() は使い方が間違っている場合にしか失敗しません。詳細は API リファレンスを参照してください。

            // エラー・EULA アプレットのコードは省略します

            break;
        }
        else
        {
            // 予期せぬエラーのハンドリング

            // 上記以外のエラーのエラーはそのことを示すエラーコード( nn::boss::SEVERE_ERROR のエラーコード)を エラー・EULA アプレットでエラーを表示してください。
            u32 errCode = 0;
            nn::Result getErrCodeResult = nn::boss::GetErrorCode( &errCode, nn::boss::SEVERE_ERROR);
            NN_UTIL_PANIC_IF_FAILED(getErrCodeResult);// boss::GetErrorCode() は使い方が間違っている場合にしか失敗しません。詳細はAPIリファレンスを参照してください。

            // エラー・EULA アプレットのコードは省略します

            break;
        }
    }

    {
        // 取得した serialIdList に対する処理
    }
}while(result == nn::boss::ResultNsDataListSizeShortage());// この例では ResultNsDataListSizeShortage の場合は GetNsDataIdList() を再実行します。
```

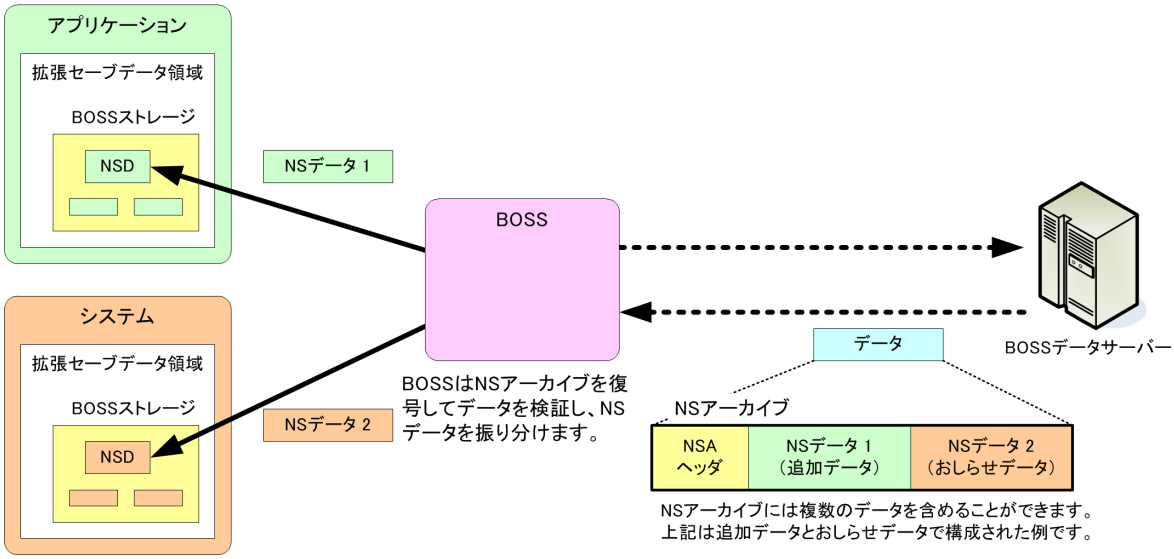
4.2.2. Nintendo アーカイブダウンロードタスク(NADL タスク)

NADL タスクは BOSSデータサーバーから HTTP/HTTPS 接続でデータをダウンロードするタスクです。

アプリケーションにダウンロードさせたいデータは BOSSデータサーバーに登録されていなければなりません。BOSSデータサーバーに登録されたデータは、Nintendo Serendipitous アーカイブ(略称 NS アーカイブ)という任天堂が独自に定義したフォーマットに変換されます。NS アーカイブには複数のデータを含ませることができますので、一度のタスク実行で複数のデータ(たとえば、アプリケーション用のデータと、そのおしらせデータなど)をダウンロードさせたい場合には、複数のデータ

をまとめて BOSSデータサーバーに登録します。

図 4-7. NADL タスクでダウンロードされたデータの流れ



BOSSデータサーバーの使い方については、BOSSデータサーバーにログインした後に閲覧できる「ヘルプ」を参照してください。なお、「ヘルプ」には配信データや運用に関する説明も書かれていますので、実際に運用を開始する前に必ず読むようにしてください。

管理ツールでは、主に以下の操作を行うことができます。

- おしらせ、配信データの作成
- おしらせ、配信データの配信スケジュールの管理
- アプリケーション開発中のテスト配信
- 実運用開始後のユーザーへの配信ログの取得

BOSSデータサーバーを利用するには、BOSS タスクを利用する旨を OMAS(Online title MAnagement System)で申請した際に、「BOSSデータサーバーにアクセスできるユーザー」の項目を申請してください。申請が許可され次第、BOSSデータサーバーのアクセス情報がメールで送信されます。

4.2.2.1. NS アーカイブ、NS データ

NS アーカイブに含まれるデータ(NS データ)はすべて暗号化され、署名とハッシュ値が付与されます。よって、BOSS は NS データの復号化だけでなく、不正なデータがダウンロードされないように改竄検知やなりすまし検知を行います。

NS アーカイブには新着フラグを設定することができます。新着フラグが有効に設定されていると、そのデータがダウンロードされたときに、アプリケーションのアイコン上に新着を示すマークが表示されます。新着フラグは BOSS が管理しているアプリケーションごとに設定され、システムはその設定を確認して新着マークを表示します。これらの処理はすべてシステム側で行われますので、新着マークの表示に関してはアプリケーション側で処理を行う必要はありません。

なお、おしらせデータが含まれている NS データをダウンロードすると、新着フラグが OFF(無効に設定)であっても、おしらせリストには常にマークが表示されます。ただし、これはおしらせリスト上に表示されるマークについてのみであり、アプリケーションのアイコン上に表示されるマークは新着フラグの設定で制御されます。

NS データには以下のものがあります。

表 4-17. NS データの種類

種類	説明
追加データ	追加アイテムや追加コースなど、アプリケーションで使用する様々なデータ。
おしらせデータ	BOSS 経由でおしらせを通知するためのデータ。
DL 拡張バナーデータ	CTR タイトルバナーで表示されるテキストやテクスチャを入れ替えるためのデータ。
DataStore データ	DataStore ダウンロードタスクでダウンロードされたデータ。

NS データはデータタイプという 32 bit の情報を持っています。データタイプは NS データの種類を示すグローバルデータタイプ(上位 16 bit)と、プライベートデータタイプ(下位 16 bit)の 2 部構成となっています。グローバルデータタイプはすべてのアプリケーションで共通の値で、上表のいずれかから指定します。プライベートデータタイプに指定する値はアプリケーションで自由に決めることができます。データタイプは「4.2.2.6. 新着フラグのチェック、データ新着イベントによる待ち受け」で説明する NS データ検索の検索条件として利用します。プライベートデータタイプの割り当てを工夫することで、NS データ検索を効率よく行うことができるようになります。

アプリケーションが一度も起動されたことがない、アプリケーションは起動させたが BOSS ストレージを登録していない、アプリケーションの拡張セーブデータをユーザーが削除したなど、BOSS が NS データを振り分ける際にデータを受け取るアプリケーションの BOSS ストレージが存在していない場合、その NS データは破棄されてしまうことに注意してください。また、NS データにはそのデータの宛先となるアプリケーションのユニーク ID が埋め込まれています。NS データのユニーク ID には、サーバーによって BOSS を使用するアプリケーションのユニーク ID が自動的に埋め込まれます。そのため、アプリケーション開発者がデータに任意のユニーク ID(宛先)を指定することはできません。ダウンロードした NS データは、どのアプリケーションが登録したタスクでダウンロードしたかに関わらず、埋め込まれたユニーク ID のアプリケーションの BOSS ストレージに保存されます。

> **注意:** タスク登録時に誤った URL を指定し、ほかのアプリケーション宛の NS データをダウンロードしてくることがないように注意してください。

NS データにはアプリケーションごとに一意なシリアル ID とバージョン(共に 32 bit の情報)が付与されます。シリアル ID はアプリケーションが NS データを読み込んだり、削除したりする際のデータの指定に利用します。バージョンは BOSS が NS データの更新を判断する際に利用します。ダウンロードしたデータと同じシリアル ID の NS データがすでに存在していた場合、BOSS はバージョンを比較し、ダウンロードしたデータが新しいバージョンであった場合にのみ、NS データを更新(上書き)します。

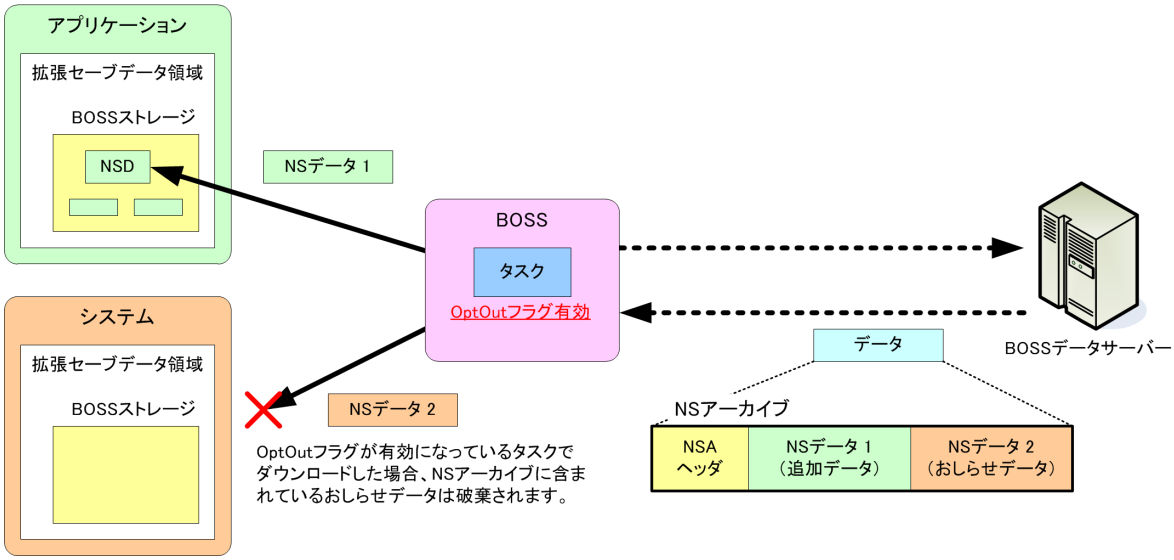
> **補足:** NS データのシリアル ID とバージョンは BOSS 管理サイト上で設定できます。ただし、おしらせデータおよび DataStore データのシリアル ID とバージョンは自動的に設定されるため、開発者の手で設定することができません。詳細は BOSS 管理サイトから取得できるマニュアルを参照してください。

> **注意:** 4294901760 ～ 4294967295(0xFFFF0000 ～ 0xFFFFFFFF)の範囲のシリアル ID はシステムリザーブ領域として一般アプリケーションでの使用が禁止されています。BOSS データサーバーの BOSS 管理サイト上でシリアル ID を設定する際には、この領域以外の値(1 ～ 4294901759(0xFFFEFFFF))を使用してください。

NS データからはデータの内容以外にも、シリアル ID やデータの種類、データサイズなどのプロパティも取得することができます。ただし、おしらせデータはシステム側で自動的に処理しますので、アプリケーションから直接読み込むことはできません。DL 拡張バナーデータもシステム側で自動的に処理されますが、アプリケーションから読み込むことができます。

BOSS には、NS アーカイブに含まれているおしらせデータを処理しない(おしらせを表示しない)ようにするためのフラグ(OptOut フラグ)を、BOSS ストレージごとに設定することができます。

図 4-8. OptOut フラグが有効に設定されている場合の動作



OptOut フラグの設定と取得は、`nn::boss::SetOptoutFlag()` と `nn::boss::GetOptoutFlag()` の呼び出しで行うことができます。

コード 4-38. OptOut フラグの設定と取得

```
nn::Result nn::boss::SetOptoutFlag(bool flag);
nn::Result nn::boss::GetOptoutFlag(bool* pFlag);
```

`nn::boss::SetOptoutFlag()` の引数 *flag* に `true` を渡して呼び出すと OptOut フラグが有効になり、NS アーカイブにおしらせデータが含まれていても無視します(BOSS はそのおしらせデータを破棄します)。おしらせリストの「このソフトからのおしらせを受け取らない」を設定した場合と同じ挙動になります。
なお、現在の設定は `nn::boss::GetOptoutFlag()` で取得することができます。

4.2.2.2. BOSS ストレージ

追加データと DL 拡張バナーデータは、アプリケーションから指定された拡張セーブデータ領域に作成された、BOSS ストレージと呼ばれるアーカイブ領域内に保存されます。BOSS ストレージはアプリケーションの拡張セーブデータ領域内に作成されますが、アプリケーションからは直接アクセスすることができません。拡張セーブデータ領域内のどれだけのサイズの領域を BOSS ストレージとして使用するかは、アプリケーションが BOSS に指定します。

BOSS ストレージがデータで一杯になると、BOSS は古いデータ(シリアル ID の若いデータ)から順に削除して、指定されたサイズ以上の領域を使わないようにします。また、アプリケーションから BOSS ストレージの登録時に「最大データ数」を指定することができます。データを保存すると最大データ数を超えるときは古いデータから順に BOSS が削除し、最大データ数以上のデータが BOSS ストレージ内に保存されないようになります。最大データ数を指定しない場合は 2000 が適用されます。これらのことを考慮し、アプリケーションはダウンロードするデータのサイズや最大データ数に合わせて、BOSS ストレージのサイズやファイル数を指定する必要があります。また、BOSS ストレージが作成される拡張セーブデータは、新規作成時に「拡張セーブデータ内に作成するファイル数の最大値」を設定します。BOSS ストレージ内の NS データの数も、拡張セーブデータ内のファイル数とカウントされますので、「拡張セーブデータ内に作成するファイル数の最大値」を設定する際には、BOSS ストレージ内に保存される NS データの数も考慮した値を設定してください。

注意: 古いデータを削除してもダウンロード対象のデータを格納するだけの空き領域が確保できない場合、タスクは中断されエラー(NSA_ERROR_STORAGE_INSUFFICIENCY)となります。これは以下のケースで発生します。

- ダウンロード対象のデータのサイズが BOSS ストレージのサイズより大きい場合
- ダウンロード対象のデータがダウンロード済みのデータのバージョン更新版であり、旧バージョンのデータのサイズと新バージョンのデータのサイズの合計が BOSS ストレージのサイズよりも大きい場合

前者は当たり前と言えますが、後者はデータのバージョン更新の可能性がある場合には注意してください。ダウンロード済みデータのバージョン更新版のダウンロードでは、新バージョンのデータが一時ファイルとしてダウンロードされたあとに旧バージョンのデータが削除され、一時ファイルのファイル名が正式なファイル名に変更されます。よって一時的に旧バージョンと新バージョンのデータが同時に BOSS ストレージに存在することになりますので、それらの合計サイズが BOSS ストレージサイズ未満である必要があります。

なお、BOSS ストレージのサイズをダウンロードする NS データの最大サイズの 2 倍以上とすれば、このエラーになることはありません。

ダウンロードされた NS データは、振り分けられる前にアプリケーションの BOSS ストレージに一時的に保存されます。これにより、図 4-7 のようにおしらせデータはシステムの BOSS ストレージに振り分けられますが、一時的にはアプリケーションの BOSS ストレージに保存されることになります。最大データ数が N の BOSS ストレージに N 個の NS データが存在する場合、そこにおしらせデータをダウンロードすると一時的に BOSS ストレージに保存するために既存の NS データを一つ削除し、そのあとに振り分けで削除されるため、ダウンロード完了後のアプリケーションの BOSS ストレージの NS データは(N - 1)個になります。おしらせデータを配信するタスクについては、この挙動を考慮して BOSS ストレージの設定を決めてください。

補足: 一時的に BOSS ストレージに保存されているおしらせデータは、すぐにシステムの拡張セーブデータに移動されます。タスクを削除して再登録しない限りは以前にダウンロードしたときの情報が残っているので、シリアル ID とバージョンの比較が行われ、前回受信したおしらせを再受信することはありません。

注意: NS データも含んだ拡張セーブデータ内のファイル数が最大値に達してしまうと、それ以降にダウンロードされた NS データは保存できず、各タスクの実行結果がエラーとなります。

BOSS ストレージに保存される NS データには、ファイルの先頭に 52 Byte のヘッダ部が付与されています。そのため、BOSS ストレージのサイズを指定する際には、各 NS データがヘッダ部を持っていることを考慮してください。

BOSS ストレージはあくまでも BOSS の作業領域ですので、データが自動的に削除される可能性があります。そのため、追加データを永続化したい場合は、BOSS ストレージ内の NS データとして保存するのではなく、BOSS 経由で NS データを読み込んで、拡張セーブデータ領域にファイルとしてコピーすることを強く推奨します。また、多数のデータが BOSS ストレージ内に保存されている場合、BOSS ストレージ内の全データを対象とした処理の効率が低下してしまいます。そのため、使い終わったデータは BOSS ストレージ内から削除することも推奨します。上記のように NS データを BOSS ストレージ外に永続化することで、そのデータを BOSS ストレージ内から削除することができます。

シリーズタイトルで拡張セーブデータ領域を共有している場合は、自身でダウンロードしたデータだけでなく、シリーズに属するタイトルがダウンロードしたデータにもアクセスすることができます。

弊社に BOSS サービスの使用を申請していただき、申請に対してアプリケーションごとに専用の URL を提供します。通信には HTTPS 通信を使用し、CA 証明書は 3DS に内蔵されている証明書を使用します。

BOSS サービスを使用するアプリケーションに対しては、申請どおりの設定でタスクが登録されているかどうかが、ロットチェック時に検査されます。

> **注意:** NS データのシリアル ID には、アプリケーション単位で(シリーズタイトルの場合はシリーズ単位で)一意の値を指定してください。

おしらせのみをダウンロードする場合

おしらせデータはシステムの拡張セーブデータに保存されるため、アプリケーションがおしらせデータのみをダウンロードする場合は、そのアプリケーションの BOSS ストレージには NS データは貯まりません。ただしこの場合でも、おしらせデータを一時的に保存できるだけのサイズを持った BOSS ストレージの登録が必要となります。これは、NS データが BOSS ストレージに一時的に保存されたあと、ハッシュや署名の検証が行われるためです。それらが成功して初めて、NS データは保存されるべき BOSS ストレージに移動されます。つまり、タスクを登録したアプリケーション用のデータであれば、保存されるべき BOSS ストレージは同じですので、NS データはそのまま残ります。おしらせデータの場合は、システムの拡張セーブデータに移動されます。

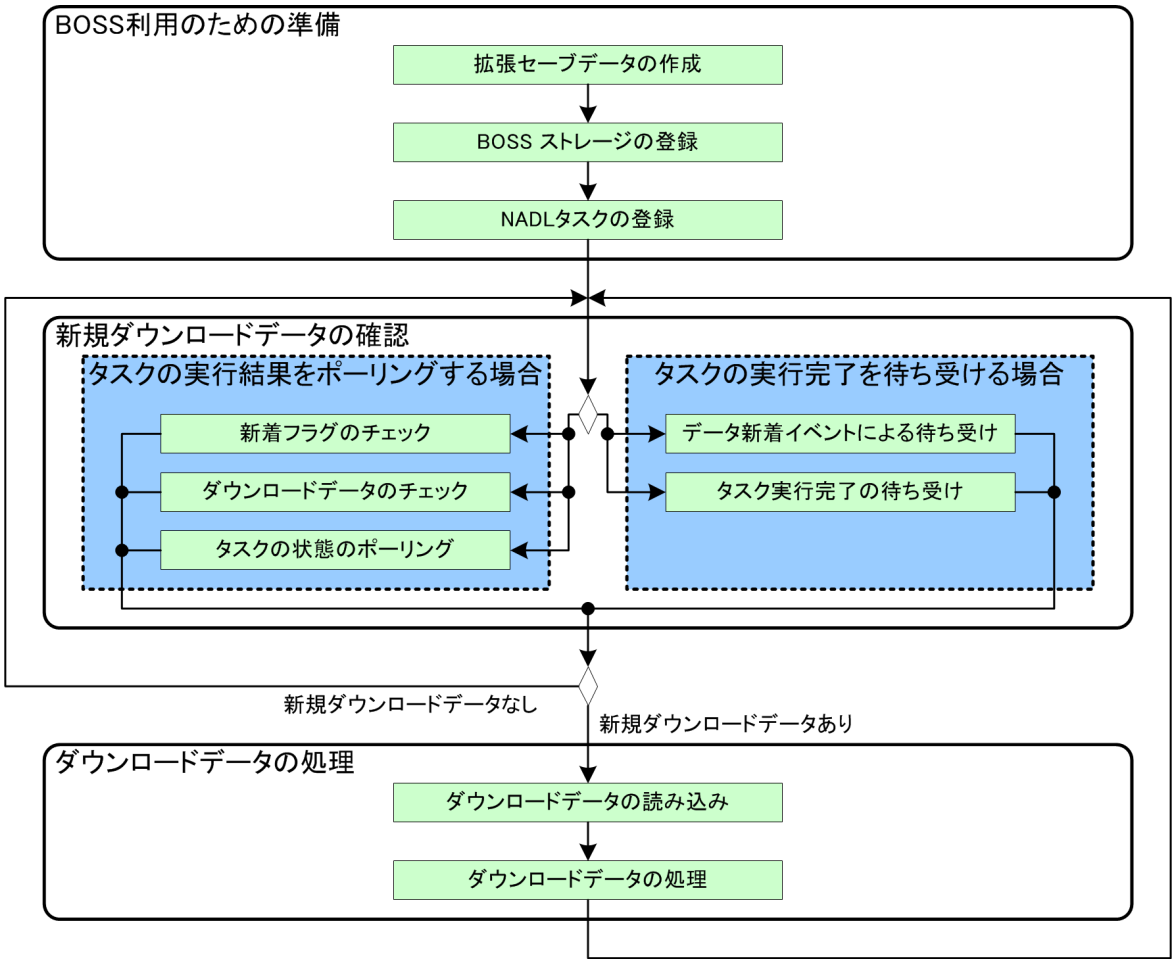
最大サイズのおしらせデータ(50 KByte の添付データを利用)をダウンロードするために必要な BOSS ストレージのサイズは 60 KByte です。

> **注意:** おしらせを配信するためだけに拡張セーブデータを使用する場合は、定期的(起動時など)に拡張セーブデータが正常にマウントできるかどうかを確認してください。これはダウンロードされたデータが拡張セーブデータに一時的に保存されるため、拡張セーブデータが破壊されていたり、削除されていたりするとタスクが残っていてもおしらせを受け取ることができなくなってしまうためです。

4.2.2.3. NADL タスクの処理に必要な手順

BOSS に NADL タスクを登録し、ダウンロードされたデータを処理するまでに必要な手順は以下のようになっています。

図 4-9. NADL タスクの処理に必要な手順



図中の各処理の概要は以下のとおりです。

表 4-18. NADL タスクの各処理の概要

処理	概要
拡張セーブデータの作成	NADL タスクでダウンロードしたデータを保存する BOSS ストレージを登録するために、拡張セーブデータを作成する必要があります。 拡張セーブデータの作成方法については、「3DS プログラミングマニュアル - システム編」を参照してください。
BOSS ストレージの登録	NADL タスクでダウンロードしたデータを保存する BOSS ストレージを BOSS に登録します。 詳細については、「4.2.2.4. BOSS 利用のための準備」を参照してください。
NADL タスクの登録	NADL タスクの動作設定を行い、BOSS に登録します。 詳細については、「4.2.2.5. NADL タスクの登録」を参照してください。
新着フラグのチェック	ライブラリで管理されている新着フラグをチェックします。 詳細については、「4.2.2.6. 新着フラグのチェック、データ新着イベントによる待ち受け」を参照してください。
ダウンロードデータのチェック	ダウンロードされたデータの一覧を取得し、新たにダウンロードされたデータが存在するかどうかをチェックします。処理済みのデータに対して、処理の都度削除する、シリアル ID を記録する、既読設定を行うなどの対応をアプリケーションで行う必要があります。 詳細については、「4.2.2.7. ダウンロードデータのチェック」を参照してください。

タスクの状態のポーリング	NADL タスクの状態をポーリングし、状態が完了になっているかをチェックします。 詳細については、「4.2.1.5. タスクの情報」を参照してください。
データ新着イベントによる待ち受け	ライブラリにイベントを登録し、そのイベントがシグナル状態になるのを待ち受けます。 詳細については、「4.2.2.6. 新着フラグのチェック、データ新着イベントによる待ち受け」を参照してください。
タスク実行完了の待ち受け	NADL タスクをフォアグラウンドで実行し、その完了を待ち受けます。 詳細については、「4.2.1.4. タスクの登録と実行」を参照してください。
ダウンロードデータの読み込み	BOSS ストレージに格納されている NS データにアクセスし、アプリケーションで用意したバッファに読み込みます。 詳細については、「4.2.2.8. ダウンロードデータの読み込み」を参照してください。
ダウンロードデータの処理	取得したダウンロードデータをアプリケーションで処理します。

4.2.2.4. BOSS 利用のための準備

NADL タスクを登録・実行するには、BOSS ストレージを先に登録しておかなければなりません。

コード 4-39. BOSS ストレージの登録と登録解除

```
nn::Result nn::boss::GetStorageInfo(size_t* pStorageSize=NULL);
nn::Result nn::boss::GetStorageInfo(size_t* pStorageSize, u16* pEntryCount);
nn::Result nn::boss::RegisterStorage(
        bit32 storageId, size_t storageSize,
        nn::boss::StorageType storageType=nn::boss::USER_EXT_SAVEDATA_STORAGE);
nn::Result nn::boss::RegisterStorage(
        bit32 storageId, size_t storageSize, u16 entryCount,
        nn::boss::StorageType storageType=nn::boss::USER_EXT_SAVEDATA_STORAGE);
nn::Result nn::boss::UnregisterStorage(void);
```

BOSS ストレージの登録を行う前に、`nn::boss::GetStorageInfo()` を呼び出して BOSS ストレージが作成済みかどうかを確認してください。未作成の場合、`nn::boss::ResultStorageNotFound` が返されます。なお、*`pStorageSize`* と *`pEntryCount`* にはそれぞれ、BOSS ストレージのサイズと最大データ数を格納するための変数へのポインタを指定してください。

`nn::boss::RegisterStorage()` の引数 *`storageId`* で指定された拡張セーブデータ領域に、全ファイルサイズの合計が *`storageSize`* バイトまで保存可能な BOSS ストレージが作成されます。また、引数 *`entryCount`* を持つオーバーライドでは BOSS ストレージ内に登録可能なデータの最大数を指定することができます。

`nn::boss::UnregisterStorage()` は BOSS ストレージの登録を解除します。アプリケーションがアクセス権を持たない拡張セーブデータ領域に BOSS ストレージを登録しようとした場合は `nn::boss::ResultStorageAccessPermission` のエラーが返されます。

1 つのアプリケーションで登録することのできる BOSS ストレージは 1 つです。アプリケーションが BOSS ストレージとして用いる拡張セーブデータ領域を切り替えたい場合は、`nn::boss::UnregisterStorage()` を呼び出して登録を解除したあとに、再度 `nn::boss::RegisterStorage()` を呼び出す必要があります。

注意: `nn::boss::UnregisterStorage()` で BOSS ストレージの登録解除をした後、登録されたタスクが残っていると、以下のような挙動となります。

- BOSS ストレージがないままタスクが実行され、ダウンロードデータは破棄されます。以降のタスク実行では重複ダウンロード防止の処理が働くことがあります。
  アプリケーションが配信データを受け取っていないとしても、重複ダウンロード防止の処理が働いている限りは、該当の配信データを受信することができません。タスクの再登録を行うと再受信することができます。
- BOSS ストレージを再登録した後でも、以前から残っていたタスクは前の BOSS ストレージサイズを覚えており、それに従います。
  以前の BOSS ストレージサイズのタスクと、新しい BOSS ストレージサイズのタスクが混在している場合、現在の BOSS ストレージサイズとは異なるサイズでデータが押し出されて消えてしまう可能性があります。

上記の制限があるので、BOSS ストレージの登録解除を行う際は、あわせて全てのタスクの登録解除を行うことを強く推奨します。

BOSS ストレージのサイズ変更は、以下の手順で安全に行うことができます。

1. 全てのタスクの登録解除を行う
2. BOSS ストレージの登録解除を行う
3. 新しいサイズで BOSS ストレージの登録を行う
4. タスクの再登録を行う

注意: BOSS ストレージとして用いる拡張セーブデータ領域を切り替えると、切り替え前の BOSS ストレージに保存されている NS データにアクセスすることができなくなります。

BOSS ストレージは 1 つの拡張セーブデータ領域に作成したものを使い続けることを推奨します。

補足: 拡張セーブデータ領域へのアクセス方法は「3DS プログラミングマニュアル - システム編」を参照してください。

「おしらせデータ」については「アプレット仕様書」を、「DL 拡張バナーデータ」については「Banner オーバービュー」および「ctr_makedlexbanner」を参照してください。

ユーザーによる拡張セーブデータ削除への対応について

ユーザーが「本体設定」で拡張セーブデータを削除する可能性があることに注意しなければなりません。削除された場合、アプリケーションが BOSS を再度利用するためには拡張セーブデータを再作成する必要があります。そのため、BOSS を利用するアプリケーションは必ず、ユーザーによる拡張セーブデータの削除に対応してください。たとえば、アプリケーション起動時に拡張セーブデータが存在しているかを確認し、存在しない場合はユーザーに拡張セーブデータを作成するかどうかを確認するような処理が必要となります。

登録できる BOSS ストレージの最大数と自動登録解除

システムに登録できる BOSS ストレージ の数は最大で127個です。この数はすべてのアプリケーション（内蔵アプリケーション含む）で登録された BOSS ストレージの総計の最大値です。登録されている BOSS ストレージが最大数に達している状態で、アプリケーションが新たに BOSS ストレージを登録しようとした場合、BOSS は「タスク登録のないアプリケーションの BOSS ストレージ」を自動的に登録を解除して、新たな BOSS ストレージを登録します。タスク登録のないアプリケーションが存在しない場合は、タスクの消尽回数と実行間隔の乗算結果がもっとも小さなタスクの登録解除をし続け、タスク登録のないアプリケーションが現れた時点で、そのアプリケーションの BOSS ストレージを登録解除し、新たな BOSS ストレージを登録します。

上記の自動登録解除により、アプリケーションを長時間起動しなかった場合には、以前登録した BOSS ストレージが BOSS

によって登録解除されている可能性があります。その間、タスク登録も解除されているのでタスク実行はされません。そのため、BOSS を利用する場合にはアプリケーション起動時に BOSS ストレージが登録されているかを確認してください。

4.2.2.5. NADL タスクの登録

NADL タスクの動作設定は nn::boss::NsaDownloadAction クラスで行います。クラスのインスタンスを生成したあと、Initialize() による初期化時に URL を指定します。

コード 4-40. nn::boss::NsaDownloadAction クラスの初期化

```
class nn::boss::NsaDownloadAction : public nn::boss::TaskAction
{
    nn::Result Initialize(const char* pUrl);
}
```

pUrl には、ダウンロードする NS アーカイブの URL を指定します。指定可能な URL は終端の NULL 文字を含めて 512 文字までです。

NS アーカイブの URL は、以下の書式となっています。

https://<CDN ドメイン>/<環境 ID>/<共通パラメータ>/<BOSS コード>/<タスク ID>/<任意のファイル名>

言語別に配信する場合は、言語情報(ISO 639-1 alpha-2)を含めた書式である必要があります。

https://<CDN ドメイン>/<環境 ID>/<共通パラメータ>/<BOSS コード>/<タスク ID>/<言語情報>/<任意のファイル名>

言語別かつ繁体字と簡体字の両方でおしらせを配信する場合は、言語情報に加えて、国情報(ISO 3166-1 alpha-2)を含めた書式である必要があります。これは、繁体字と簡体字が ISO 639-1 alpha-2 ではどちらも中国語(zh)であるため、言語情報だけでは BOSSデータサーバーがどちらの文字体で配信すべきかを判断できないからです。

https://<CDN ドメイン>/<環境 ID>/<共通パラメータ>/<BOSS コード>/<タスク ID>/<国情報>/<言語情報>/<任意のファイル名>

注意: ユーザーが本体設定で言語設定や国設定を変更した場合、タスクの再登録を行わなければ、言語別の配信を行っているタスクは本体設定と異なる言語のデータをダウンロードすることになります。

補足: 言語別の配信を行う場合は OMAS で言語別配信の利用を申請する必要があります。さらに、繁体字と簡体字の両方で配信する場合は利用申請の際に使い方の欄にその旨を記載してください。

おしらせリストの表示は、中国リージョン(中国)の本体では簡体字、台湾リージョン(台湾・香港)の本体では繁体字で行われます。

NsaDownloadAction クラスは nn::boss::TaskAction クラスを継承しており、タスク共通の動作設定を行うことができます。

表 4-19. nn::boss::NsaDownloadAction クラスで指定可能なプロパティ識別子

プロパティ識別子	型	プロパティ
ACTION_URL	char[MAX_URL_LENGTH]	接続先の URL
ACTION_HTTP_HEADER	struct httpRequestHeader[MAX_HTTP_HEADER]	追加の HTTP リクエストヘッダ
ACTION_CLIENT_CERT	u32	機器内蔵のクライアント証明書 ID
ACTION_ROOT_CA	u32	機器内蔵のルート CA 証明書 ID
ACTION_AP_INFO	ApInfoType	HTTP リクエストに付与される AP 情報
ACTION_CLIENT_CERT_NUM	u32	機器内蔵のクライアント証明書の登録数
ACTION_ROOT_CA_NUM	u32	機器内蔵のルート CA 証明書の登録数

以下に、NADL タスク特有の動作設定を説明します。

使用可能なルート CA 証明書

NADL タスクのルート CA 証明書はライブラリが自動で登録しますので、アプリケーションでの登録は不要です。

ライブラリによって登録されるルート CA 証明書は、製品向けの本配信用 BOSSデータサーバーへの接続で使用する CA ルート証明書と、テスト配信用 BOSSデータサーバーへの接続で使用する CA ルート証明書の 2 つです。なお、本配信データは BOSSデータサーバーから製品実機でのみダウンロードすることができます。ただし、BOSS ライブラリのサンプルデモは例外として、開発機で本配信用のデータをダウンロードすることができます。テスト配信データは本体設定で専用の DNS サーバーを設定した開発機および製品実機でのみダウンロードすることができます。DNS サーバーの IP アドレスについては BOSSデータサーバーの「BOSS 管理ツール」にてテスト配信を行うと確認できます。

4.2.2.6. 新着フラグのチェック、データ新着イベントによる待ち受け

新たなデータがダウンロードされたかどうかの確認には、以下の関数を使用することができます。

コード 4-41. 新着データの確認

```
nn::Result nn::boss::GetNewArrivalFlag(bool* pFlag);
nn::Result nn::boss::RegisterNewArrivalEvent(nn::os::Event* event);
```

`nn::boss::GetNewArrivalFlag()` は、新たなデータが存在する場合に *pFlag* に `true` を格納します。

`nn::boss::RegisterNewArrivalEvent()` は、新たなデータがダウンロードされると *event* に渡されたイベントクラスをシグナル状態にします。イベントクラスは事前にアプリケーションで初期化してください。

> **補足：** ダウンロードした NS アーカイブの新着フラグが OFF（無効）の場合、そのデータは新着扱いにはなりません。そのようなデータがダウンロードされても、`GetNewArrivalFlag()` で取得するフラグは `false` のままですが、`RegisterNewArrivalEvent()` に渡されたイベントはシグナル状態になります。

4.2.2.7. ダウンロードデータのチェック

タスクの実行が完了した時点で、タスクに指定されていた拡張セーブデータ領域の BOSS ストレージにダウンロードされた NS データが保存されています。BOSS ストレージに保存されている NS データにアクセスするには、まず NS データのシリアル ID の一覧を取得しなければなりません。

コード 4-42. NS データのシリアル ID 一覧の取得

```
nn::Result nn::boss::GetNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
nn::Result nn::boss::GetNewDataNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
```

nn::boss::GetNsDataIdList() と nn::boss::GetNewDataNsDataIdList() は、アプリケーションから読み込むことのできる NS データの一覧を取得します。前者はシリアル ID の降順ですべてを、後者は未読分のみ取得します。

これらの関数を呼び出したときは、アプリケーションのアイコンに表示されていた新着マークがクリア(nn::boss::GetNewArribalFlag() で取得される新着フラグが false に設定)されます。新たにダウンロードされた NS データは未読となります。未読・既読の設定は nn::boss::NsData::SetReadFlag() で行い、アプリケーションで既読に設定しない限り、NS データは未読のままです。

> **補足:** ダウンロードデータの処理が必要かどうかをシリアル ID の一覧で判断する場合は、データを処理したあとに毎回削除する、処理済みのシリアル ID を記録しておく、NS データを既読に設定するなどの対応が必要です。
>
> シリーズタイトルで拡張セーブデータ領域を共有している場合、シリーズのあるタイトルが登録したタスクでダウンロードされた NS データは、シリーズ内のほかのタイトルからもアクセスすることができます。詳細については「NS データの共有」を参照してください。
>
> これらの関数はBOSS ストレージ内の全データに対して検索を行いますので、BOSS ストレージ内に保存されているデータの数が多いほど処理に時間がかかります。アプリケーションの処理効率を高めるためにも、BOSS ストレージ内の使い終わったデータは逐次削除することを推奨します。

どの関数も *dataType* には取得の対象とする NS データの検索条件を指定し、*pNsDataId* には一覧を格納するための nn::boss::NsDataIdList クラスのインスタンスへのポインタを指定します。

NS データの検索条件は以下のように指定します。

すべての NS データを検索対象にする場合は DATA_TYPE_ALL を指定します。

それ以外の場合は、グローバルデータタイプ(データ種別)とプライベートデータタイプの論理和で検索対象を指定します。グローバルデータタイプが一致し、かつプライベートデータタイプによるマスク処理でも取得対象と判断された NS データの ID のみが一覧に格納されます。

グローバルデータタイプ(データ種別)には以下の 3 種類があります。

表 4-20. NS データの種別

定義	説明
DATA_TYPE_APPDATA	追加データ
DATA_TYPE_NEWS	おしらせに表示されるデータ(アプリケーションでは取得できません)
DATA_TYPE_EXBANNER	DL 拡張バナーデータ

DATA_TYPE_DATASTORE	DataStore ダウンロードタスクで取得したデータ

プライベートデータタイプは 16 bit の値で、NS データのプライベートデータタイプ(データタイプの下位 16 bit)のマスク値として扱われます。指定したプライベートデータタイプと NS データのプライベートデータタイプの同じ位置のビットが 1 であるものが 1 箇所でもある(つまりプライベートデータタイプ同士の論理積が 0 ではない)場合、その NS データは取得対象になります。たとえば、すべての追加データを取得したい場合は、*dataType* に "DATA_TYPE_APPDATA | 0x0000FFFF" を指定します。NS データのデータタイプを活用することで、アプリケーションは追加データを独自に分類した種類ごとに取得することができます。

> **補足:** SD カードが抜けたなどで BOSS ストレージに正常にアクセスできなかった場合、nn::boss::GetNsDataIdList() は nn::boss::ResultFileAccess エラーを返します。この状況をアプリケーションで復旧させることはできませんので、エラーコードをユーザー向けに表示するようにしてください。このエラーとそのほかの予期せぬエラーに対するエラーコード表示の実装例については、boss のサンプルデモの sample_nadl_simple(CTR-SDK 7.1 以降のもの)を参照してください。

nn::boss::NsDataIdList クラスのインスタンスの生成には、シリアル ID を格納するための u32 型の配列が必要です。配列のサイズが一度の検索で取得することのできるシリアル ID の個数です。

検索結果のシリアル ID すべてをインスタンスに格納することができなかった場合、検索関数は nn::boss::ResultNsDataListSizeShortage を返します。その場合、同じインスタンスで再検索する(再度同じ関数を実行する)ことで、前回取得した分の続きからシリアル ID の一覧を取得することができます。

検索の実行中に新しい NS データが BOSS ストレージに格納された場合、検索関数は nn::boss::ResultNsDataListUpdated を返します。その場合は nn::boss::NsDataIdList インスタンスのメンバ関数 Intialize() を呼び出してリストの初期化を行ってから、再度先頭からシリアル ID の一覧を取得しなおしてください。

最後のシリアル ID まで取得した場合、検索関数の返す nn::Result のインスタンスは IsSuccess() で true を返します。

コード 4-43. nn::boss::NsDataIdList クラス

```
class nn::boss::NsDataIdList
{
    explicit NsDataIdList(u32* pSerial, u16 size);
    virtual ~NsDataIdList(void);
    void Initialize(void);
    u16 GetSize(void);
    u32 GetNsDataId(u16 index);
}
```

インスタンスを生成したあとは、必ず Initialize() で初期化してください。検索に使用したインスタンスを Initialize() で初期化すると、そのインスタンスで再び先頭からシリアル ID の一覧を取得することができます。

取得したシリアル ID の個数は GetSize() で確認することができます。

シリアル ID を取得するには、さらに配列内のインデックスを指定して GetNsDataId() を呼び出します。範囲外のインデックスを指定した場合は INVALID_SERIAL_ID が返されます。

NS データの共有

シリーズタイトルで拡張セーブデータを共有している場合、nn::boss::GetNsDataIdList() と nn::boss::GetNewDataNsDataIdList() は、関数を呼び出したアプリケーション以外のシリーズタイトルが登録したタスクでダウンロードされた NS データも含めて検索を行います。また、nn::boss::NsData クラスを用いて、シリーズのほかのタイトルが登録したタスクでダウンロードされた NS データへのアクセスも可能になります。つまりシリーズタイトル内では、NADL タスクでダウンロードされた NS データが共有されることになります。

> **注意:** シリーズタイトル内で NS データは共有されますので、シリーズタイトル用の NS データのシリアル ID は、シリーズタイトル内で一意(つまりタイトルごとではなく、シリーズ全体で一意)になるように付与してください。

シリーズタイトル内のほかのタイトルの NS データが不要な場合、nn::boss::GetOwnNsDataIdList() と nn::boss::GetOwnNewDataNsDataIdList() は、そのアプリケーション自身が登録したタスクでダウンロードされた NS データのみを検索対象にすることができます。

コード 4-44. NS データのシリアル ID 一覧の取得(アプリケーション限定)

```
nn::Result nn::boss::GetOwnNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
nn::Result nn::boss::GetOwnNewDataNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
```

上記の 2 関数は、nn::boss::GetNsDataIdList() や nn::boss::GetNewDataNsDataIdList() と検索対象が異なるだけで、呼び出した際の動作に変わりません。ただし、NS データが関数を呼び出したアプリケーションのタスクでダウンロードされたものかどうかを確認する処理が追加で実行されるため、多少処理速度が劣ります。

4.2.2.8. ダウンロードデータの読み込み

nn::boss::NsData クラスのインスタンスを作成し、取得した NS データのシリアル ID を引数に Initialize() を呼び出して初期化すると、そのインスタンスを介して NS データにアクセスすることができるようになります。

コード 4-45. nn::boss::NsData クラスの初期化と読み込み関連関数

```
class nn::boss::NsData
{
    nn::Result Initialize(u32 serial);
    nn::Result GetHeaderInfo(HeaderInfoType type, void* pValue, size_t size);
    nn::Result SetReadDataPosition(s64 position, PositionBase base);
    s32 ReadData(u8* pDataBuf, size_t bufLen);
    nn::Result SetReadFlag(bool flag);
    nn::Result GetReadFlag(bool* pFlag);
}
```

NS データに関するプロパティ情報は GetHeaderInfo() で取得できるヘッダ情報から取得します。*type* に指定されたヘッダ種別でアクセスするヘッダ情報が決定します。*pValue* と *size* には、ヘッダ情報を格納する変数とそのバイトサイズ(ヘッダ情報の型により変化)を指定します。

表 4-21. ヘッダ種別

ヘッダ種別	型	説明
NSD_TITLEID	s64	タイトル ID
NSD_FLAGS	u32	NSD フラグ(未使用)
NSD_DATATYPE	DataType	データタイプ
NSD_LENGTH	u32	データ長
NSD_SERIALID	u32	シリアル ID
NSD_VERSION	u32	バージョン番号

NS データの読み込みは ReadData() の呼び出しで行います。*pDataBuf* に読み込みバッファを、*bufLen* にそのサイズを指定してください。返り値は読み込んだバイト数です。0 が返された場合はデータの最後まで読み込みが完了しています。保存領域を確保するために自動的に削除されたデータなど、存在しないデータを読み込もうとした場合はエラーとなり、負の値が返されます。読み込み中のデータが新しいダウンロードデータで上書きされた場合はエラーとなり、NN_BOSS_NSDATA_READ_ERROR_UPDATED が返されます。

また、SetReadDataPosition() は、NS データの読み込み位置を指定することができます(ファイルシステムの Seek() に相当する関数です)。

> **注意:** ファイルシステムの Seek() と同様に、SetReadDataPosition() で設定された読み込み位置がファイルの先頭から 4 の倍数でない場合、NS データの読み込みがかなり遅くなります。

NS データを既読設定にするには、SetReadFlag() を、引数 *flag* に true を指定して呼び出します。NS データが既読かどうかは、GetReadFlag() で確認することができます。*pFlag* に true が返されれば、その NS データは既読です。

NS データの読み込みに関するもの以外には、以下の関数が用意されています。

コード 4-46. 付加情報の設定・取得とデータの削除

```
class nn::boss::NsData
{
    nn::Result SetAdditionalInfo(u32 info);
    nn::Result GetAdditionalInfo(u32* pInfo);
    nn::Result GetLastUpdated(nn::fnd::DateTime* PTime);
    nn::Result Delete(void);
}
```

SetAdditionalInfo() と GetAdditionalInfo() で設定および取得可能な u32 型の値は、NS データにアプリケーションが自由に付加することのできる情報です。付加された情報は、新しいバージョンのデータで上書きされたときに削除されることに注意してください。

GetLastUpdated() を呼び出して、NS データが作成された日時を取得することができます。新しいバージョンのデータで上書きされている場合は、その上書きが実施された日時を取得することになります。

NS データを BOSS ストレージから削除するには、Delete() を呼び出します。明示的に削除を行わなくても、BOSS ストレージが NS データで一杯になる(NS データの合計サイズが nn::boss::RegisterStorage() で指定したサイズ以上になる)と、BOSS が自動的に古いデータ(シリアル ID の若いデータ)から順に NS データを削除し、BOSS ストレージ内の

NS データの合計サイズが nn::boss::RegisterStorage() で指定されたサイズ以上にならないように制御します。ただし、不要な NS データを明示的に削除することで、そのデータを以降の BOSS の処理対象から除外することができ、処理の効率を高めることができます。たとえば、シリアル ID 一覧の取得処理の対象から除外されることで、シリアル ID 一覧の検索処理の効率が上がります。

重複ダウンロードの防止

BOSS は HTTP リクエストの If-Modified-Since フィールドを用いて、同じ NS アーカイブが複数回ダウンロードされるのを防いでいます。

ただし、以下の条件が同時に満たされた場合は、同じ NS アーカイブが再度ダウンロードされてしまいます。

- 前回の NS アーカイブのダウンロード後に、そのタスクで別の NS アーカイブを 50 個以上ダウンロードしている。
- 最後にダウンロードした NS アーカイブよりも新しい NS アーカイブがサーバー上に登録されている。つまりサーバー上の NS アーカイブの更新時刻が、最後にダウンロードした NS アーカイブの LastModifiedTime よりも未来の時刻になっている。

アプリケーションが意識する必要はありませんが、NS アーカイブには個別に ID が付与されており、同じ NS アーカイブが再度ダウンロードされないように制御されています。ID はタスクごとに最新の 50 個までしか記録されないため、異なる NS アーカイブを 50 個ダウンロードしたあとに同じ NS アーカイブが配信されると、そのデータを再度ダウンロードしてしまいます。ただし、NS アーカイブの LastModifiedTime が最後にダウンロードした NS アーカイブよりも未来の時刻に更新されていなければ、If-Modified-Since フィールドによって重複ダウンロードは防止されます。

上記の条件が通常の運用では満たされない場合でも、BOSSデータサーバーへのデータ登録時のオペレーションミスや予期せぬ不具合によって、同じ NS アーカイブや NS データが複数回ダウンロードされてしまう可能性があります。そのためアプリケーションには、このような状況が生じても致命的な不具合とならないように対応してください。たとえば、すでに取得したデータのシリアル ID を記録しておき、同じシリアル ID のデータを再度取得した場合には、そのデータを削除するような対応です。

> **注意:** タスクを削除して再度タスクを登録した場合、以前にダウンロードしたときの情報も削除されるため、基本的には重複受信防止機能が正しく働かなくなることに注意してください。
>
> おしらせを何度も受信しないように、おしらせを受信するタスクはできるだけ削除を行わない実装にしてください。たとえば起動時に毎回タスクの削除と登録を行うような実装は避けてください。

ダウンロードの中断

ダウンロードは電源が切断されたときやバッテリー切れ、通信環境の悪化が原因でアクセスポイントとの接続が切断されたときに中断されます。通常は 10 分後にリトライしますが、通信環境が悪いときには次回のタスク実行時刻までリトライされない可能性があります。

スリープの影響

ダウンロード中にスリープ状態になったときはダウンロードが一度中断されますが、アクセスポイントが見つかればすぐに再開されます。スリープ状態から復帰したときに、スリープ前に同じアクセスポイントに接続されていた場合はそのままダウンロードを継続します。スリープ前にオフラインだった場合は復帰したときに(スリープ中に接続していた)アクセスポイントとの接続が切断されますが、次にアクセスポイントと接続できたタイミングでダウンロードが再開されます。

NS データの破損、改竄

NS データが破損している、もしくは改竄されていることは、以下の手順で検知することができます。

1. nn::boss::GetNsDataIdList() で NS データの ID を取得する。
2. nn::boss::NsData::GetHeaderInfo() もしくは nn::boss::NsData::ReadData() で、使い方が不正

であるエラー

- GetHeaderInfo() の場合、ResultInvalidNsDataValue、ResultIpcNotSessionInitialized、ResultStorageAccessPermission のいずれか以外のエラーが返されたときに NS データが壊れていると判断してください。
- ReadData() の場合、NsData::NN_BOSS_NSDATA_READ_ERROR_READ_DATA が返ったときにデータが壊れていると判断してください。NsData::NN_BOSS_NSDATA_READ_ERROR_GET_HEADER が返った場合には、詳細なエラーを GetHeaderInfo() で取得して破損、改竄の判定を行ってください。

NS データが破損、改竄されていることを検知した場合は、必ず以下のように対応してください。

1. 破損、改竄されていた NS データを nn::boss::NsData::Delete() で削除する。
2. 「おしらせ」が同時に届いている場合や、アプリケーション内で「データ」が届いていることを事前に知らせていた場合は、ユーザーに「データ」が壊れていたので削除する旨を通知する。

なお、NS データを削除しても、「重複ダウンロードの防止」によって、同じタスクで同じ NS アーカイブを再びダウンロードすることは基本的に(条件がそろわない限り)できません。そのため、削除した NS データを再ダウンロードする場合は、対象のタスクを nn::boss::UnregisterTask() で登録解除してから、nn::boss::RegisterTask() で再登録してください。ただし、サーバー上の NS データが更新され、削除したデータとは異なるデータになっている可能性がありますので、その点を考慮して再ダウンロード処理を実装してください。

4.2.3. NSA リスト

NSA リストは、BOSSデータサーバー上にある NS アーカイブの一覧を取得する機能です。いつの間に通信の利用申請時に割り振られる BOSS コードやタスク ID、属性を指定して検索し、条件に合致した NS アーカイブのリストを返します。

> **補足:** BOSS コードや NSA リストの設定方法については「OMAS ヘルプ」を参照してください。

4.2.3.1. NsaList クラス

NSA リストの機能は nn::boss::NsaList クラスで定義されています。

コード 4-47. NsaList クラスのコンストラクタ/デストラクタ

```
class nn::boss::NsaList
{
    NsaList(const char* nsaListFilePath);
    ~NsaList(void);
}
```

コンストラクタの引数 *nsaListFilePath* には、取得した NSA リストの保存先パスを指定します。拡張セーブデータのように、アプリケーションからの書き込みが可能なアーカイブ内のパスを指定してください。

4.2.3.2. NSA リストの取得

指定された BOSS コードとタスク ID の NSA リストを Download() で取得します。即時実行専用タスクとしてフォアグラウンドで処理されるため、事前に AC ライブラリでインターネット接続を確立しておかなければなりません。
取得する際、自動的に本体設定に登録されている国・言語情報をクエリストリングに埋め込みます。

コード 4-48. NSA リストを取得する関数

```
nn::Result nn::boss::NsaList::Download(
        const char* bossCode, const char* taskId,
        const nn::boss::NsaList::SearchAttributes* attributes = NULL,
        u32 waitTimeoutSec = NN_BOSS_NSALIST_WAIT_FINISH_TIMEOUT_DEFAULT_SEC,
        s64 fileSize = NN_BOSS_NSALIST_MAX_SIZE);
nn::Result nn::boss::NsaList::Cancel(void);
```

bossCode には BOSS コードを、*taskId* にはタスク ID を指定します。

waitTimeoutSec には、ダウンロードのタイムアウト時間を秒単位で指定します。引数を省略したときは nn::boss::NN_BOSS_NSALIST_WAIT_FINISH_TIMEOUT_DEFAULT_SEC（現状は 30 秒）が指定されます。0 を指定するとタイムアウトしません。

attributes には、3 つの属性（attribute1～attribute3。各属性は ASCII で 9 文字までの文字列）による検索条件を指定します。Download() で指定する属性と NS アーカイブに指定されている属性が一致するかどうかで取得すべきかが決定されます。それぞれの属性は組み合わせではなく順列として扱われますので、attribute1 が "item" というアーカイブの NSA リストを取得する場合、attribute2 や attribute3 に "item" が指定されていても、attribute1 が "item" でない場合はリストアップの対象にはなりません。さらに attribute1 = "A"、attribute2 = "B"、attribute3 = "C" という条件で NSA リストを取得する場合、条件はすべて and で連結されますので、すべての属性の順列が一致したアーカイブのみがリストアップの対象になります。属性を指定した場合は一致するもののみを取得し、指定しなかった（NULL または空文字列）場合は一致判定を行いません。ただし、アーカイブ側に指定がない属性に対して、属性を指定した Download() を行うと、その NS アーカイブはリストに含まれません。

それぞれの属性で以下の判定を行い、3 属性とも「取得する」と判定された NS アーカイブがリストに追加されます。

表 4-22. 属性の指定と取得判定

アーカイブ側の属性	Download() 側の属性	取得判定
指定なし	指定なし	取得する
指定なし	指定あり	取得しない
指定あり	指定なし	取得する
指定あり	指定あり	一致した場合は取得する

Download() での属性の指定はリスト作成時の抽出条件であり、1 つの属性には 1 つしか条件を指定できません。1 つの属性に複数の条件を指定する場合は、属性を指定せずに取得したあと、リストの内容を解析する必要があります。

fileSize には、作成する NSA リストの最大サイズを指定します。NSA リスト機能では、このサイズの空ファイルを作成したあと、そのファイルに取得された NS アーカイブのリストを書き込みます。省略した場合は、NSA リストの最大サイズである nn::boss::NN_BOSS_NSALIST_MAX_SIZE の値が設定されたことになります。

NSA リストのサイズは以下の式で計算することができます。

NSA リストのサイズ = 54 + 236 * リストに含まれる NS アーカイブの数

最大 1000 個の NS アーカイブの情報を含んだリストを取得しますので、NSA リストの最大サイズは 236,054 Byte となります。最大個数の NS アーカイブの情報を含んだリストが取得される可能性がなく、また NSA リスト用に確保されるファイル容量を削減したい場合のみ、明示的に値を設定してください。

Cancel() で Download() の処理を中断することができます。中断の要求は非同期で処理されるため、この関数は必ず

成功します。

4.2.3.3. 取得結果のチェック

NSA リストが取得できたかどうかは GetResult() で確認することができます。

コード 4-49. 取得結果のチェックに使用する関数

```
nn::boss::TaskResultCode nn::boss::NsaList::GetResult(void);
u32 nn::boss::NsaList::GetHttpStatusCode(void);
```

GetResult() の返り値は NSA リストがダウンロードで使用したタスクの実行結果です。TASK_SUCCESS が返されたときは NSA リストの取得が完了しています。

GetResult() が HTTP_ERROR_* や SSL_ERROR_* を返したときは通信中のエラーが発生しています。また、GetHttpStatusCode() では、サーバーから返ってきた HTTP ステータスコードを取得することができます。エラー時に詳細情報を取得する場合に使用してください。

4.2.3.4. 正当性と更新のチェック

ダウンロードした NSA リストの正当性は CheckValidity() でチェックすることができます。また、NSA リストの更新の確認には、GetDigest() で取得するダイジェスト値を利用することができます。

コード 4-50. 正当性と更新のチェックに使用する関数

```
bool nn::boss::NsaList::CheckValidity(void* pWorkBuf, size_t workBufSize);
nn::Result nn::boss::NsaList::GetDigest(u8* pDigestBuf, size_t digestBufSize);
```

pWorkBuf と *workBufSize* には、作業用バッファとそのサイズを指定します。作業用バッファのサイズは 256 バイト以上でなければなりません。また、取得した NSA リストのサイズが大きい場合は、それ以上のサイズで確保することで処理速度が速くなる可能性があります。CheckValidity() が false を返した場合、リストの内容が壊れている可能性があります。

GetDigest() で取得するダイジェスト値は、NSA リストの更新の確認に利用することができます。ダイジェスト値は英数字 40 文字で返されますので、*pDigestBuf* と *digestBufSize* で指定するバッファは 41 バイト以上のサイズである必要があります。

ダイジェスト値は、同じ属性の指定で取得したリストに更新がなければ同じ値となります。リスト全体の更新を判断することはできますが、NS アーカイブごとの更新の判断はできません。

4.2.3.5. NSA リストの解析

取得した NSA リストの解析は Parse() で行います。

コード 4-51. NSA リストの解析に使用する関数

```
nn::boss::NsaList::ParseResult nn::boss::NsaList::Parse(
        u32* pOutParseCount,
        nn::boss::NsaList::NsaInformation pNsaInformationArray[],
        u32 nsaInformationArraySize,
        void* pWorkBuf, size_t workBufSize, u32 nsaFirstPos = 0);
```

pNsaInformationArray と *nsaInformationArraySize* には、解析結果を格納する NsaInformation 構造体の配列とその要素数を指定します。配列に格納されたリストの件数は *pOutParseCount* に格納されます。

pWorkBuf と *workBufSize* には、解析で使用する作業用バッファとそのサイズを指定します。作業用バッファのサイズは 256 バイト以上でなければなりません。作業用バッファのサイズを大きくすることで、取得した NSA リストの件数が多くなっ

たときの処理速度を速くすることができます。

返り値に 0 が返されたときはすでに全件の解析が完了しています。負の値が返されたときはエラーが発生したことを示しています。正の値が返されたときは *pNsaInformationArray* のサイズが小さく、一度の呼び出しで解析が完了できなかったことを示しています。この場合は *nsaFirstPos* に前回の返り値を指定することで、続きから解析を続行することができます。

NsaInformation 構造体は以下のように定義されています。

コード 4-52. NsaInformation 構造体の定義

```
struct nn::boss::NsaList::NsaInformation
{
    char fileName[32];
    u32  fileSize;
    u32  updateEpochTime;
    char attribute1[10];
    char attribute2[10];
    char attribute3[10];
    u8   caption[150];
};
```

それぞれのメンバに NS アーカイブの情報が格納されています。

表 4-23. NsaInformation 構造体のメンバに格納されている情報

メンバ	格納されている情報
fileName	ファイル名
fileSize	ファイルサイズ
updataEpochTime	更新日時(1970/01/01 00:00:00(GMT)からの経過秒数)
attribute1 ～ attribute3	アーカイブに設定されている属性 1 ～ 属性 3 の内容
caption	説明文(文字コードは UTF-8、最大 50 文字)

NS アーカイブの URL

NADL タスクで指定する NS アーカイブの URL は以下のように構成されています。

https://npdl.cdn.nintendowifi.net/p01/nsa/(BOSS コード)/(タスク ID)/(ファイル名)?(クエリ文字列)

クエリ文字列のパラメータには「lm」があります。このパラメータに更新日時(updateEpochTime の値)を指定した NADL タスクでアーカイブを取得すると、更新日時が異なる場合に 404 (Not Found) を返すようになります。これは NS アーカイブを差し換えて CDN キャッシュと NS アーカイブの内容が一致しない期間が発生したときに、NSA リストで取得した内容とアーカイブの内容に齟齬が生じないようにするために指定します。

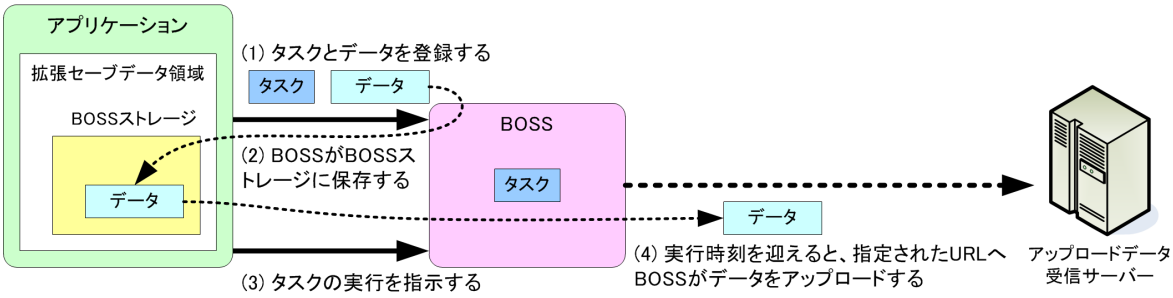
キャッシュ制御への利用

前回の取得時に保存しておいた NsaInformation 構造体の updateEpochTime メンバの内容と今回の内容を比較することで、NS アーカイブの簡易なキャッシュ制御を行うことができます。ただし、3DS 本体で計時している時計は誤差やユーザーの手による変更の影響があるため、更新日時の比較は updateEpochTime の値同士で行う必要があります。

4.2.4. データアップロードタスク

データアップロードタスクは任意のデータサーバーに HTTP/HTTPS 接続でデータをアップロードするタスクです。

図 4-10. データアップロードタスクによるアップロード



データアップロードタスクでは、消尽回数が残っている限りアップロードが実行されますが、一度アップロードに成功するとタスクを再登録しない限り同じデータを再度アップロードすることはありません。
また、すでにタスクに登録されているデータとは異なるデータをアップロードしたい場合、タスクの再利用はできませんので、新たにタスクを登録する必要があります。

タスクの登録直後に実行(開始の指示)を行うように実装してください。タスクの登録中に電源が切断された場合はタスクやデータの情報が不正なものになります。電源が切断されたことで開始できなかったタスクは `nn::boss::Task::GetState()` が `TASK_REGISTERED` を返すかどうかで識別することができます。再度タスクを登録する場合は、`UnregisterTask()` で該当のタスクを削除してから行ってください。

> **補足:** データアップロードタスクを利用するには、弊社への申請が必要です。

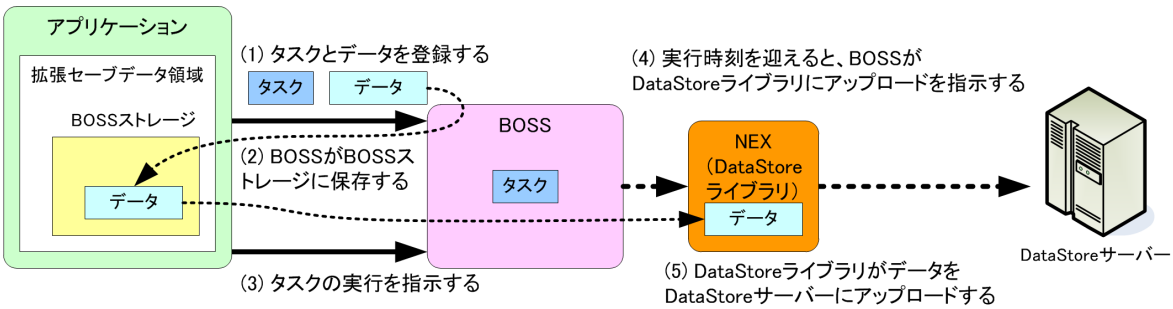
4.2.5. DataStore アップロードタスク

DataStore アップロードタスクは、NEX ライブラリのデータストア機能を利用して、データをアップロードするタスクです。

アップロードしたデータを参照できるユーザーを指定することができ、ユーザー間のデータのやり取りに利用することができます。

DataStore アップロードタスクは、いつの間に通信を経由することで、アプリケーションが起動していないタイミング(スリープ中など)でも NEX ライブラリのデータストア機能を利用することができます。

図 4-11. DataStore アップロードタスクによるアップロード



DataStore アップロードタスクでは、消尽回数が残っている限りアップロードが実行されますが、一度アップロードに成功するとタスクを再登録しない限り同じデータを再度アップロードすることはありません。
また、すでにタスクに登録されているデータとは異なるデータをアップロードしたい場合、タスクの再利用はできませんので、

新たにタスクを登録する必要があります。

タスクの登録直後に実行(開始の指示)を行うように実装してください。タスクの登録中に電源が切断された場合はタスクやデータの情報が不正なものになります。電源が切断されたことで開始できなかったタスクは `nn::boss::Task::GetState()` が `TASK_REGISTERED` を返すかどうかで識別することができます。再度タスクを登録する場合は、`UnregisterTask()` で該当のタスクを削除してから行ってください。

アップロードしたデータがダウンロード可能になるまでには、最大で 20 秒のタイムラグがあります。

> **補足:** NEX ライブラリの機能を利用していますが、アプリケーションに NEX ライブラリ自体を含める必要はありません。
>
> データストア機能の詳細については、NEX ライブラリのドキュメントを参照してください。

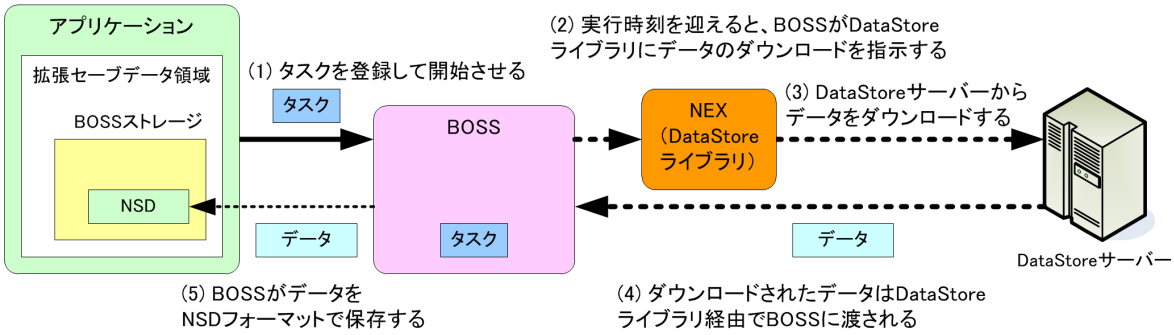
4.2.6. DataStore ダウンロードタスク

DataStore ダウンロードタスクは、NEX ライブラリのデータストア機能を利用して、データをダウンロードするタスクです。

ダウンロードしたデータの情報を記録していますので、ユーザーが参照できるデータのうち、まだダウンロードされていないものだけをダウンロードします。

DataStore ダウンロードタスクは、いつの間に通信を経由することで、アプリケーションが起動していないタイミング(スリープ中など)でも NEX ライブラリのデータストア機能を利用することができます。

図 4-12. DataStore ダウンロードタスクによるダウンロード



ダウンロードされたデータは NSD フォーマットで BOSS ストレージに保存されます。そのため、データの読み込みには NADL タスクでダウンロードしたデータと同様の関数を利用します。また、NADL タスクでダウンロードされたデータと同様に、BOSS ストレージが一杯になったときは古いデータ(シリアル ID の若いデータ)から自動的に削除されます。

タスクの登録時におしらせ発行設定を行うことで、新着データがダウンロードされた際におしらせリストにおしらせを投稿することができます。

> **補足:** NEX ライブラリの機能を利用していますが、アプリケーションに NEX ライブラリ自体を含める必要はありません。
>
> データストア機能の詳細については、NEX ライブラリのドキュメントを参照してください。

4.3. プレゼンス機能

ニンテンドー DS の NITRO-DWC や TWL-DWC ではアプリケーションごとに実装して管理する必要があったフレンド間通信を、3DS では本体機能と SDK でサポートします。

> **補足:** プレゼンス機能については、「システムアプリ・アプレット仕様書」を一読しておくことを推奨します。

4.3.1. 概要

3DS ではすべてのゲームから利用できる共通のフレンドリストを本体に持ち、本体標準の機能として HOMEメニューからほかのユーザーを登録したり削除したりすることができます。したがって、各アプリケーションがフレンドリストを管理するインターフェースを個別に実装する必要はなく、ユーザーがアプリケーションごとに登録をしなおす必要もなくなりました。

また、フレンドの Mii やひとことコメント、今遊んでいるゲームのタイトルなどを、本体機能として取得・表示できるようになりました。アプリケーションはこのうちの一部の情報を API を呼ぶだけで設定・取得でき、自分とフレンドの情報は自動で同期が行われます。このための通信処理はデーモンプロセスがバックグラウンドで行うため、送受信の詳細についてアプリケーションが気にする必要はありません。

ネットワーク機能の拡張ライブラリである NEX と連携し、オンライン対戦中の自分のグループにフレンドを招待することや、逆にフレンドのグループに合流することをサポートする合流機能があります。合流機能の使用方法については、「NEX プログラミングマニュアル - サーバサービス機能編」を参照してください。

4.3.2. ユーザーアカウント

プレゼンス機能は独自のアカウントを持ち、そのアカウント内の情報をフレンドと共有します。アカウントの ID はシステムから自動で付与されます。

4.3.2.1. ユーザー ID の種類

プレゼンス機能のユーザーを識別する ID として、**プリンシパル ID** と**ローカルフレンドコード**の 2 種類の ID があります。どちらの ID も、本体初期化ごとに別の値が割り振られます。

プリンシパル ID

フレンドサーバーに初回接続したときに割り振られる、完全固有の ID です。他者と衝突することはありませんが、フレンドサーバーに初めて接続するまでは無効な値となっており、使用することができません。

ローカルフレンドコード

本体に固有の ID を元に生成される、プレゼンス機能用のユーザー識別 ID です。発生頻度は極めて低いものの、衝突を完全に防ぐことはできません。

また、これらの ID を元に構成される副次的な ID として、**フレンドキー**と**フレンドコード**があります。

フレンドキー

プリンシパル ID とローカルフレンドコードを包括的に扱う概念です。原則的には、プリンシパル ID が発行されるまではローカルフレンドコードを使用し、発行後はプリンシパル ID を使用します。

フレンドコード

プリンシパル ID から生成される、人間が扱うための 12 桁の数字です。プリンシパル ID と一対一で相互変換可能で、簡易な誤り検出能力を持ちます。

4.3.2.2. アカウント内情報

プレゼンス機能を介してフレンドに公開したりフレンドから取得できたりする情報には、以下のようなものがあります。

- フレンドキー
- プレイ中ゲームのタイトル
- ゲームモード説明文字列
- プレイ中ゲームの合流情報(NEX との連携用)
- Mii
- 表示名(Mii の名前)
- 本体プロフィール情報(国や地域設定など)
- フレンド関係
- お気に入りゲーム
- ひとことコメント

アプリケーションはこのうちの一部を設定・取得することができます。また、アプリケーションが直接設定・取得できない項目でも、HOMEメニューなどの本体機能で設定・確認が可能なものがあります。

表 4-24. プレゼンス情報の設定と取得

項目	自分の情報の設定・編集		フレンドの情報の取得・確認	
	アプリケーション	本体機能	アプリケーション	本体機能
フレンドコード	不可	不可	不可	可
フレンドキー	不可	不可	可	不可
プレイ中ゲームのタイトル	不可	可	不可	可
ゲームモード説明文字列	可	不可	不可	可
プレイ中ゲームの合流情報	可	不可	可	不可
Mii	不可	可	可	可
表示名(Mii の名前)	不可	可	可	可
本体プロフィール情報 (国や地域設定など)	不可	可	可	不可
フレンド関係	不可	可	可	可
お気に入りゲーム	不可	可	不可	可
ひとことコメント	不可	可	不可	可

4.3.3. フレンドリストの管理

3DS では、フレンドリストへのユーザーの登録や削除はフレンドリスト(システムアプレット)で行われ、本体に保存されます。

このフレンドリストは 3DS のすべてのゲームタイトルに共通で利用することができます。

4.3.3.1. フレンドリストへの登録

フレンドリストにほかのユーザーを登録するには、以下の 3 通りの方法があります。

フレンドコードの入力

フレンドリスト(システムアプレット)で相手のフレンドコードを入力することで、その相手をフレンドリストに登録することができます。入力時にはフレンドコードの有効性をフレンドサーバーに接続して確認するため、インターネットに接続できる環境が必要です。自分のフレンドコードはフレンドリスト(システムアプレット)で確認することができます。

先に相手のフレンドコードを入力した側は、相手が自分をフレンドリストに追加してくれるまでその相手とはフレンド関係成立待ち状態となり、双方向に相手を登録したのが確認された時点でフレンド関係が成立します。

ローカル通信での登録

フレンドリスト(システムアプレット)で「近くの人とフレンド登録」を選択することにより、ローカル通信を利用して相手をフレンドリストに登録することができます。この場合にインターネット接続は必要ありません。

こちらの方法ではその場で双方がフレンドリストに登録されるので、成立待ちをすることなくフレンド関係が成立します。

アプリケーション内での登録

フレンドリストへの登録に必要な情報を送受信し、アプリケーション内でフレンドリストへの登録を行うことができます。

この方法による登録を行う場合は、必ず双方のユーザーの同意を確認した上で行ってください。

> **補足:** アプリケーションでの登録方法については「4.3.9. フレンド登録」を参照してください。 登録に関する UI はアプリケーションで実装する必要があります。

> **注意:** アプリケーション内でのフレンドの登録を、すれちがい通信や事前に相手を特定できない状態でのローカル通信を介して行うことは禁止されています。
>
> 詳細については、ガイドラインの「インターネット通信」にある「フレンド登録」を参照してください。

4.3.3.2. フレンド関係の状態

フレンド関係は、双方が相手をフレンドリストに登録することで成立します。それぞれの登録状況により、フレンド関係は以下のような状態を遷移します。

フレンド関係成立待ち

先に相手をフレンドリストへ登録して、相手が自分をフレンドリストに追加してくれるのを待っている状態です。相手が自分をフレンドリストに追加したことはフレンドサーバーに登録されるので、相手がフレンドリストへ追加後、自分が最初にフレンドサーバーへ接続したときにフレンド関係成立済みへ移行します。

フレンド関係成立済み

双方が相手をフレンドとして登録している状態です。接続状態やプレイ中のゲームタイトルなどをリアルタイムに通知するにはこの状態になっている必要があります。

> **補足:** フレンドサーバーでは、各ユーザーがフレンドリストに登録している他ユーザーのリストを保持します。そして、互いにフレンドリストに登録しているユーザー同士をフレンド関係成立済みとして扱います。したがって、ローカル通信で成立したフレンド関係は、成立後に各々のユーザーが一度でもオンラインになるまで、フレンドサーバーではフレンド関係成立済みとして扱われません。通常、この状態を意識する必要はありませんが、特別に区別する場合にはローカルフレンド状態と呼びます。

フレンド関係解消

フレンド関係が成立していた相手のフレンドリストから、自分が削除された状態です。自分のフレンドリストから勝手に相手が削除されることはありませんが、この状態になると、実際には相手がオンラインになっていたとしても、あたかもオフラインであるかのように見えます。フレンド関係が成立していたときに相手から受け取った情報には引き続きアクセスできます。相手がもう一度自分のフレンドコードを入力することで、フレンド関係成立済みへと戻ります。

4.3.4. 初期化と終了

FRIENDS ライブラリの API を使用するには、事前に `nn::friends::Initialize()` を呼んでライブラリを初期化する必要があります。また、ライブラリをそれ以上使用しない場合は `nn::friends::Finalize()` で終了処理を行ってください。

FRIENDS ライブラリは、内部に初期化回数のカウンタを持ちます。`nn::friends::Initialize()` を複数回呼んだ場合は、ライブラリを完全に終了するまでに同じ回数 `nn::friends::Finalize()` を呼ぶ必要があります。ライブラリが初期化済み状態であるかを調べるには `nn::friends::IsInitialized()` を使用します。

コード 4-53. 初期化と終了の API

```
nn::Result nn::friends::Initialize();
nn::Result nn::friends::Finalize();
bool nn::friends::IsInitialized();
```

4.3.5. オンラインとオフライン

3DS がフレンドサーバーに接続している状態をオンライン、接続していない状態をオフラインと呼びます。オンライン状態では、そのときプレイ中のゲームのタイトルやモードなどがフレンドに公開される一方、同時にオンラインになっているフレンドからも同様の情報を受け取ることができます。オフライン状態でも、フレンドの名前や Mii、プロフィール情報などの一部の情報は取得可能ですが、それらが最新のものであることは保証されません。

4.3.5.1. 自律接続とログイン

3DS がオンライン状態になるには、以下の 2 通りの方法があります。

自律接続

フレンドプレゼンスデーモンは、アプリケーションから明示的な要求がなくても、可能であれば自律的にフレンドサーバーとの接続を試みてオンライン状態になろうとします。この挙動はデーモンマネージャによって管理され、デーモンマネージャの判断によって確認なく停止される可能性があります。

また、アプリケーションが明示的にこの挙動を停止したい場合は、デーモンマネージャにフレンドプレゼンスデーモンを停止させるよう指示する必要があります。

ログイン

アプリケーションが能動的にプレゼンスの機能を使用する場合、自律接続の状態とは関係なくオンライン状態になるためにログイン要求を出すことができます。

ログイン要求を出すには `nn::friends::Login()` を使用します。この関数が成功するとバックグラウンドで接続処理が走り、処理が終了した時点で *pEvent* に渡した `nn::os::Event` オブジェクトがシグナル状態になります。非同期処理の結果は `nn::friends::GetLastResponseResult()` で取得できます。ログイン要求を出した上でオンライン状態になっている場合、`nn::friends::HasLoggedIn()` は `true` を返します。

ただし、オンライン対応タイトルでない場合はログイン要求を出さないでください。ログイン要求を出すには、事前に AC(自

動接続)ライブラリを使用してインターネット接続を確立しておく必要があります。そのため、ログイン要求は、特にインターネット接続を要求する部分でのみ行うようにしてください。

ログインが必要なくなった場合、ログアウトすることでログイン要求を取り消すことができます。また、明示的にログアウトしなくても、FRIENDS ライブラリの終了時やアプリケーションの終了時にもログイン要求は取り消されます。ログアウト後も自律接続が有効な場合はそのままオンライン状態が維持されます。

コード 4-54. ログイン API

```
nn::Result nn::friends::Login(nn::os::Event* pEvent);
nn::Result nn::friends::Logout();
nn::Result nn::friends::GetLastResponseResult();
bool nn::friends::HasLoggedIn();
```

コード 4-55. ログイン処理のサンプルコード

```
nn::Result result;
nn::os::Event event(false);

result = nn::friends::Login(&event);
if (result.IsSuccess())
{
    // Login が成功した場合は非同期処理が発生
    if (event.Wait(nn::fnd::TimeSpan::FromMinutes(2)))
    {
        result = nn::friends::GetLastResponseResult();
        if (result.IsSuccess())
        {
            // ログイン成功
        }
        else
        {
            // エラー処理
        }
    }
    else
    {
        // ログイン処理の失敗が確定したわけではないので要求のキャンセルが必要
        nn::friends::Logout();
    }
}
else
{
    // エラー処理
}
event.Finalize();
```

注意: nn::friends::Login() で開始される非同期処理は、自律接続によってすでにオンライン状態だった場合はすぐに終了しますが、この操作によりオンライン状態へ遷移する場合は数秒から数十秒程度の時間が必要になります。この非同期処理の終了を待たずにログイン要求を取り消す場合は、バックグラウンドではフレンドサーバーへの接続処理が継続中ですので、明示的に nn::friends::Logout() でログイン要求を取り消してください。

4.3.5.2. フレンドサーバーやフレンドとの情報の同期

ユーザーの情報はオンライン時にフレンドサーバーに送られ、フレンドサーバーを介してフレンドに共有されます。

自分の情報の更新はオフラインでも行えますが、その更新はすぐにフレンドサーバーへは送信されません。オフライン中に行われた自分の情報の更新は、フレンドサーバーに接続されてオンラインになったときにフレンドサーバーに送信されて同期されます。

一方、オンライン中に行われた更新は、フレンドプレゼンスデーモンのバックグラウンド動作によって随時フレンドサーバーに送信されます。ただし、同一項目のフレンドサーバーへの送信間隔は 10 秒に 1 回以下になるようにライブラリで制限されているため、これより細かい間隔で情報を更新しても、フレンドサーバーやフレンドに対して随時性のある同期はできません。あまり高い頻度でフレンドサーバーと通信しないでください。詳しくはガイドラインを参照してください。

コード 4-56. 自分情報の更新 API

```
nn::Result nn::friends::UpdateGameModeDescription(
        const char16 description[nn::friends::MODE_DESCRIPTION_SIZE]);
```

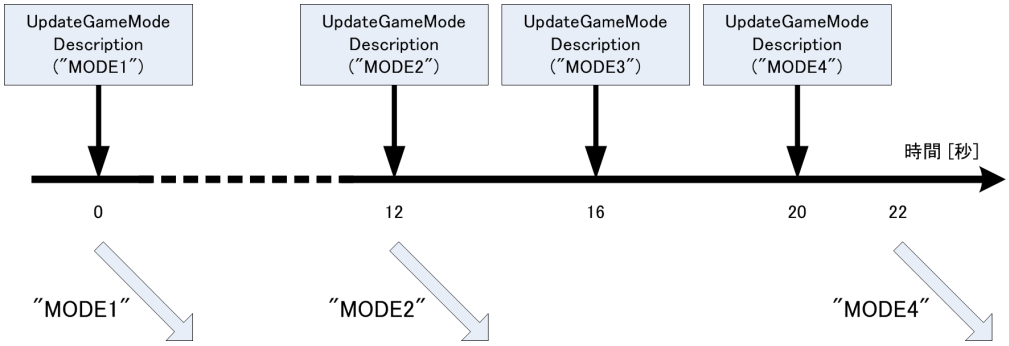
`nn::friends::UpdateGameModeDescription()` は、自分のゲームモード説明文字列を更新します。

description で指定した文字列は、自分のフレンドカードの表示に反映されます。フレンドカードには、最大幅の内蔵フォント(日米欧リージョンは ″%″、そのほかのリージョンはひらがな、漢字、ハングルなど)で 16 文字分の幅で 2 行まで表示されます。なお、フレンドカードで表示することのできる文字は本体のリージョンによって以下のように異なり、表示できない文字は ″?″(0xE011)に変換されます。

表 4-25. フレンドカードに表示することのできる文字

リージョン	表示することのできる文字
日本・米州・欧州	内蔵フォントの日米欧文字に含まれる文字
中国	内蔵フォントの簡体字に含まれる文字
韓国	内蔵フォントのハングルに含まれる文字
台湾	内蔵フォントの繁体字に含まれる文字

図 4-13. 短い間隔で自分のゲームモード説明文字列を更新した場合のバックグラウンド通信の例



ユーザー情報の一部はフレンドサーバーに保存されます。これにより、フレンドと同時にオンラインにならなくても、相手が最後にオンラインだった時点の情報を、自分がオンラインになったときにフレンドサーバーから受け取ることができます。また、最後に受け取ったフレンドの情報は 3DS 本体に保存されるため、自分がオフラインであってもフレンドの情報を取得することができます。ただし、双方がオンラインでない状態で取得したフレンドの情報は、最新のものではない可能性があります。プレイ中のゲームモードなど、自分とフレンドが同時にオンラインになっていないと意味がない情報はフレンドサーバーに保

存されず、双方がオンラインであるときにしか取得できません。

- 双方がオンラインでないと取得できない情報
  - プレイ中ゲームのタイトル
  - ゲームモード説明文字列
  - プレイ中ゲームの合流情報
- いつでも取得できる情報
  - フレンドキー
  - Mii
  - 表示名（Mii の名前）
  - 本体プロフィール情報（国や地域設定など）
  - フレンド関係
  - お気に入りゲーム
  - ひとことコメント

フレンドの情報を取得するための API については、「4.3.6. フレンドの情報の取得」を参照してください。

4.3.5.3. 非公開モード

自分の情報はフレンドリスト（システムアプレット）から非公開に指定することができます。

公開レベルには、

- プレイ中ゲームのみ非公開
- プレゼンス情報全体の非公開

の 2 段階があります。

プレゼンス情報全体を非公開にした場合、自分がオンラインであってもフレンドにはオフラインであるかのように見えます。すなわち、プレイ中のゲームに関する情報はフレンドに送られず、接続状態もオフラインとして扱われます。また、フレンドサーバーにも自分の情報の更新をしないため、Mii や本体プロフィール情報なども最後に公開設定でオンラインになったときに指定していたものが伝えられます。フレンドの情報は、プレゼンス情報を公開しているときと同様に取得することができます。

4.3.5.4. 自分の情報の取得

自分の情報を取得するための API には、以下のものが用意されています。

コード 4-57. 自分の情報を取得する API

```
nn::friends::PrincipalId nn::friends::GetMyPrincipalId();
bool nn::friends::IsMyPreferenceValid();
nn::Result nn::friends::GetMyPreference(
        bool* pIsPublicMode, bool* pIsShowGameName);
nn::Result nn::friends::GetMyProfile(nn::friends::Profile* pProfile);
nn::Result nn::friends::GetMyPresence(nn::friends::MyPresence* pMyPresence);
nn::Result nn::friends::GetMyScreenName(char16 screenName[SCREEN_NAME_SIZE]);
nn::Result nn::friends::GetMyMii(nn::mii::StoreData* pMiiData);
```

フレンドサーバーに一度も接続されていない場合、プリンシパル ID には INVALID_PRINCIPAL_ID が返されます。

フレンドリスト（システムアプレット）を一度も起動していない場合、フレンド同士の通知設定がユーザー自身の手で行われていないため、IsMyPreferenceValid() は false を返します。また、GetMyPreference() の各引数にはデフォルトの値（true）が格納されます。

「じぶんの Mii」を作成していない場合やフレンドリスト（システムアプレット）を一度も起動していない場合でも、GetMyProfile() で取得する本体プロフィール情報には、本体の初回起動時や本体設定での設定内容が反映されます。

「Mii スタジオ」で「じぶんの Mii」を作成していない場合、GetMyScreenName() は先頭が NULL の空文字列(2 文字目以降は不定)を返します。GetMyMii() は空の Mii データを返します。取得した Mii データが有効であるかどうかの判断や表示を行うには、CFL ライブラリを利用しなければなりません。

4.3.6. フレンドの情報の取得

FRIENDS ライブラリを使用してフレンドの情報を取得する場合、フレンドキーもしくはプリンシパル ID でフレンドを指定します。フレンドリストに登録済みのユーザーのフレンドキーは nn::friends::GetFriendKeyList() で取得できますが、通信で外部から受け取ったフレンドキーやプリンシパル ID を指定することもできます。

4.3.6.1. フレンドキーの取得

フレンドリストに登録されたユーザーのフレンドキーを取得するには nn::friends::GetFriendKeyList() を使用します。この関数は、フレンドリストの *offset* 番目から *size* 人分のユーザーのフレンドキーを順番に *pFriendKeyList* が指すバッファへ格納しようと試み、実際に格納したフレンドキーの個数を *pNum* が指すバッファに返します。取得できたフレンドキーの個数が指定した *size* よりも少なかった場合、フレンドキーが格納されなかったバッファの値は保証されません。

コード 4-58. フレンドキーの取得 API

```
nn::Result nn::friends::GetFriendKeyList(
        nn::friends::FriendKey* pFriendKeyList, size_t* pNum,
        size_t offset = 0, size_t size = nn::friends::FRIEND_LIST_SIZE);
```

ローカル通信で得られるローカルフレンドコードは、プライバシーを保護する意味合いからスクランブルがかけられているため、そのままではフレンドを特定するためのローカルフレンドコードとして利用することができません。そのため、FRIENDS ライブラリにはスクランブルを解除するための API が用意されています。

コード 4-59. ローカルフレンドコードのスクランブルを解除する API

```
nn::Result nn::friends::UnscrambleLocalFriendCode(
        nn::friends::LocalFriendCode* pLocalFriendCodeList,
        const nn::uds::ScrambledLocalFriendCode* pScrambledLocalFriendCodeList,
        size_t size = 1);
```

pScrambledLocalFriendCodeList と *size* には、ローカル通信で取得したローカルフレンドコード(スクランブル解除前)のリストとその個数を指定します。

スクランブルを解除した結果は *pLocalFriendCodeList* に格納されますので、*size* で指定した個数を格納できるように指定してください。フレンド以外のスクランブルは解除できません。フレンドリストに登録されていないユーザーのローカルフレンドコードは INVALID_LOCAL_FRIEND_CODE となります。

複数のフレンドが同じローカルフレンドコードを持つ可能性を考慮して、アプリケーションを実装してください。スクランブル解除後のローカルフレンドコードは一意の値ではありませんので、極めて低い確率で同じローカルフレンドコードを持つ複数のフレンドがフレンドリストに登録されている可能性があります。

4.3.6.2. フレンドの情報の取得

フレンドの情報を取得するには、取得 API にフレンドキーもしくはプリンシパル ID のリストを渡します。結果は、渡したフレンドキーもしくはプリンシパル ID のリストと同じ個数・順番で、各関数の第 1 引数に渡したバッファに格納されます。指定したフレンドキーやプリンシパル ID に該当するユーザーがフレンドリストに含まれていなかった場合は、無効なデータが返ります。

> **補足:** ローカルフレンドコードのリストを指定してフレンドの情報を取得する API は用意されていません。ローカルフレンドコードをフレンドキーに格納して渡すことは可能ですが、発生頻度は極めて低いものの衝突を完全には回避できないローカルフレンドコードの性質上、同じローカルフレンドコードを持つ複数のユーザーがフレンドリストに登録されている可能性があります。フレンドリストから取得したフレンドキーであれば、ローカルフレンドコードが等しいユーザーがリスト内に複数いたとしても、その中でプリンシパル ID を持たないユーザーは必ず 1 人以下となる(フレンド関係成立前に衝突した場合、あとから登録したユーザーが残ります)ため、一意にリスト内のユーザーを特定することができます。

コード 4-60. フレンドの情報取得 API

```
nn::Result nn::friends::GetFriendPresence(
        nn::friends::FriendPresence* pFriendPresenceList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendPresence(
        nn::friends::FriendPresence* pFriendPresenceList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
nn::Result nn::friends::GetFriendScreenName(
        char16 (*pScreenNameList)[nn::friends::SCREEN_NAME_SIZE],
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1,
        bool replaceForeignCharacters = true, u8* pFontRegionList = NULL);
nn::Result nn::friends::GetFriendScreenName(
        char16 (*pScreenNameList)[nn::friends::SCREEN_NAME_SIZE],
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1,
        bool replaceForeignCharacters = true, u8* pFontRegionList = NULL);
nn::Result nn::friends::GetFriendMii(
        nn::mii::StoreData* pMiiDataList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendMii(
        nn::mii::StoreData* pMiiDataList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
nn::Result nn::friends::GetFriendProfile(
        nn::friends::Profile* pProfileList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendProfile(
        nn::friends::Profile* pProfileList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
nn::Result nn::friends::GetFriendAttributeFlags(
        bit32* pAttributeFlagsList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendAttributeFlags(
        bit32* pAttributeFlagsList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
```

4.3.7. 通知

フレンドサーバーとの接続状態の変化や、フレンド情報の更新をポーリングすることなく把握するために、アプリケーションが必要とする項目を指定して、フレンドプレゼンスデーモンから変更通知を受け取ることができます。

4.3.7.1. 通知の種類

アプリケーションが受け取れる通知は以下の 9 種類です。変更の種類と、フレンドが関係する通知ではそのフレンドのフレンドキーが一緒に通知されます。通知が届いた場合、それより以前に API から取得したフレンドの情報は古くなっている可能

性がありますので、アプリケーションの都合にあわせて適切なタイミングで取得しなおしてください。

表 4-26. 通知の種類

定義	説明(再取得が必要な API)
NOTIFICATION_ONLINE	自分がオンライン状態になったことを表します。フレンドキーには無効な値が格納されます。
NOTIFICATION_OFFLINE	自分がオフライン状態になったことを表します。フレンドキーには無効な値が格納されます。
NOTIFICATION_FRIEND_ONLINE	フレンドがオンラインになったことを表します。 GetFriendPresence(), GetFriendScreenName(), GetFriendMii(), GetFriendProfile(), GetFriendAttributeFlags()
NOTIFICATION_FRIEND_PRESENCE	フレンドのゲームモードが変更されたことを表します。 GetFriendPresence()
NOTIFICATION_FRIEND_MII	フレンドの Mii が変更されたことを表します。 GetFriendScreenName(), GetFriendMii()
NOTIFICATION_FRIEND_PROFILE	フレンドのプロフィールが変更されたことを表します。 GetFriendProfile()
NOTIFICATION_FRIEND_OFFLINE	フレンドがオフラインになったことを表します。 GetFriendPresence()
NOTIFICATION_BECOME_FRIEND	ユーザーとフレンド関係が成立したことを表します。 GetFriendPresence(), GetFriendScreenName(), GetFriendMii(), GetFriendProfile(), GetFriendAttributeFlags()
NOTIFICATION_INVITATION	フレンドからお誘いを受けたことを表します。 GetFriendPresence()

4.3.7.2. 通知に関する API

フレンドプレゼンスデーモンからの通知を受け取るためには、以下の API を使用します。

コード 4-61. 通知 API

```
nn::Result nn::friends::AttachToEventNotification(nn::os::Event* pEvent);
nn::Result nn::friends::SetNotificationMask(bit32 mask);
u32 nn::friends::GetEventNotification(
        nn::friends::EventNotification* pEventNotificationList,
        size_t size = 1, bool* pHasOverflowed = NULL);
```

イベントの登録

nn::friends::AttachToEventNotification() の *pEvent* に初期化済みの nn::os::Event オブジェクトへのポインタを渡すと、通知発生時にこのイベントがシグナル状態になります。

通知マスクの設定

受け取りたい通知の種類を指定するには、nn::friends::SetNotificationMask() の *mask* に nn::friends::NotificationMask 列挙体の各列挙子の論理和を渡します。ここで有効にしなかった種類の通知は、デーモンから届いても nn::friends::AttachToEventNotification() で渡した nn::os::Event オブジェクトはシグナル状態にならず、nn::friends::GetEventNotification() で取得される通知履歴からも除外されます。

通知内容の取得

通知の内容は、nn::friends::GetEventNotification() で取得できます。実際に取得できた通知の数が返り値になるので、この関数が 0 を返すまで繰り返しコールすれば、デーモンのバッファにたまっている通知を発生順に残らず取得することができます。

> **補足:** 通知は nn::friends::Initialize() で FRIENDS ライブラリを初期化したときからたまり始めます。ある時点から新しく通知を取得し始めたい場合は、nn::friends::SetNotificationMask() の *mask* に 0 を渡してすべての通知を除外した上で nn::friends::GetEventNotification() を 1 回呼べば、過去に届いた通知をすべて消去できます。

> **注意:** 通知履歴は、システムに届くプライベートな通知も含めて 128 件までデーモンのバッファに保存され、あふれた場合は古いものから順に消去されます。通知があふれた場合は、情報の古くなった箇所がわからなくなりますので、速やかに自分の接続状態とフレンドの情報を再取得してください。

4.3.8. エラーハンドリング

FRIENDS ライブラリの API が失敗を返した場合、その nn::Result オブジェクトを nn::friends::ResultToErrorCode() の *result* に渡すことでエラーコードが取得できます。

コード 4-62. エラーコード取得 API

```
u32 ResultToErrorCode(const nn::Result& result);
```

この関数が 0 を返した場合は、エラーコードを表示する必要はありません。また、ログインに失敗したがすぐに自動でリトライするときなど、ユーザーが不都合を感じずにゲームを継続できる場合も、エラーコードを表示する必要はありません。

> **注意:** nn::Result オブジェクトの内容をエラーコードへ変換する際、nn::friends::ResultToErrorCode() は、そのときのデーモンの状態に関する情報を付加することがあります。したがって、エラーコードを取得するときは、失敗の結果を受け取ったあと速やかに nn::friends::ResultToErrorCode() を呼んでください。

4.3.9. フレンド登録

アプリケーション内でのフレンド登録は、nn::friends::GetMyApproachContext() で取得したフレンド登録用の情報を相互に受信し合い、双方のユーザーが受信した情報を nn::friends::AddFriendWithApproach() に渡して呼び出すことで行います。

コード 4-63. アプリケーション内でのフレンド登録に使用する関数

```
nn::Result nn::friends::GetMyApproachContext(
                        nn::friends::ApproachContext* pApproachContext);
nn::Result nn::friends::AddFriendWithApproach(
                        nn::os::Event* pEvent,
                        const nn::friends::ApproachContext& approachContext);
```

これらの関数を使用するには、FRIENDS ライブラリの初期化は必要ですが、フレンドサーバーへのログインは必要ありません。また、事前に nn::cfg::Initialize() で CFG ライブラリの初期化を行っておく必要があります。

GetMyApproachContext() で取得する nn::friends::ApproachContext 構造体は、すでに暗号化されていますので、送受信のためにアプリケーション側で暗号化などを行わなくてもかまいません。

AddFriendWithApproach() の呼び出しが成功する(返された Result の IsSuccess() が true を返す)とフレンドが登録され、非同期処理が発生します。それ以外の Result が返された場合には、非同期処理は発生しません。非同期処理の完了は、*pEvent* に渡したイベントがシグナル状態になることでアプリケーションに通知されます。非同期処理の結果は nn::friends::GetLastResponseResult() で取得しますが、ライブラリの仕様上、複数の非同期処理を同時にハンドリングすることができないことに注意してください。

なお、この関数で登録されたフレンドはローカルフレンドですので、双方がインターネットに接続し、フレンドサーバーでのフレンド関係が成立するまでプレゼンス情報を取得することができません。非同期処理ではフレンドサーバーへの登録処理を試みます。双方のユーザーがオンライン状態で登録処理が成功した場合でも、非同期処理の完了直後はプレゼンス情報を取得できない可能性があります。

受信したフレンド登録用の情報(ApproachContext 構造体)をアプリケーションで表示するために、以下の関数で表示名や Mii のデータを取得することができます。

コード 4-64. フレンド登録用の情報から情報を取得する関数

```
nn::Result nn::friends::GetApproachContextScreenName(
                        char16 screenName[nn::friends::SCREEN_NAME_SIZE],
                        const nn::friends::ApproachContext& approachContext,
                        bool replaceForeignCharacters, u8* pFontRegion);
nn::Result nn::friends::GetApproachContextMii(
                        nn::mii::StoreData* pMiiData,
                        const nn::friends::ApproachContext& approachContext);
```

GetApproachContextScreenName() は、フレンドの表示名を *screenName* に渡したバッファに返します。このとき、*replaceForeignCharacters* に true を指定していた場合は、取得する表示名に含まれている、本体リージョンの標準フォントでは表示できない文字を「?」に置き換えて取得することができます。置き換えを行わず、ほかのリージョンのであっても表示できるようにする場合は false を指定し、*pFontRegion* に u8 型の変数へのポインタを渡してください。*pFontRegion* には、どのリージョンの標準フォントであれば表示可能なのかが nn::friends::FontRegion 列挙体の値で返されます。

GetApproachContextMii() は、フレンドの Mii を表示するための情報を *pMiiData* に渡したバッファに返します。取得した Mii の情報を扱うには、別途「似顔絵ライブラリ」が必要です。

5. 通信補助ライブラリ

無線通信を利用する際にアプリケーションを補助するために、以下のライブラリが用意されています。

- 受信拒否リスト
- NG ワードリスト
- おしらせ
- アカウント

この章では、それぞれのライブラリを使用したアプリケーションの開発に必要な情報とプログラミング手順について説明します。

5.1. 受信拒否リスト

受信拒否リストとは、不適切なユーザー作成コンテンツ(以下 UGC と呼びます)がユーザー同士の直接通信(以下 P2P 通信)によって、広範囲に蔓延してしまうことを抑えるために用意された機能です。

5.1.1. 概要

UGC が特定のサーバー上で管理されている場合は、登録されている UGC の全数チェックを行ったりすることで、管理者が不適切な UGC を削除することができます。しかし、P2P 通信はサーバーを介さずに本体同士が直接通信するため、不適切な UGC に対応することができません。

受信拒否リストはそのような P2P 通信でも、できるだけ不適切な UGC の蔓延を抑えることを目的としています。

受信拒否リストでは、本体に用意されたリスト(ローカル受信拒否リスト)に不適切な UGC の作者を登録することができます。そこに登録されている作者の UGC を表示できなくすることで、不適切な UGC を制限します。

ユーザーはローカル受信拒否リストに対して、HOMEメニューから以下の操作を行うことができます。

- 登録した UGC 作者情報の一括消去(解除)

受信拒否リストは UGC 作者の ID の登録と参照をするためだけのものですので、受信拒否リストに登録されたユーザーが作成した UGC を受け取っても、それが自動的に削除されるわけではありません。そのため、受信拒否リストへの対応が必要な UGC を受け取った場合は、その UGC 作者 ID が受信拒否リストに登録されていないかをアプリケーション自身で確認して、UGC の削除などを行う必要があります。また、受信拒否リストへの UGC 作者 ID の登録もアプリケーションで行う必要があります。

> **補足:** ローカル受信拒否リストは共有拡張セーブデータ領域に保存されます。

5.1.2. 初期化と終了

アプリケーションで受信拒否リストの機能を利用するには、UBL ライブラリを使用します。UBL ライブラリの初期化と終了は、`nn::ubl::Initialize()` と `nn::ubl::Finalize()` で行います。

コード 5-1. UBL ライブラリの初期化と終了

```
void nn::ubl::Initialize(void);
void nn::ubl::Finalize(void);
```

5.1.3. ローカル受信拒否リスト

ローカル受信拒否リストは、受信した UGC の作者を対象とする自分専用のリストで、本体につき 1 つだけ保存されます。ユーザーには、「受信拒否設定」という名称で提供されます。

ローカル受信拒否リストには、ユーザーが不適切なものであると判断した UGC の UGC 作者情報を登録します。ローカル受信拒否リストへの登録は、`nn::ubl::Entry()` を呼び出して行います。

コード 5-2. ローカル受信拒否リストへの登録

```
nn::Result nn::ubl::Entry(u64 id, nn::fnd::DateTime *dt);
```

id には、ローカル受信拒否リストに登録する UGC 作者 ID を指定します。自分自身を登録しようとした場合はエラーが返されます。

UGC 作者 ID は UGC を作成した作者の ID で、受信拒否リストのためにのみ使用される ID です。本体ごとにユニークな 8 Byte の IDで、本体を特定することはできませんが、異なるアプリケーションで作成した UGC でも同じ ID となります。受信した UGC を編集することのできるアプリケーションの場合、編集された UGC からは、その UGC を最初に作成したユーザーの ID（新規作者 ID）ではなく、編集したユーザーの ID（編集者 ID）が登録されます。

> **補足：** アプリケーション側で必要な対応についてはガイドラインの「受信拒否リスト」を参照してください。

dt には、登録日時（現在日時）を指定します。UGC 作者情報には、UGC 作者 ID をローカル受信拒否リストに登録した日時が記録され、ローカル受信拒否リストが一杯になったときに削除する情報の判断に使用されます。

登録可能な UGC 作者情報は最大 1000 件です。登録時に、登録件数が最大数に達している場合は、古いものから自動的に上書きされます。

5.1.4. 受信拒否リスト該当チェック

受信拒否リストが対象とする UGC（ガイドラインの「受信拒否リスト」参照）を受信または表示するときには、その UGC の UGC 作者 ID がローカル受信拒否リストに登録されていないかをチェックしてください。

UGC 作者 ID のチェックは `nn::ubl::IsExist()` で行うことができます。

コード 5-3. 受信拒否リストのチェック

```
bool nn::ubl::IsExist(u64 authorId, u32 titleId, u64 dataId);
```

authorId には、チェックする UGC 作者 ID を指定します。受信した UGC を編集可能ならば、チェックは新規作者 ID と編集者 ID の両方に対して行ってください。

titleId と *dataId* は将来の拡張機能で使用する引数です。現在は未対応の機能ですので、指定しても無視されます。

受信拒否リストに該当した UGC は、アプリケーションで削除または表示されないようにしてください。

5.1.5. UGC 閲覧モード

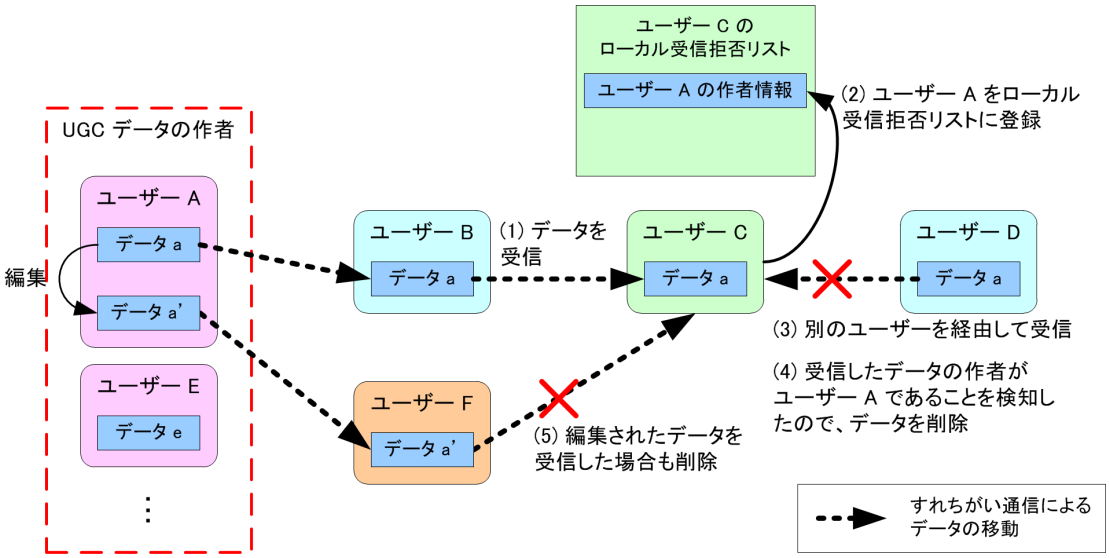
UGC 閲覧モードは、アプリケーション内で表示される可能性がある UGC の内容を閲覧できるモード(閲覧画面)です。UGC を作成することのできるアプリケーションには、必ず UGC 閲覧モードを用意してください。

UGC 閲覧モードでは、アプリケーションで表示される、まだローカル受信拒否リストに登録されていない作者の UGC を表示し、その中からユーザーが不適切だと判断した UGC をローカル受信拒否リストに登録できるように実装してください。

5.1.6. ローカル受信拒否リスト登録の流れ

下図は、すれちがい通信によるデータ受信を例にして、ローカル受信拒否リストの登録の流れを示したものです。

図 5-1. ローカル受信拒否リストの登録の流れ



1. ユーザー A が作成した UGC を含むデータ(「データ a」)を、ユーザー B からユーザー C が受信したとします。
2. ユーザー C がアプリケーション内で「データ a」を見て不適切だと感じたため、UGC 閲覧モードから「データ a」を選択します。アプリケーションは以下のようなメッセージを表示して、ローカル受信拒否リストに登録するかどうかをユーザーに確認します。「はい」が選択されたときは、ローカル受信拒否リストにユーザー A を登録します。
「受信拒否設定に登録しますか？ はい いいえ」
3. その後、同じ「データ a」をユーザー D から受信したとします。
4. すれちがい通信の場合、「データ a」を受信したあとでユーザー C が対応するアプリケーションを起動したときに、「データ a」の作者ユーザー A がローカル受信拒否リストに登録されていることを検知して、「データ a」が削除されます。(ガイドラインの「受信拒否リスト」参照)
5. そのあと、ユーザー A が "編集" した「データ a'」をユーザー F から受信したとしても、「データ a'」の編集者 ID がローカル受信拒否リストに登録されているため、「データ a'」は削除されます。

5.2. NG ワードリスト

インターネット経由の通信だけでなく、すれちがい通信のようなローカル通信においても、文字列情報を含むユーザー作成コンテンツ(以下 UGC と呼びます)を送受信する際には、必ず NG ワードリストによる NG ワードのチェックを行ってください。

NG ワードのチェックは、リージョンや言語設定が異なる本体と通信できる場合でも、送信側もしくは受信側のどちらか一方だけで構いません。また、送信側と受信側の両方でチェックしても構いません。

CTR-SDK では、NG ワードのチェックを行う NGC ライブラリを用意しています。NGC ライブラリは Wii の DWC ライブラリな

どとは異なり、サーバーを仲介せずに問題のある語句かどうかをチェックすることができます。

5.2.1. 初期化と終了

NG ワードのチェックは nn::ngc::ProfanityFilter クラスで行います。事前に FS ライブラリを初期化しておく必要があります。

コード 5-4. nn::ngc::ProfanityFilter クラスの初期化

```
class nn::ngc::ProfanityFilter
{
    ProfanityFilter();
    ProfanityFilter(const nn::WithInitialize&);
    ProfanityFilter(uptr pWorkMemory);

    nn::Result Initialize();
    nn::Result Initialize(uptr pWorkMemory);
}
```

NG ワードのチェックで使用する作業メモリは、ライブラリでメモリブロックから確保させるか、アプリケーションで確保しなければなりません。作業メモリとして必要なサイズは nn::ngc::ProfanityFilter::WORKMEMORY_SIZE (65,536 Byte)です。

nn::WithInitialize() を引数にインスタンスを生成した場合と、引数なしのコンストラクタで生成したインスタンスで引数なしの Initialize() を呼び出した場合は、作業メモリがメモリブロックから確保されます。事前にメモリブロック(nn::os::MemoryBlock クラス)が利用できるようにしなければならないことに注意が必要です。

pWorkMemory に作業メモリの先頭アドレスを指定してインスタンスを生成した場合と、引数なしのコンストラクタで生成したインスタンスで引数ありの Initialize() を呼び出した場合は、作業メモリをアプリケーションで確保しなければなりません。作業メモリは nn::ngc::ProfanityFilter::WORKMEMORY_ALIGNMENT (4 Byte) アライメント、nn::ngc::ProfanityFilter::WORKMEMORY_SIZE (65,536 Byte) 以上の連続したメモリ領域で確保してください。

チェックを完了し、インスタンスが不要になったときは Finalize() を呼び出して終了処理を行ってください。

コード 5-5. nn::ngc::ProfanityFilter クラスの終了

```
class nn::ngc::ProfanityFilter
{
    nn::Result Finalize();
}
```

終了処理後は作業メモリを解放することができます。メモリブロックから確保していた場合は、終了処理の中で解放されています。

5.2.2. NG ワードのチェック

NG ワードのチェックは ProfanityFilter クラスのメンバ関数 CheckProfanityWords() で行います。

コード 5-6. NG ワードのチェック

```
class nn::ngc::ProfanityFilter
{
    nn::Result CheckProfanityWords(
        bit32* pCheckResults, const wchar_t** ppWords, size_t nWordCount);
    nn::Result CheckProfanityWords(
        bit32* pCheckResults, nn::ngc::ProfanityFilterPatternList nPatternCode,
        const wchar_t** ppWords, size_t nWordCount);
    nn::Result CheckProfanityWords(
        bit32* pCheckResults, bool bCommunicateWithOtherRegions,
        const wchar_t** ppWords, size_t nWordCount);
}
```

各オーバーロードは上から、全リージョン・全言語でのチェック、パターンリスト(リージョンと言語の組み合わせ)を指定してチェック、本体のリージョンから適切なパターンリストを自動で判定してチェックするためのものです。

ppWords と *nWordCount* には、チェックする文字列の配列と文字列の個数を指定します。文字列は文字コード UTF16-LE の NULL 終端で指定します。複数の文字列を同時に確認することができ、一度に複数の文字列をチェックすることで個別にチェックするよりも短い時間でチェックすることができます。

pCheckResults には判定結果が格納されますので、*nWordCount* 以上の要素を持つ bit32 型の配列を指定してください。判定結果はビットフラグで返され、フラグが立っている(1 になっている)ビットが、どのパターンリストで NG ワードであると判定されたかを示します。nn::ngc::ProfanityPatternList に定義されている値で 1 を左ビットシフトしたものが、そのパターンリストに対応するビットです。すべてのパターンリストで問題がなければ 0 が格納されます。

nPatternCode には、チェックに使用するパターンリストを nn::ngc::ProfanityPatternList に定義されている値で指定します。nn::ngc::ProfanityPatternList には、以下のパターンリストが定義されています。

表 5-1. パターンリスト

定義	リージョン	言語
PATTERNLIST_JAPAN_JAPANESE	日本	日本語
PATTERNLIST_AMERICA_ENGLISH	米州	北米英語
PATTERNLIST_AMERICA_FRENCH	米州	フランス語(カナダ)
PATTERNLIST_AMERICA_SPANISH	米州	スペイン語(ラテンアメリカ)
PATTERNLIST_EUROPE_ENGLISH	欧州	イギリス英語
PATTERNLIST_EUROPE_FRENCH	欧州	フランス語
PATTERNLIST_EUROPE_GERMAN	欧州	ドイツ語
PATTERNLIST_EUROPE_ITALIAN	欧州	イタリア語
PATTERNLIST_EUROPE_SPANISH	欧州	スペイン語
PATTERNLIST_EUROPE_DUTCH	欧州	オランダ語
PATTERNLIST_KOREA_KOREAN	韓国	韓国語
PATTERNLIST_CHINA_SIMP_CHINESE	中国	中国語(簡体字)
PATTERNLIST_EUROPE_PORTUGUESE	欧州	ポルトガル語

PATTERNLIST_EUROPE_RUSSIAN	欧州	ロシア語
PATTERNLIST_AMERICA_PORTUGUESE	北米	ポルトガル語(ブラジル)
PATTERNLIST_TAIWAN_TRAD_CHINESE	台湾	中国語(繁体字)
PATTERNLIST_TAIWAN_ENGLISH	台湾	英語(台湾)

引数 *bCommunicateWithOtherRegions* を持つオーバーロードは、チェックすべきパターンリストを本体のリージョンから判断します。そのため、このオーバーロードを呼び出す場合は CFG ライブラリの初期化が必要です。この引数に false を指定した場合は本体のリージョンと一致するパターンリストでのみチェックを行います。現在、この引数は参照されず、false を指定した場合のチェックを行います。

NG ワードのチェックは処理をブロックし、完了までに時間がかかる場合がありますので、別のスレッドを作成してチェックすることを推奨します。

5.2.3. 数字のチェック

nn::ngc::ProfanityFilter クラスによるチェックでは、メールアドレスの表記に使われる可能性のあるアットマーク記号が含まれていることは判定できますが、電話番号などの表示に利用される可能性のある数字が多く含まれているかどうかについては判定できません。

文字列に含まれている数字の数をチェックするには、nn::ngc::CountNumbers() を呼び出します。この関数の呼び出しには、nn::ngc::ProfanityFilter クラスの初期化は必要ありません。

コード 5-7. 文字列に含まれる数字の数のチェック

```
int nn::ngc::CountNumbers(const wchar_t *pString);
```

pString には、チェックする文字列を指定します。文字列は文字コード UTF16-LE の NULL 終端で指定します。

返り値は文字列に含まれている数字の数です。エラーが発生したときには負の値が返されます。

5.2.4. 文章のチェック

nn::ngc::ProfanityFilter クラスの MaskProfanityWordsInText() では、文章のように長い文字列に対する NG ワードのチェックを行い、問題となる文字列をアスタリスク(「*」)に置き換えることができます。表示名などのように短い文字列に対する NG ワードのチェックは、CheckProfanityWords() を利用してください。

コード 5-8. 文章内の NG ワードのチェック

```
class nn::ngc::ProfanityFilter
{
    nn::Result MaskProfanityWordsInText(
        int* pProfanityWordCount, wchar_t* pText);
    nn::Result MaskProfanityWordsInText(
        int* pProfanityWordCount,
        nn::ngc::ProfanityFilterPatternList nPatternCode, wchar_t* pText);
    nn::Result MaskProfanityWordsInText(
        int* pProfanityWordCount,
        bool bCommunicateWithOtherRegions, wchar_t* pText);
    void SetMaskMode(bool bOverWrite);
}
```

各オーバーロードは上から、全リージョン・全言語でのチェック、パターンリスト(リージョンと言語の組み合わせ)を指定してチェック、本体のリージョンから適切なパターンリストを自動で判定してチェックするためのものです。

pText にチェック対象となる文章へのポインタを渡します。*pProfanityWordCount* に NULL ではない int 型変数へのポインタを渡すと、問題となる文字列が文章中に現れた回数を返します。

nPatternCode によるパターンリストの指定や、*bCommunicateWithOtherRegions* の指定については、CheckProfanityWords() の説明を参照してください。

NG ワードのチェックは処理をブロックし、完了までに時間がかかる場合がありますので、別のスレッドを作成してチェックすることを推奨します。

SetMaskMode() の *bOverWrite* には、MaskProfanityWordsInText() で問題となる文字列をアスタリスクに置き換える際に、そのままの文字数で置き換える(true:デフォルト)か、文字数に関係なく 1 文字で置き換える(false)かを指定します。そのままの文字数で置き換えた場合、元の文字とアスタリスクの文字幅の違いにより、プロポーショナルフォントで表示したときに枠外などにはみ出てしまう可能性があることに注意してください。

5.3. おしらせ

HOMEメニュー機能の「おしらせリスト」には、Wii の伝言板のように任天堂からのおしらせやアプリケーションからのおしらせなどが表示されます。このおしらせをアプリケーションから投稿することができます。

5.3.1. 初期化と終了

アプリケーションからおしらせを投稿するためには、NEWS ライブラリを利用します。おしらせを投稿する前に、nn::news::Initialize() で NEWS ライブラリの初期化を行ってください。おしらせを投稿する必要がなくなったときは、nn::news::Finalize() を呼び出してください。

コード 5-9. NEWS ライブラリの初期化と終了

```
nn::Result nn::news::Initialize();
nn::Result nn::news::Finalize();
```

5.3.2. おしらせの投稿

おしらせの投稿は nn::news::PostNews() の呼び出しで行われます。

コード 5-10. おしらせの投稿

```
nn::Result nn::news::PostNews(
                    const wchar_t* subject, const wchar_t* message,
                    const u8* picture = NULL, size_t pictureSize = 0,
                    u32 serialId = 0, u32 dataVersion = 0, u64 jumpParam = 0);
```

おしらせのタイトル(*subject*)と本文(*message*)の指定は必須です。どちらも、文字コード UTF16-LE の NULL 終端文字列で指定し、改行には 0x000A(LF)の制御文字を使用します。

picture と *pictureSize* には、おしらせに画像を添付するときに、画像データとそのサイズを指定します。

serialId と *dataVersion* には、おしらせのシリアル ID とデータバージョンを指定しますが、現時点では 0 を指定してください。

jumpParam には、おしらせリストからアプリケーションにジャンプする際のパラメータを指定します。アプリケーションでは、

おしらせリストから起動されたことを nn::news::IsFromNewsList() で検知することができ、*jumpParam* で指定されたパラメータを取得することができます。

それぞれの引数で指定されたタイトル、本文、添付画像は関数内でコピーされますので、投稿が完了した時点で解放することができます。

タイトル、本文、添付画像の仕様は以下のとおりです。投稿間隔の制限などについてはガイドラインを参照してください。

表 5-2. おしらせの仕様

項目	仕様
タイトル	文字コード UTF16-LE。終端文字を含めて、nn::news::SUBJECT_LEN 文字以下。最大幅の内蔵フォント(日米欧リージョンは "%"、そのほかのリージョンはひらがな、漢字、ハングルなど)17 文字分の幅まで表示されます。最大表示幅を越えた場合は、文字サイズが最大 80% まで縮小されます。それでも最大表示幅を越えた場合は、越えた分の文字が非表示になります。
本文	文字コード UTF16-LE。終端文字を含めて、nn::news::MESSAGE_LEN 文字以下。最大幅の内蔵フォント 18 文字分の幅までが 1 行に表示されます。
添付画像	MPO 形式で 3D 写真可。nn::news::PICTURE_SIZE バイト以下。

仕様で定められた文字列長を超えたタイトル、本文を指定したときや、不正な形式の画像やデータサイズの制限を超える添付画像を指定したときは、以下のエラーが返されます。

表 5-3. おしらせの投稿で返されるエラー

返り値	エラーの原因
ResultInvalidSubjectSize	タイトルに指定した文字列が長すぎます。
ResultInvalidMessageSize	本文に指定した文字列が長すぎます。
ResultInvalidPictureSize	添付画像の制限サイズを超えています。
ResultInvalidPicture	対応していない形式の画像データです。

5.3.3. URL 付おしらせの投稿

URL 付おしらせは、通常のおしらせに URL を付与したものです。おしらせに URL が付与されると、おしらせリストで表示したときに、指定された URL のページをインターネットブラウザで開くためのボタンが表示されます。

コード 5-11. URL 付おしらせの投稿で使用する関数

```
nn::Result nn::news::PostNewsUrl(
                    const wchar_t* subject, const wchar_t* message,
                    const u8* url, u8* workBuf,
                    const u8* picture = NULL, size_t pictureSize = 0,
                    u32 serialId = 0, u32 dataVersion = 0, u64 jumpParam = 0);
size_t nn::news::GetWorkBufferSizeForNewsUrl(
                    const wchar_t* message, const u8* url);
```

url と *workBuf* 以外の引数への指定方法は nn:news::PostNews() と同じです。

url には、付与する URL を文字コード UTF-8 で指定します。付与できる URL の最大長は、NULL 終端を含んで

nn::news::MESSAGE_URL_URL_SIZE Byte です。なお、URL は本文が格納される領域を利用するため、URL を付与したときの本文の最大文字数は以下の式で求めた値となります。

nn::news::MESSAGE_LEN -(NULL 終端を含む URL の文字数)/ 2

workBuf には、GetWorkBufferSizeForNewsUrl() で返されるサイズで確保されたワークバッファを指定します。本文や URL の文字数が最大長をオーバーしている、不正なポインタを指定しているなど、渡された引数が不正である場合は 0 が返されます。

5.4. アカウント

アカウントライブラリ(ACT ライブラリ)は、アプリケーションから 3DS のアカウントシステムの機能を利用するためのライブラリです。アカウントシステムの全体像については、CTR-SDK に同梱されている「CTR アカウントシステム デベロッパーズガイド」を参照してください。

アプリケーションは ACT ライブラリを利用することにより、本体内にキャッシュされているニンテンドーネットワークアカウントの情報の取得や、登録された情報に基づく認証を行うことができます。

5.4.1. 初期化と終了

ACT ライブラリの初期化は nn::act::Initialize() で行います。

ネットワーク時計やログインアプレットのように、インターネット通信を行う ACT ライブラリの関数を利用する場合は、それらの関数を呼び出す前に AC ライブラリでインターネット接続を完了させておいてください。また、インターネット通信を行う関数には、ACT ライブラリの初期化時に通信用バッファの指定が必要になるものもあります。

> **補足:** 現在公開されている ACT ライブラリの関数には、通信用バッファの指定が必要なものはありません。

通信用バッファとして渡すバッファは **デバイスメモリ以外から確保** しなければなりません 。また、バッファの先頭アドレスのアライメントは nn::act::BUFFER_ALIGNMENT (4096 Byte) 、バッファのサイズは nn::act::BUFFER_UNITSIZE (4096) の倍数でなければなりません。

nn::act::Initialize() は多重呼び出しが可能です。ただし、通信用バッファには ACT ライブラリが初期化されていない状態でのときに指定されたバッファが使用され、初期化済みの状態で指定されたバッファは使用されません。

nn::act::Initialize() が呼び出された回数は記録されます。そのため、nn::act::Finalize() を nn::act::Initialize() と同じ回数呼び出すまで ACT ライブラリは初期化済みと判断されます。

5.4.2. ニンテンドーネットワークアカウントの情報

ACT ライブラリを使用することで、ニンテンドーネットワークアカウントの情報をアプリケーションで利用することができます。

ニンテンドーネットワークアカウントの情報を取得する関数のほとんどがローカルにキャッシュされたデータを返すため、関数の実行で通信が行われることはありません。

ニンテンドーネットワークアカウントの情報と取得関数の対応は下表のとおりです。詳細については関数リファレンスを参照してください。

表 5-4. ニンテンドーネットワークアカウントの情報と取得関数の対応

情報	関数	備考
ネットワークアカウント	`IsNetworkAccount()`	本体アカウントがネットワークアカウントかどうかを判定します。
アカウント ID	`GetAccountId()`	
プリンシパル ID	`GetPrincipalId()`	ACT ライブラリで取得するプリンシパル ID はニンテンドーネットワークアカウントの登録時にアカウントサーバーで発行された ID です。 同じプリンシパル ID という名前でも、FRIENDS ライブラリで取得するプリンシパル ID はフレンドサーバーで発行された ID であり、ACT ライブラリで取得する ID とは異なります。2 つの ID を混在して管理したり、FRIENDS ライブラリの関数に ACT ライブラリで取得したプリンシパル ID を渡したりしないように注意してください。
パスワード	—	取得することができません。
メールアドレス	—	企画上取得する必要がある場合は、弊社窓口までご相談ください。
生年月日	—	企画上取得する必要がある場合は、弊社窓口までご相談ください。
年齢	`IsOverAge()`	指定年齢以上かの判定のみが可能です。詳しくは「5.4.2.1. 年齢判定」を参照してください。
居住国	`GetCountry()`	ISO 3166-1 alpha-2 形式で返します。
居住地域	`GetSimpleAddressId()` `GetWiiUSimpleAddressId()`	
タイムゾーン	`GetTimeZoneId()`	居住地域のタイムゾーンを tz database 形式で返します。
時差	`GetUtcOffset()`	居住地域の時刻と UTC との時差をマイクロ秒単位で返します。 夏時間を考慮した時差を返しますが、キャッシュのタイミングによっては正確な時差とはならないことがあります。
性別	—	企画上取得する必要がある場合は、弊社窓口までご相談ください。

5.4.2.1. 年齢判定

`nn::act::IsOverAge()` は、ニンテンドーネットワークアカウントのユーザーの年齢が引数 *age* で指定された年齢以上であるかどうかを判定します。

年齢判定には、ニンテンドーネットワークアカウントの生年月日とネットワーク時計が使用されます。本体アカウントがネットワークアカウントではない場合や、ネットワーク時計が有効でない場合、この関数は必ず `false` を返します。ネットワーク時計を有効にする方法については「5.4.3. ネットワーク時計」を参照してください。

5.4.3. ネットワーク時計

ネットワーク時計を利用する関数を呼び出す場合は、事前にネットワーク時計の有効化と補正が行われていないと正しい時刻をもとにした処理が行われません。ネットワーク時計が利用可能になっているかを `nn::act::IsNetworkTimeValidated()` で確認し、利用可能でない場合は

nn::act::InquireNetworkTime() を呼び出してください。

nn::act::InquireNetworkTime() はアカウントサーバーとの通信を行い、処理完了までアプリケーションの実行をブロックすることに注意してください。

なお、アカウントサーバーとの通信が行われる関数の実行が正常に完了したあとは、ネットワーク時計が利用可能な状態になっています。

5.4.4. ログインアプレット

ログインアプレットは、アカウントサーバーとの通信を行い、アカウント認証処理と各種サービストークンの取得処理などをアプリケーションの代わりに行います。なお、一連の処理が完了するまでアプリケーションに処理は戻りません。

ログインアプレット内でアカウント認証が行われますので、本体アカウントがネットワークアカウントでなければエラーが返されます。事前に nn::act::IsNetworkAccount() でネットワークアカウントであるかどうかを確認し、ネットワークアカウントでなければログインアプレットを利用しないようにすることを推奨します。

> **注意:** ログインアプレットではペアレンタルコントロールの設定をチェックしません。ペアレンタルコントロールで制限される機能を提供しているアプリケーションは、ペアレンタルコントロールの設定をアプリケーションで適切に反映してください。たとえば、アカウント認証後に利用するすべてのサービスがペアレンタルコントロールで制限されているならば、ログインアプレットの呼び出し自体を行わないなどです。

> **補足:** ほかのライブラリアプレットと同様に、ログインアプレットもプリロードに対応しています。また、アプリケーションに復帰した直後に HOMEボタンや電源ボタン、ソフトウェアリセットが要因となって復帰したかどうかのチェックと対応が必要です。

5.4.4.1. 独自サービス向けサービストークン

独自サーバーを利用したサービス向けに、接続するユーザーが任天堂のアカウント認証を受けたユーザーであることを証明するサービストークンを nn::act::applet::AcquireIndependentServiceToken() で取得することができます。

コード 5-12. 独自サービス向けサービストークンを取得する関数

```
nn::Result nn::act::applet::AcquireIndependentServiceToken(
    char*           pServiceToken,
    const char*     pClientId,
    u32             reusableRangeInSeconds = 0
);
```

引数 *reusableRangeInSeconds* に 0 以外の値を渡した場合、以前にアカウントサーバーと通信を行ってサービストークンを取得してから指定された秒数が経過していなければ、通信を行わずにキャッシュされたサービストークンを返します。期限切れや独自サーバー側の拒否など、サービストークンが無効になったときには、この引数に 0 を指定し、確実にサービストークンの再取得を行ってください。

ログインアプレット内で通信を行いますが、ACT ライブラリの初期化時に通信用バッファを指定する必要はありません。

コード 5-13. 独自サービス向けサービストークンを取得する際のエラーハンドリングの例

```
char serviceToken[NN_ACT_INDEPENDENT_SERVICE_TOKEN_SIZE];
nn::Result result = nn::act::applet::AcquireIndependentServiceToken(
    serviceToken, CLIENT_ID_INDEPENDENT, SERVICE_TOKEN_REUSABLE_RANGE );

if ( result.IsSuccess() )
{
    // serviceTokenにサービストークンが格納されています
}
else if ( nn::act::ResultCanceled::Includes( result ) )
{
    // ユーザーによって明示的にキャンセルされたので終了シーケンスに進む
}
else if ( nn::act::ResultApplicationUpdateRequired::Includes( result ) )
{
    // アプリケーションの更新が必要なのでニンテンドーeショップのパッチページにジャンプする(任意)
}
else
{
    // キャンセル以外のエラーが発生したため、エラーコードを表示する
    u32 networkErrorCode = nn::act::GetErrorCode( result );

    // 設定構造体初期化
    nn::applet::AppletWakeupState wstate;
    nn::erreula::Parameter ere_param;     // エラーEULAの設定構造体
    ere_param.config.errorType = nn::erreula::ERROR_TYPE_ERROR_CODE;
    ere_param.config.errorCode = networkErrorCode;
    ere_param.config.upperScreenFlag = nn::erreula::UPPER_SCREEN_NORMAL;
    ere_param.config.homeButton = true;
    ere_param.config.softwareReset = false;
    ere_param.config.appJump = false;

    nn::erreula::StartErrEulaApplet( &wstate, &ere_param );
}
```

5.4.4.2. Miiverse 向けサービストークン

Miiverse 向けのサービストークンを nn::act::applet::AcquireOlvServiceToken() で取得することができます。Miiverse 向けサービストークンは OLV ライブラリを利用する際に必要となります。

ログインアプレット内で通信を行いますが、ACT ライブラリの初期化時に通信用バッファを指定する必要はありません。

ハンドリングする必要のあるエラーは「5.4.4.1. 独自サービス向けサービストークン」と同じです。

5.4.5. UUID の生成

自動生成されたオブジェクトに割り振るなど、一意に特定可能な識別子として利用されている UUID(Universally Unique Identifier)を nn::act::GenerateUuid() で生成することができます。

生成される UUID は RFC 4122 - Version 1 の仕様に準拠していますが、今後、ユニーク性が損なわれない範囲でフォーマットが変更される可能性があります。そのため、nn::act::GenerateUuid() で返される UUID は返された値のまま使用してください。また、UUID の一部分だけを使用したり、時刻などの情報を取り出して使用したりしないでください。

6. デバッグ用ライブラリ

> **注意:** これらのライブラリはノーサポート扱いであり、基本的に製品に含めることができません。製品に利用する場合は弊社窓口までご相談ください。

デバッグ用途に限定して、無線通信モジュールを利用したソケット通信などのネットワーク接続に関するライブラリを用意しています。

表 6-1. デバッグ用ライブラリ

機能	ライブラリ名	名前空間	説明
ソケット通信	SOCKET	nn::socket	ソケットを通じてリモートホストとのデータ送受信などを実現するライブラリです。
SSL 通信	SSL	nn::ssl	SSL による秘匿通信を補助するライブラリです。
HTTP 通信	HTTP	nn::http	HTTP による通信を行うライブラリです。

> **補足:** HTTP ライブラリは内部でソケット通信を行いますが、アプリケーションによる socket API の操作が HTTP ライブラリのソケット通信に影響を与えることはありません。
>
> たとえば、HTTP ライブラリを使用している状態で nn::socket::Initialize() や nn::socket::Finalize() を呼び出しても、HTTP ライブラリの処理に影響はありません。

6.1. ソケット通信

SOCKET(ソケット通信)ライブラリは、アドレスおよびポート番号と結び付けられたソケットを介してリモートホストとの通信を行うライブラリです。ソケット通信は、ローカル(マシン)とリモートホストとの間のデータの送受信やリモートホストの接続待ち受け、ローカルからリモートホストへの接続といった低位の通信機能を提供します。

6.1.1. 初期化

SOCKET ライブラリの初期化は nn::socket::Initialize() の呼び出しで行われます。

コード 6-1. SOCKET ライブラリの初期化

```
size_t nn::socket::GetRequiredMemorySize(size_t bufferSizeForSockets,
                                         s32 maxSessions);
nn::Result nn::socket::Initialize(uptr bufferAddress, size_t bufferSize,
                                  s32 bufferSizeForSockets, s32 maxSessions);
```

bufferAddress と *bufferSize* には、ライブラリが使用するワークメモリの先頭アドレスとそのサイズを指定します。ワークメモリは先頭アドレスのアライメントが nn::socket::BUFFER_ALIGNMENT (4096 Byte) で、**デバイスメモリ以外から確保** しなければなりません。ワークメモリに必要なサイズは、ソケット全体で確保する送受信バッファのサイズ(*bufferSizeForSockets*)と最大セッション数(*maxSessions*)を引数に、nn::socket::GetRequiredMemorySize() を呼び出して取得することができます。このとき指定する送受信バッファのサイズ(*bufferSizeForSockets*)は nn::socket::BUFFER_UNITSIZE_FOR_SOCKETS (4096) の倍数でなければなりません。

ソケットごとに割り当てる送受信バッファは nn::socket::SetSockOpt() で指定することができます。デフォルトでは 1 ソケットあたりの送受信バッファに、TCP の場合は 16 KByte(送信 8 KByte、受信 8 KByte)、UDP の場合は 32 KByte が割り当てられます。*bufferSizeForSockets* には、最大ソケット数が 1 であっても、最低でも 64 KByte 程度は確保してください。

maxSessions にはソケットを使用するスレッドの数(最大セッション数)を指定します。処理がブロックされるライブラリ関数の呼び出しを指定された数までのスレッドから受け付けることができます。厳密には、ブロックするライブラリ関数を同時に呼び出さなければ指定数以上のスレッドからライブラリ関数を呼び出すことができますが、そのような運用は推奨しません。ライブラリ関数の呼び出し中に最大セッション数を超えた場合、非同期呼び出し、同期呼び出しに関わらず、セッションが空くまで処理がブロックされます。また、データの着信や送信完了などでブロックが解除される条件を満たしていても、セッションが空くまで処理はブロックされます。非同期処理のはずが、同期処理と同じような挙動になる恐れがありますので注意してください。nn::socket::Poll() を利用すれば、ソケットの状態を一回の呼び出しで確認することができますので、必要なセッションの数を減らすことができます。

> **注意:** デフォルトで割り当てられる送受信バッファのサイズは、今後変更になる可能性があります。

6.1.2. ソケットの作成

リモートホストもそこに接続するローカル(マシン)も、まずソケットを作成しなければなりません。ソケットの作成は nn::socket::Socket() の呼び出しで行われます。アプリケーションで同時に使用することのできるソケットの数は 16 に制限されています。

> **補足:** 同時使用可能ソケット数は、今後変更になる可能性があります。

コード 6-2. ソケットの作成

```
s32 nn::socket::Socket(s32 af, s32 type, s32 protocol);
```

af に指定することができるのは PF_INET のみです。

type には作成するソケットの種類を SOCK_STREAM(ストリーム)または SOCK_DGRAM(データグラム)から指定します。ストリームソケットは双方のソケットから接続を確立させる必要がありますが、データグラムソケットは一方的な送信と受信をデータブロック単位で行います。また、前者はデータブロックの到達順が保証されていますが、後者は保証されていない代わりに通信速度が高速になるという利点があります。

protocol にはソケットに用いるプロトコルを指定しますが、現時点では 0 を指定してください。*protocol* が 0 の場合、*af* と *type* で指定したプロトコルファミリとタイプに対するデフォルトのプロトコルが使われます。デフォルトのプロトコルは、ストリームソケットならば TCP、データグラムソケットならば UDP です。

返り値に 1 以上が返された場合、返り値はソケットを判別するためのソケット記述子です。0 以下が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-2. ソケット作成時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EAFNOSUPPORT	指定されたプロトコルファミリはサポートしていません。

EPROTONOSUPPORT	指定されたプロトコルはサポートされていません。
EMFILE	ソケット記述子をこれ以上作成することはできません。
ENOMEM	メモリ不足のため、ワークメモリを確保することができません。
EPROTOTYPE	指定されたソケットの種類はサポートしていません。
EINVAL	不正な呼び出しです。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.3. アドレスとポート番号の結び付け

ソケットは、どのアドレスでどのポート番号であるかを示すソケットアドレスが結び付けられて(バインドされて)いなければ通信を行うことができません。作成されただけのソケットには、まだソケットアドレスが結び付けられていません。ソケットアドレスをソケットに結び付けるには nn::socket::Bind() を呼び出します。

コード 6-3. アドレスおよびポート番号の結び付け

```
s32 nn::socket::Bind(s32 s, const nn::socket::SockAddrIn* sockAddr);
```

s には作成したソケットのソケット記述子を指定します。すでにソケットアドレスが結び付けられているソケットを指定した場合はエラーとなります。

sockAddr にはソケットアドレス(アドレスファミリ、ポート番号、アドレス)を設定した nn::socket::SockAddrIn 構造体へのポインタを渡します。アドレスの指定は IPv4 のみ対応しています。

処理に成功した場合は返り値に 0 が返されます。0 以外が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-3. アドレス、ポート番号の結び付け時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
EINVAL	無効な処理(ソケットがすでにバインド済みなど)です。
EAFNOSUPPORT	指定されたアドレスファミリはサポートしていません。
EADDRINUSE	指定されたソケットアドレスはすでに使用されています。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.4. 動作モード

ソケットの動作モードを nn::socket::Fcntl() の呼び出しで取得・設定することができます。

コード 6-4. 動作モードの設定・取得

```
s32 nn::socket::Fcntl(s32 s, s32 cmd, s32 val);
```

s には動作モードの設定・取得を行うソケットのソケット記述子を指定します。

cmd には動作モードを設定する(F_SETFL)のか、取得する(F_GETFL)のかを指定します。

val には 0 または動作モードに設定するフラグの論理和を指定します。*cmd* に F_GETFL を渡している場合、この引数は無視されます。

設定可能なフラグは O_NONBLOCK(非封鎖モード)だけです。このフラグが設定されていないソケットの動作モードは封鎖モードとなります。ソケット作成時は封鎖モードに設定されています。

cmd に F_GETFL を渡している場合の返り値は、設定されている動作モードを示すフラグの論理和です。*cmd* に F_SETFL を渡している場合、返り値に 0 が返されたときは処理に成功しています。どちらの場合も、負の値が返されたときはエラーです。下表は発生するエラーの一覧です。

表 6-4. 動作モードの設定・取得時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
EINVAL	無効な処理(*cmd* の指定が不正)です。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.5. 接続の待ち受け

データグラムソケットはソケットアドレスのバインドが完了してすぐにデータの送受信を行うことができますが、ストリームソケットはデータの送受信を行う前に接続を確立していなければなりません。

サーバー側となるストリームソケットは、nn::socket::Listen() でクライアント側となるストリームソケットからの接続要求を待ち受けるキューを作成し、nn::socket::Accept() で着信(接続要求)を受け付けます。

コード 6-5. 接続の待ち受け

```
s32 nn::socket::Listen(s32 s, s32 backlog);
s32 nn::socket::Accept(s32 s, nn::socket::SockAddrIn* sockAddr);
```

どちらの関数でも、*s* には待ち受けに使用するソケットを指定します。このソケットはストリームソケットとして作成され、ソケットアドレスがバインドされていなければなりません。

backlog にはソケットのリッスンバックログとして使用するキューの最大数を指定します。0 または負の値を指定した場合は 1 を指定したものとみなされます。

sockAddr には受け付けた着信のソケットアドレスを格納する nn::socket::SockAddrIn 構造体へのポインタを指定します。

nn::socket::Listen() の返り値に 0 が返された場合は処理に成功しています。負の値が返された場合はエラーです。

nn::socket::Accept() は接続待ちのソケットがキューに存在しない場合、待ち受けに使用しているソケットが非封鎖モードでなければ処理をブロックします。返り値に 1 以上が返された場合、返り値は待ち受けに使用しているソケットと同じアドレスで新たに作成されたソケットのソケット記述子です。0 以下が返された場合はエラーです。

下表は発生するエラーの一覧です。

表 6-5. 接続の待ち受け時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
EOPNOTSUPP	サポートされていない処理です。
EINVAL	無効な処理です。
ENOBUFS	リソースが不足しています。
EAGAIN	接続待ちのソケットがありません。(非封鎖モード)
ECONNABORTED	接続がキャンセルされました。
EMFILE	ソケット記述子をこれ以上作成することはできません。
ENOMEM	リッスンバックログ作成のためのメモリが不足しています。
EWOULDBLOCK	EAGAIN と同じです。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.6. リモートホストへの接続

クライアント側となるストリームソケットはサーバー側のストリームソケットに nn::socket::Connect() で接続を試み、接続が確立されるまでデータの送受信を行うことはできません。データグラムソケットが nn::socket::Connect() を呼び出した場合は、送信先ソケットアドレスの書き換えだけが行われます。

コード 6-6. リモートホストへの接続

```
s32 nn::socket::Connect(s32 s, const nn::socket::SockAddrIn* sockAddr);
```

s には接続に使用するソケットのソケット記述子を指定します。ソケットにソケットアドレスがまだ結び付けられていない場合は、関数内で未使用のローカルソケットアドレスに結び付けられます。

sockAddr には接続先のソケットアドレスを設定した nn::socket::SockAddrIn 構造体へのポインタを指定します。

処理に成功した場合は返り値に 0 が返されます。0 以外が返された場合はエラーです。発生するエラーの一覧は以下のようになっています。

表 6-6. リモートホストへの接続時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EAFNOSUPPORT	指定されたプロトコルファミリはサポートしていません。
EBADF	指定されたソケット記述子が正しくありません。
EALREADY	既に非封鎖モードで接続を試みています。
ECONNREFUSED	接続する相手と同時オープンになった後、リセットされました。
ECONNRESET	接続がリセットされました。
EINPROGRESS	指定されたソケットは現在状態を変更しています。
EINVAL	無効な処理(*sockAddr* の指定が不正)です。
EISCONN	すでに使用されているソケットのため、処理に利用できません。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
ENETUNREACH	接続先が見つかりません。
ENOBUFS	空いているソケットへのソケットアドレスの一時割り当てに失敗しました。
ETIMEDOUT	接続先の応答がタイムアウトしました。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

s に指定したソケットがデータグラムソケットの場合、ソケットの送信先ソケットアドレスが *sockAddr* に設定されているソケットアドレスに書き換えられるだけですので、動作モードによる処理の違いはありません。

s に指定したソケットがストリームソケットの場合、動作モードが封鎖モードならば接続が確立するまで処理がブロックされます。非封鎖モードならばすぐに制御が戻され、接続が確立しているかどうかは nn::socket::Poll() で確認することができます。

コード 6-7. リモートホストへの接続の確認

```
struct nn::socket::PollFd
{
    s32         fd;
    s32         events;
    s32         revents;
};
s32 nn::socket::Poll(nn::socket::PollFd fds[], u32 nfds, s32 timeout);
```

nn::socket::Poll() は複数のソケット記述子から、送受信が可能な状態のソケットがあるかどうかを調査します。

fds には調査対象のソケット記述子と調査条件を設定した nn::socket::PollFd 構造体の配列を指定します。構造体のメンバ変数 fd には調査対象のソケット記述子を、events には調査条件を設定します。revents には調査結果が格納されますので 0 を設定してください。

nn::socket::Connect() で接続が確立できなかった場合、revents には POLLRDNORM と POLLWRNORM のフラグが立てられ、そのあとのデータの送受信時にエラーが返されます。下表は、events に設定する調査条件と revents に格納される調査結果のフラグの一覧です。

表 6-7. 調査条件と調査結果に設定されるフラグ

フラグ	説明
POLLRDNORM	受信可能状態を示すフラグです。
POLLRDBAND	受信可能状態(優先データ)を示すフラグです。
POLLPRI	受信可能状態(最優先データ)を示すフラグです。
POLLWRNORM	送信可能状態を示すフラグです。
POLLWRBAND	送信可能状態(優先データ)を示すフラグです。
POLLERR	エラーが発生した状態を示すフラグです。調査結果でのみ設定されます。
POLLHUP	ソケットが切断された状態を示すフラグです。調査結果でのみ設定されます。
POLLNVAL	不正なソケット記述子が指定された状態を示すフラグです。調査結果でのみ設定されます。
POLLIN	受信可能状態を示すフラグです。POLLRDNORM と POLLRDBAND の論理和です。
POLLOUT	送信可能状態を示すフラグです。POLLWRNORM と同じです。

nfds には *fds* で指定した配列の要素数を指定します。

timeout には条件に一致するソケットが見つからなかったときのタイムアウト時間をミリ秒単位で指定します。0 以上の値または INFTIM(タイムアウトなし)で指定してください。

nn::socket::Poll() を呼び出すと、条件に一致するソケットが見つかるまでブロックは解除されません。ただし、ソケットが見つからないまま指定されたタイムアウト時間が経過するか、ソケットが切断されたりソケットに異常が発生したりしたときにはブロックが解除されます。

返り値に 1 以上が返された場合、返された値は条件に一致したソケットの数です。処理がタイムアウトした場合は 0 が返されます。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-8. リモートホストへの接続の確認時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
EINVAL	無効な処理です。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.7. データの受信

データの受信は、nn::socket::RecvFrom()、nn::socket::Recv()、nn::socket::Read() のいずれかの関数で行うことができます。ストリームソケットの場合、サーバー側とクライアント側で接続が確立していなければデータの受信を行うことができません。

コード 6-8. データの受信

```
s32 nn::socket::RecvFrom(s32 s, void* buf, s32 len, s32 flags,
                         nn::socket::SockAddrIn* sockFrom);
s32 nn::socket::Recv(s32 s, void* buf, s32 len, s32 flags);
s32 nn::socket::Read(s32 s, void* buf, s32 len);
s32 nn::socket::SockAtMark(s32 s);
```

s にはローカルソケットのソケット記述子を指定します。

buf には受信データを格納するバッファへのポインタを、*len* にはバッファのバイトサイズをそれぞれ指定します。

flags には受信時に行う特殊な動作を指示するフラグを指定します。MSG_DONTWAIT を指定した場合、ソケットの動作モードが封鎖モードであってもブロック状態にはなりません。MSG_PEEK を指定した場合、データの受信は行われますが状態は変化せず、再度同じデータを受信することができます。MSG_OOB を指定した場合、帯域外のデータを受信することができます。受信可能なデータに帯域外のデータが含まれているかは nn::socket::SockAtMark() の返り値が 1 であるかどうかで判断することができ、この関数は TCP プロトコルで緊急データの最後の 1 バイトが受信可能データの先頭にあるかどうかの判断にも使用することができます。

sockFrom に nn::socket::SockAddrIn 構造体へのポインタを指定した場合、データの送信元ソケットアドレスが格納されます。

ストリームソケットで受信している場合、動作モードが封鎖モードならばデータを受信するまで処理がブロックされます。動作モードが非封鎖モードもしくは *flags* に MSG_DONTWAIT を指定しているときは、関数の呼び出し時に受信可能なデータのみをバッファに格納するため、処理がブロックされません。

データグラムソケットで受信している場合、受信データのすべてを 1 回の呼び出しで受信しなければなりません。受信データがバッファ以上のサイズである場合、*flags* に MSG_PEEK が指定されていなければ、バッファを超過したデータが破棄されてしまいます。1 回の呼び出しで受信できるデータの最大サイズは 1500 バイトです。

返り値に 1 以上が返された場合、返された値は受信したデータのバイト数です。ストリームソケットで返り値に 0 が返された場合、リモートホストからのデータ送信が終了したことを示します。データグラムソケットでは 0 が返されることはありません。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-9. データの受信時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
EAGAIN	受信待ちのデータがありません。(非封鎖モードまたは *flags* に MSG_DONTWAIT)
EINVAL	無効な処理です。
EOPNOTSUPP	サポートされていない処理です。
ENOTCONN	指定されたソケットはリモートホストに接続されていません。
ECONNRESET	接続がリセットされました。
EINTR	処理が中止されました。
ETIMEDOUT	処理がタイムアウトしました。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
EWOULDBLOCK	EAGAIN と同じです。

EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.8. データの送信

データの送信は、nn::socket::SendTo()、nn::socket::Send()、nn::socket::Write() のいずれかの関数で行うことができます。ストリームソケットの場合、サーバー側とクライアント側で接続が確立していなければデータの送信を行うことができません。

コード 6-9. データの送信

```
s32 nn::socket::SendTo(s32 s, const void* buf, s32 len, s32 flags,
                       const nn::socket::SockAddrIn* sockTo);
s32 nn::socket::Send(s32 s, const void* buf, s32 len, s32 flags);
s32 nn::socket::Write(s32 s, const void* buf, s32 len);
```

s にはローカルソケットのソケット記述子を指定します。

buf には送信データが格納されているバッファへのポインタを、*len* にはバッファのバイトサイズをそれぞれ指定します。

flags には送信時に行う特殊な動作を指示するフラグを指定します。

sockTo に nn::socket::SockAddrIn 構造体へのポインタを指定した場合、構造体に設定されているソケットアドレスにデータを送信します。

データの送信が完了するまで処理がブロックされます。

データグラムソケットで送信している場合、1 回の呼び出しで送信できるデータの最大サイズは 1500 バイトです。

処理に成功した場合は返り値に 0 が返されます。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-10. データの送信時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EAFNOSUPPORT	指定されたプロトコルファミリはサポートしていません。
EBADF	指定されたソケット記述子が正しくありません。
EAGAIN	封鎖モードの場合、すでに送信を試みましたがブロックされています。 非封鎖モードのストリームソケットの場合、帯域外に送信しようとするデータを内部送信バッファに送信予約できません。
ECONNRESET	接続がリセットされました。
EINTR	処理が中止されました。
EINVAL	無効な処理です。
EOPNOTSUPP	サポートされていない処理です。
EDESTADDRREQ	ストリームソケットの場合、ソケットは接続要求を受け付けるために使用されています。 データグラムソケットの場合、データを送信しようとする通信先のソケットアドレスが決まっていません。
EMSGSIZE	データが内部送信バッファのサイズを超えています。(データグラムソケットのみ)

ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
ENETUNREACH	接続先が見つかりません。
ENOBUFS	空いているソケットへのソケットアドレスの一時割り当てに失敗しました。または、一時送信バッファの確保に失敗しました。
ENOTCONN	指定されたソケットはリモートホストに接続されていません。
ETIMEDOUT	処理がタイムアウトしました。
EWOULDBLOCK	EAGAIN と同じです。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.9. オプション設定

nn::socket::SetSockOpt() と nn::socket::GetSockOpt() の呼び出しで、ソケットの内部設定値や内部状態情報など、オプション設定の変更と取得を行うことができます。

コード 6-10. オプション設定

```
s32 nn::socket::SetSockOpt(s32 s, s32 level, s32 optname, const void* optval,
                           s32 optlen);
s32 nn::socket::GetSockOpt(s32 s, s32 level, int optname, void* optval,
                           int* optlen)
```

引数に指定可能な値についてはヘッダファイル(nn/socket/socket_User.autogen.h)を参照してください。

処理に成功した場合は返り値に 0 が返されます。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-11. オプション設定時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
EINVAL	無効な処理です。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
ENOMEM	メモリ不足のため、ワークメモリを確保することができません。
ENOPROTOOPT	サポートされていないオプションです。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.10. ソケットの切断

nn::socket::Shutdown() の呼び出しで、ソケット通信を遮断してソケットを切断することができます。

コード 6-11. ソケットの切断

```
s32 nn::socket::Shutdown(s32 s, s32 how);
```

s にはソケット通信を遮断するソケットのソケット記述子を指定します。

how には遮断方法を指定します。以降のデータの受信を遮断する場合は SHUT_RD を、送信を遮断する場合は SHUT_WR を、送受信ともに遮断する場合は SHUT_RDWR を指定してください。

処理に成功した場合は返り値に 0 が返されます。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-12. ソケットの切断時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
ENOTCONN	指定されたソケットはリモートホストに接続されていません。
EINVAL	無効な処理です。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.11. ソケットの破棄

同時に使用することのできるソケットの数は制限されています。不要になったソケットは nn::socket::Close() を呼び出して破棄してください。

コード 6-12. ソケットの破棄

```
s32 nn::socket::Close(s32 s);
```

s には破棄する(閉じる)ソケットのソケット記述子を指定します。

ソケットが閉じられると、そのソケットは以後使用できなくなります。閉じられるソケットを使用して呼び出された関数の処理がブロックされている場合、それぞれの関数のブロックが解除されてエラーが返されます。非封鎖モードに設定されていないストリームソケットが閉じられる場合、Linger オプションの設定に従って接続が閉じられます。

デフォルトでは、この関数はブロックせずに処理をすぐに返します。そして、バックグランドで残りの送信データを自動的に転送し、そのあとでソケットが使用していたリソースを解放します。

処理に成功した場合は返り値に 0 が返されます。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-13. ソケットの破棄時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
ENOTCONN	指定されたソケットはリモートホストに接続されていません。

EINVAL	無効な処理です。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

6.1.12. 終了

SOCKET ライブラリの使用を終了するときは nn::socket::Finalize() を呼び出して終了処理を行ってください。

コード 6-13. SOCKET ライブラリの終了

```
nn::Result nn::socket::Finalize(void);
```

ライブラリが使用していたリソースはすべて解放され、使用中のソケット記述子もすべて破棄されます。

6.1.13. ユーティリティ関数

SOCKET ライブラリには、ソケットアドレスの取得や、ネットワークアダプタ情報の取得、DNS を介したリモートホスト情報の取得、数値と文字列間でのアドレスの相互変換、数値のホストバイトオーダー変換とネットワークバイトオーダー変換などのユーティリティ関数が用意されています。

コード 6-14. ソケットアドレスの取得

```
s32 nn::socket::GetSockName(s32 s, nn::socket::SockAddrIn* sockAddr);
s32 nn::socket::GetPeerName(s32 s, nn::socket::SockAddrIn* sockAddr);
```

nn::socket::GetSockName() は *s* に指定されたソケット記述子で示されるソケットのローカルソケットアドレスを *sockAddr* に取得します。ローカルソケットアドレスは通信元のソケットアドレスです。nn::socket::Bind() や nn::socket::Connect() でローカルソケットアドレスを確定していない状態で、UDP プロトコルのソケット(データグラムソケット)に対してこの関数を呼び出した場合は 0.0.0.0 のアドレスが返され、TCP プロトコルのソケット(ストリームソケット)に対してこの関数を呼び出すとエラーが返されます。

nn::socket::GetPeerName() は *s* に指定されたソケット記述子で示されるソケットのリモートソケットアドレスを *sockAddr* に取得します。リモートソケットアドレスは通信先のソケットアドレスです。

どちらの関数も処理に成功した場合は返り値に 0 が返されます。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-14. ソケットアドレスの取得時に発生するエラー

返り値	エラーの詳細
ENETRESET	ライブラリが初期化されていません。
EBADF	指定されたソケット記述子が正しくありません。
ENOTCONN	リモートホストに接続されていないか通信先がいません。(GetPeerName() のみ)
EINVAL	無効な処理です。
ENETDOWN	ローカルのネットワークインタフェースがダウンしています。
EBUSY	内部処理がビジー状態です。Initialize() で指定するセッションが不足している可能性があります。

コード 6-15. ネットワークアダプタ情報の取得

```
u32 nn::socket::GetHostId(void);
```

nn::socket::GetHostId() はローカルホストの IP アドレス(IPv4)を返します。0 が返されたときはネットワークを使用することができない状態です。返り値は IP アドレスを 32 ビット数値(ネットワークバイトオーダー)にしたものです。

コード 6-16. DNS を介したリモートホスト情報の取得

```
nn::socket::HostEnt* nn::socket::GetHostByName(const char8* name);
nn::socket::HostEnt* nn::socket::GetHostByAddr(const void* addr, s32 len,
                                               s32 type);
s32 nn::socket::GetAddrInfo(const char8* nodeName, const char8* servName,
                const nn::socket::AddrInfo* hints, nn::socket::AddrInfo** res);
void nn::socket::FreeAddrInfo(nn::socket::AddrInfo* head);
s32 nn::socket::GetNameInfo(const void* sa, char8* node, s32 nodeLen,
                            char8* service, s32 serviceLen, s32 flags);
```

nn::socket::GetHostByName() と nn::socket::GetHostByAddr() はどちらも指定されたホストの情報を取得します。前者はホスト名またはドット十進記法のアドレスの文字列から、後者はアドレスからホストを検索します。検索中は処理がブロックされ、DNS サーバーへの問い合わせを行う場合があります。返り値が示す構造体の実体はライブラリ内部のバッファですので、これらの関数はスレッドセーフではありません。

nn::socket::GetAddrInfo() はホスト名とサービス名で検索したホストの情報を取得します。検索動作の設定は *hints* に指定された nn::socket::AddrInfo 構造体のメンバ変数 flags で行います。flags には nn::socket::AddrInfoFlag 列挙子に定義されているフラグの論理和を設定します。

表 6-15. nn::socket::AddrInfoFlag 列挙子

フラグ	説明
AI_PASSIVE	*nodeName* に NULL を指定した場合、0.0.0.0 を使用します。
AI_CANONNAME	*nodeName* に対応する正規名を取得します。
AI_NUMERICHOST	*nodeName* にドット十進記法のアドレスを指定している。
AI_NUMERICSERV	*servName* にポート番号を文字列で指定している。

検索中は処理がブロックされ、DNS サーバーへの問い合わせを行う場合があります。*res* に返される検索結果の構造体は nn::socket::Initialize() で指定したアロケータからライブラリが確保したメモリに格納されます。検索結果が不要になったときは nn::socket::FreeAddrInfo() でライブラリが確保したメモリを解放してください。

nn::socket::GetAddrInfo() は処理に成功した場合、返り値に 0 を返します。負の値を返した場合はエラーです。下表は発生するエラーの一覧です。

表 6-16. nn::socket::GetAddrInfo() によるホスト情報の取得時に発生するエラー

返り値	エラーの詳細
EAI_BADFLAGS	*hints* に AI_CANONNAME が指定されているにも関わらず *nodeName* が NULL または空文字列です。
EAI_FAIL	修復不可能なエラーが発生しました。
EAI_FAMILY	*hints* に指定したアドレスファミリがサポートされていません。

EAI_MEMORY	メモリの確保に失敗しました。
EAI_NONAME	検索対象のホストが見つかりません。
EAI_SOCKTYPE	*hints* に指定したソケットの種類、ソケットのプロトコル、またはそれらの組み合わせがサポートされていません。
EAI_SYSTEM	システムエラーが発生しました。Initialize を呼び出さずに本関数を呼び出した場合などに発生します。

nn::socket::GetNameInfo() はアドレスからホスト名とサービス名を検索します。検索動作の設定は *flags* に nn::socket::NameInfoFlag 列挙子に定義されているフラグの論理和を指定することで行います。

表 6-17. nn::socket::NameInfoFlag 列挙子

フラグ	説明
NI_NOFQDN	完全修飾ドメイン名（FQDN）のうちノード名の部分だけをホスト名として取得します。
NI_NUMERICHOST	*node* にホストアドレスをドット十進記法による文字列に変換したものを格納します。
NI_NAMEREQD	検索対象のホストが見つからない場合に、数値形式のホストアドレスを文字列に変換することで代替せずにエラー（EAI_NONAME）として扱います。
NI_NUMERICSERV	*service* にサービス（ポート番号）を数値記法による文字列に変換したものを格納します。

検索中は処理がブロックされ、DNS サーバーへの問い合わせを行う場合があります。

処理に成功した場合は返り値に 0 が返されます。負の値が返された場合はエラーです。下表は発生するエラーの一覧です。

表 6-18. nn::socket::GetNameInfo() によるホスト情報の取得時に発生するエラー

返り値	エラーの詳細
EAI_FAMILY	*sa* に指定したソケットアドレスに含まれるプロトコルファミリはサポートされていません。
EAI_NONAME	*nodeLen* または *serviceLen* に指定したバッファの大きさが不正です。 または、*flags* に NI_NAMEREQD を指定した呼び出しで検索対象のホストが見つかりません。
EAI_SYSTEM	システムエラーが発生しました。Initialize を呼び出さずに本関数を呼び出した場合などに発生します。

コード 6-17. アドレス変換

```
s32 nn::socket::InetAtoN(const char* cp, nn::socket::InAddr* inp);
char* nn::socket::InetNtoA(nn::socket::InAddr in);
s32 nn::socket::InetPtoN(int af, const char* src, void* dst);
const char* nn::socket::InetNtoP(int af, const void* src, char* dst,
                                 unsigned len);
```

nn::socket::InetAtoN() と nn::socket::InetNtoA() を呼び出すことで、アドレスを十進記法の文字列と nn::socket::InAddr 構造体の数値形式間で相互変換することができます。変換することのできるアドレスは IPv4 のホストアドレスで、本来ならば nn::socket::InetPtoN() と nn::socket::InetNtoP() はアドレスファミリを指定することができますが、CTR-SDK では AF_INET のみが指定可能です。

これらの関数は nn::socket::Initialize() を呼び出していなくても使用することができます。

コード 6-18. バイトオーダー変換

```
bit32 nn::socket::HtoNl(bit32 v);
bit32 nn::socket::NtoHl(bit32 v);
bit16 nn::socket::HtoNs(bit16 v);
bit16 nn::socket::NtoHs(bit16 v);
```

数値を、ホストバイトオーダーからネットワークバイトオーダーへ(HtoN)またはネットワークバイトオーダーからホストバイトオーダーへ(NtoH)変換します。関数名の末尾が "l" ならば 32 ビット数値、"s" ならば 16 ビット数値のバイトオーダー変換を行うことができます。

6.2. SSL 通信

SSL(SSL 通信)ライブラリは、ソケット通信をラッピングして通信の秘匿化を補助します。

6.2.1. 初期化

SSL ライブラリの初期化は nn::ssl::Initialize() の呼び出しで行われます。

コード 6-19. SSL ライブラリの初期化

```
nn::Result nn::ssl::Initialize(void);
```

6.2.2. 接続クラスの生成

SSL 通信は接続クラス(nn::ssl::Connection)にラッピングされたソケットを介して行われます。

コード 6-20. 接続クラスのコンストラクタとソケットのアサイン

```
class nn::ssl::Connection
{
    explicit Connection(s32 socket);
    explicit Connection();
    bool AssignSocket(s32 socket);
}
```

コンストラクタにはソケット記述子を引数に持つものと、引数なしのものがあります。引数なしで生成された接続クラスは、メンバ関数の AssignSocket() でソケットを結び付ける必要があります。

6.2.3. 通信先の設定

メンバ関数の Initialize() で、通信先のサーバーを接続クラスに設定します。

コード 6-21. 通信先の設定

```
class nn::ssl::Connection
{
    nn::Result Initialize(const char* pServerName,
                          u32 verifyOpt = VERIFY_NONE);
}
```

pServerName には通信先サーバーのホスト名を指定します。SSL 通信時には、このホスト名とサーバー証明書の

CommonName もしくは subjectAltName 拡張領域の dnsName/ipAddress を比較し、一致していなかった場合は接続エラーを返します。ホスト名として指定できる文字列の長さは、終端文字を含めて nn::socket::MAXDNAME バイトまでです。

verifyOpt には SSL 通信のサーバー検証に関するオプションを指定します。デフォルトのサーバー検証を用いる場合はこの引数を省略することができます。デフォルトでは実施されないサーバー検証のオプションを実施するときは、実施する検証オプションを表す nn::ssl::VerifyOption 列挙子を *verifyOpt* に指定します。複数の検証オプションを同時に設定したい場合は、その論理和を指定します。引数には、以下の値を設定することができます。

表 6-19. サーバー検証オプションに設定可能な値

値	説明
VERIFY_DATE	証明書の期限切れ検証を実施します。
USE_SESSION_CACHE	resumption を利用し、同じホストに連続で接続する際にセッションの再利用を試みます。
VERIFY_EV	EV 証明書検証を実施します。サーバー証明書が EV 証明書に紐づいたものでない限り、検証は失敗します。 信頼する EV 証明書は AddEVPolicyID() で設定することができます。
VERIFY_IGNORE	サーバー証明書検証を行いますが、検証結果は無視して接続します。検証結果は、GetCertVerifyErrors() で取得することができます。
GET_ALL_SERVER_CERT_CHAIN	SetServerCertBuffer() で指定したバッファにサーバー証明書を保存する際、証明書チェーン内の全証明書データを保存します。このオプションを指定しない場合は、サーバー証明書のみが保存されます。

nn::ssl::VerifyOption に定義されている上記以外のオプションはデフォルトで実施されます。

> **補足:** 証明書チェーン内の全証明書データを保存する場合、データは "[証明書データ長(4 Byte)]+[証明書データ]" が並ぶ形で保存されます。データの並び順は、チェーン先頭の証明書(=サーバー証明書)から、チェーン末尾(CA 証明書)の順です。ただし CA 検証が NG の場合、CA 証明書が特定できなため、CA 証明書データは含まれません。

6.2.4. 証明書と CRL の設定

サーバーとのハンドシェイクに使用する、証明書と CRL の 2 つのストア、クライアント証明書を接続クラスに設定します。

コード 6-22. 証明書と CRL の設定

```
class nn::ssl::Connection
{
    nn::Result SetServerCertStore(nn::ssl::CertStore& certStore);
    nn::Result SetClientCert(nn::ssl::ClientCert& clientCert);
    nn::Result SetCRLStore(nn::ssl::CrlStore& crlStore);
}
```

証明書ストアの登録は SetServerCertStore() で、CRL ストアの登録は SetCRLStore() で、クライアント証明書の登録は SetClientCert() で行います。

6.2.4.1. 証明書ストアのクラス

証明書ストアのクラスは nn::ssl::CertStore で定義されています。

コード 6-23. 証明書ストアのクラス

```
class nn::ssl::CertStore
{
    explicit CertStore();
    virtual ~CertStore(void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);

    nn::Result RegisterCert(const u8* pCertData, size_t certDataSize,
                            nn::ssl::CertId* pCertIdCourier=NULL);
    nn::Result RegisterCert(nn::ssl::InternalCaCert inCaCertName,
                            nn::ssl::CertId* pCertIdCourier=NULL);
    nn::Result UnRegisterCert(nn::ssl::CertId certId);
}
```

コンストラクタに引数はありません。インスタンスを用意し、Initialize() で初期化を行ってください。

メンバ関数の RegisterCert() で証明書を証明書ストアに登録します。引数 *pCertData* と *certDataSize* には、証明書データとそのバイトサイズを指定してください。機器に内蔵された証明書を証明書ストアに登録するためには、代わりに nn::ssl::InternalCaCert を引数に指定します。個別に証明書の登録を解除する必要がなければ、引数 *pCertIdCourier* を指定しなくてもかまいません。登録を解除していなかった証明書は、Finalize() を呼び出したときに登録が解除されます。証明書ストアに複数の証明書を登録する場合は、続けて RegisterCert() メンバ関数を呼び出してください。

メンバ関数の UnRegisterCert() で証明書ストアから個別に証明書の登録を解除することができます。RegisterCert() で証明書を登録したときに *pCertIdCourier* で受け取った証明書 ID を *certId* に指定してください。

証明書ストアが不要になったときには、必ず Finalize() を呼び出してください。

6.2.4.2. CRL ストアのクラス

CRL ストアのクラスは nn::ssl::CrlStore で定義されています。

コード 6-24. CRL ストアのクラス

```
class nn::ssl::CrlStore
{
    explicit CrlStore();
    virtual ~CrlStore(void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);

    nn::Result RegisterCrl(const u8* pCrlData, size_t crlDatasize,
                           nn::ssl::CrlId* pCrlIdCourier=NULL);
    nn::Result RegisterCrl(nn::ssl::InternalCrl inCrlName,
                           nn::ssl::CrlId* pCrlIdCourier=NULL);
    nn::Result UnRegisterCrl(nn::ssl::CrlId crlId);
}
```

コンストラクタに引数はありません。インスタンスを用意し、Initialize() で初期化を行ってください。

メンバ関数の RegisterCrl() で CRL を CRL ストアに登録します。引数 *pCrlData* と *crlDatasize* には、CRL

データとそのバイトサイズを指定してください。機器に内蔵された CRL を CRL ストアに登録するためには、代わりに nn::ssl::InternalCrl を引数に指定します。個別に CRL の登録を解除する必要がなければ、引数 *pCrlIdCourier* を指定しなくてもかまいません。登録を解除していなかった CRL は、Finalize() を呼び出したときに登録が解除されます。CRL ストアに複数の CRL を登録する場合は、続けて RegisterCrl() メンバ関数を呼び出してください。

メンバ関数の UnRegisterCrl() で CRL ストアから個別に CRL の登録を解除することができます。RegisterCrl() で CRL を登録したときに *pCrlIdCourier* で受け取った CRL の ID を *crlId* に指定してください。

CRL ストアが不要になったときには、必ず Finalize() を呼び出してください。

6.2.4.3. クライアント証明書のクラス

クライアント証明書のクラスは nn::ssl::ClientCert で定義されています。

コード 6-25. クライアント証明書のクラス

```
class nn::ssl::ClientCert
{
    explicit ClientCert();
    virtual ~ClientCert(void);

    nn::Result Initialize(const u8* pCertData, size_t certDataSize,
                          const u8* pPrivateKeyData, size_t privateKeyDataSize);
    nn::Result Initialize(nn::ssl::InternalClientCert inClientCertName);
    nn::Result Finalize(void);
}
```

コンストラクタに引数はありません。インスタンスを用意し、Initialize() で初期化を行います。

引数 *pCertData* と *certDataSize* には、クライアント証明書の証明書データとそのバイトサイズを指定してください。引数 *pPrivateKeyData* と *privateKeyDataSize* には、秘密鍵のデータとそのバイトサイズを指定してください。内蔵のクライアント証明書を使用する場合には、代わりに nn::ssl::InternalClientCert を指定します。

クライアント証明書が不要になったときには、必ず Finalize() を呼び出してください。

6.2.5. ハンドシェイク

メンバ関数の DoHandshake() で、接続クラスに設定された証明書ストア、CRL ストア、クライアント証明書を使って、接続先のサーバーとのハンドシェイクを行います。

コード 6-26. ハンドシェイク

```
class nn::ssl::Connection
{
    nn::Result SetServerCertBuffer(uptr bufferAddress, size_t bufferSize);
    nn::Result DoHandshake(void);
    nn::Result DoHandshake(size_t* pServerCertSize,
                           u32*    pServerCertNum = NULL);
}
```

サーバーから通知されたサーバー証明書についての情報が必要なときは、引数ありの DoHandshake() を呼び出してください。サーバー証明書のバイトサイズは *pServerCertSize* に指定された変数のポインタに格納されます。

サーバー証明書を格納するバッファは SetServerCertBuffer() で指定します。指定するバッファはデバイスメモリ以外から確保しなければなりません。また、指定するバッファの先頭アドレスは

nn::ssl::Connection::BUFFER_ALIGNMENT (4096 Byte) のアライメント、サイズは nn::ssl::Connection::BUFFER_UNITSIZE (4096) の倍数でなければなりません。SetServerCertBuffer() でバッファを指定せずに引数ありの DoHandshake() を呼び出した場合は、引数なしの DoHandshake() を呼び出したときと同じ動作になります。

6.2.6. データの送受信

ハンドシェイクに成功したことを確認できたら、SSL 接続経由でのデータの送受信を開始することができます。

コード 6-27. SSL 接続経由でのデータの送受信

```
class nn::ssl::Connection
{
    nn::Result Read(u8* pDataBuf, size_t dataBufSize,
                    size_t* pReadSizeCourier = NULL);
    nn::Result Peek(u8* pDataBuf, size_t dataBufSize,
                    size_t* pReadSizeCourier = NULL);
    nn::Result Write(const u8* pDataBuf, size_t dataBufSize,
                     size_t* pWrittenDataSizeCourier = NULL);
}
```

Read() と Peek() はともにデータの受信を行うメンバ関数ですが、Peek() で受信した場合は状態を変化させずに受信済みのデータを受け取ることができます。引数 *pDataBuf* と *dataBufSize* には、受信データを受け取るバッファとそのバイトサイズを指定してください。引数 *pReadSizeCourier* には、受け取ったデータのサイズが必要な場合にサイズを受け取る変数のポインタを指定してください。

Write() はデータの送信を行うメンバ関数です。引数 *pDataBuf* と *dataBufSize* には、送信するデータが格納されているバッファとそのバイトサイズを指定してください。引数 *pWrittenDataSizeCourier* には、送信されたデータのサイズが必要な場合にサイズを受け取る変数のポインタを指定してください。

6.2.7. 接続の切断

同時に使用することのできる SSL 接続の数は制限されています。そのため、不要になった接続クラスに対してメンバ関数 Shutdown() を呼び出し、SSL 接続の切断と終了処理を行ってください。

コード 6-28. SSL 接続の切断

```
class nn::ssl::Connection
{
    nn::Result Shutdown(void);
}
```

6.2.8. 終了

SSL ライブラリの使用を終了するときは nn::ssl::Finalize() を呼び出して終了処理を行ってください。

コード 6-29. SSL ライブラリの終了

```
nn::Result nn::ssl::Finalize(void);
```

6.3. HTTP 通信

HTTP(HTTP 通信)ライブラリは、指定された URL へ接続して HTTP プロトコルでのネットワーク通信を行うライブラリです。HTTPS 通信にも対応しています。

6.3.1. 初期化

HTTP ライブラリの初期化は nn::http::Initialize() の呼び出しで行われます。

コード 6-30. HTTP ライブラリの初期化

```
nn::Result nn::http::Initialize(uptr bufferAddress = 0, size_t bufferSize = 0);
```

HTTP ライブラリを利用するときは、事前にこの関数を呼び出しておかなければなりません。すでに初期化が行われている場合は、返り値に nn::http::ResultAlreadyInitializedErr が返されます。

bufferAddress と *bufferSize* を指定します。通信用バッファとして渡すバッファは **デバイスメモリ以外から確保** しなければなりません。また、バッファの先頭アドレスのアライメントは nn::http::BUFFER_ALIGNMENT (4096 Byte)、バッファのサイズは nn::http::BUFFER_UNITSIZE (4096) の倍数でなければなりません。

6.3.2. 接続クラスの生成

HTTP 接続には nn::http::Connection クラスを使用します。このクラスのインスタンスは、1 URL への接続と通信を単位に生成しなければなりません。システム全体およびアプリケーションで同時に使用することのできる通信リソース(接続数)には制限がありますので、接続数の管理には注意してください。

コンストラクタには、引数ありとなしの 2 種類が用意されています。引数なしのコンストラクタでインスタンスを生成した場合は、メンバ関数の Initialize() を呼び出して、接続先の URL とメソッド、デフォルトプロキシの使用についての設定を行う必要があります。

コード 6-31. HTTP 接続クラスのコンストラクタと初期化

```
class nn::http::Connection
{
    explicit Connection(void);
    explicit Connection(const char* pUrl,
                        RequestMethod method = REQUEST_METHOD_GET,
                        bool isUseDefaultProxy = true);
    nn::Result Initialize(const char* pUrl,
                          RequestMethod method = REQUEST_METHOD_GET,
                          bool isUseDefaultProxy = true);
}
```

pUrl には接続先の URL を指定します。

method には HTTP リクエストのメソッドを指定します。メソッドは以下の 3 種類から選択します。

表 6-20. メソッドの一覧

定義	メソッドの種類
REQUEST_METHOD_GET	GET メソッド。送信するデータは接続先の URL に埋め込みます。
REQUEST_METHOD_POST	POST メソッド。送信するデータは POST データとして追加します。

REQUEST_METHOD_HEAD	HEAD メソッド。接続先の URL からメッセージヘッダだけを受信します。

isUseDefaultProxy に true を指定した場合は、接続時にデフォルトのプロキシ設定を利用します。デフォルト以外のプロキシ設定を利用する場合はこの引数に false を指定し、別途プロキシの設定を行ってください。

Initialize() で返される nn::Result クラスがエラーのときは、Description でエラーの原因を判断することができます。

表 6-21. 初期化時に発生するエラー

Description の値	エラーの原因
ER_CONN_ADD	システム全体の通信リソースに空きがありません。
ER_REQ_URL	指定された URL が不正です。
ER_ALREADY_ASSIGN_HOST	すでに接続先の URL が指定されています。
ER_CONN_PROCESS_MAX	アプリケーションに割り当てられている通信リソースに空きがありません。

6.3.3. 通信設定

プロキシ設定やベーシック認証の設定、通信時に使用するソケットの受信バッファサイズの設定など、通信に関する設定を行います。これらの設定は接続を開始する前に行わなければなりません。

コード 6-32. 通信設定

```
class nn::http::Connection
{
    nn::Result SetProxy(const char* pProxyName, u16 port,
                        const char* pUserName, const char* pPassword);
    nn::Result SetBasicAuthorization(const char* pUserName,
                                     const char* pPassword);
}
```

デフォルトのプロキシ設定を利用しない場合は、SetProxy() でプロキシ設定を行います。*pProxyName* と *port* にはプロキシの URL とポート番号を、*pUserName* と *pPassword* には認証に使用するユーザーとパスワードを指定します。

接続時にベーシック認証が必要な場合は、SetBasicAuthorization() でベーシック認証の設定を行います。*pUserName* と *pPassword* には認証に使用するユーザーとパスワードを指定します。

6.3.3.1. HTTPS 通信を行う場合の設定

HTTPS 通信を行う場合は、CA 証明書や CRL 、クライアント証明書など、SSL 通信に必要な設定を行わなければなりません。

コード 6-33. HTTPS 通信の設定

```
class nn::http::Connection
{
    nn::Result SetRootCa(const u8* pCertData, size_t certDataSize);
    nn::Result SetRootCa(nn::http::InternalCaCertId inCaCertName);
    nn::Result SetRootCaStore(nn::http::CertStore& certStore);
    nn::Result SetCrl(const u8* pCrlData, size_t crlDataSize);
```

```
    nn::Result SetCrl(nn::http::InternalCrlId inCrlName);
    nn::Result SetCrlStore(CrlStore& crlStore);
    nn::Result SetClientCert(const u8* pCertData, size_t certDataSize,
                             const u8* pPrivateKeyData, size_t privateKeyDataSize);
    nn::Result SetClientCert(nn::http::InternalClientCertId inClientCertName);
    nn::Result SetClientCert(nn::http::ClientCert& clientCert);
    nn::Result GetSslError(s32* pResultCodeBuf) const;
    nn::Result SetVerifyOption(u32 verifyOption);
    nn::Result DisableVerifyOptionForDebug(u32 excludeVerifyOptions);
}
```

CA 証明書の登録は SetRootCa() で行います。証明書は証明書データを直接登録することも、機器に内蔵された CA 証明書を nn::http::InternalCaCertId で指定して登録することもできます。

複数の CA 証明書を設定する必要がある場合は、SetRootCa() を複数回呼び出して設定する以外にも、nn::http::CertStore クラスのインスタンスを引数に SetRootCaStore() を呼び出して設定することもできます。この場合、複数の接続で同じ CA 証明書セットを利用することができます。

CA 証明書セットを利用して通信しているときは、nn::http::CertStore クラスのインスタンスを破棄しないでください。

コード 6-34. nn::http::CertStore クラスの定義

```
class nn::http::CertStore
{
    explicit CertStore();
    virtual ~CertStore (void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);
    nn::Result RegisterCert(const u8* pCertData, size_t certDataSize,
                            nn::http::CertId* pCertIdCourier=NULL);
    nn::Result RegisterCert(nn::http::InternalCaCertId inCaCertName,
                            nn::http::CertId* pCertIdCourier=NULL);
    nn::Result UnRegisterCert(nn::http::CertId certId);
}
```

nn::http::CertStore クラスを利用するときは、必ず Initialize() で初期化を行ってください。CA 証明書の登録は RegisterCert() で行います。複数回呼び出すことで、複数の CA 証明書が登録された CA 証明書セットになります。

個別に CA 証明書の登録を解除する場合は、登録時に *pCertIdCourier* に返される値を引数に UnRegisterCert() を呼び出してください。インスタンスそのものが不要になったときは、必ず Finalize() で終了処理を行ってから破棄してください。

CRL の登録は SetCrl() で行います。CRL データを直接登録することも、機器に内蔵された CA 証明書を nn::http::InternalCrlId で指定して登録することもできます。ただし、現時点では内蔵 CRL は存在しません。

CA 証明書の登録と同じように、複数の CRL を設定する必要がある場合は、SetCrl() を複数回呼び出して設定する以外にも、nn::http::CrlStore クラスのインスタンスを引数に SetCrlStore() を呼び出して設定することができます。nn::http::CrlStore クラスの使い方は nn::http::CertStore クラスと同様です。

コード 6-35. nn::http::CrlStore クラスの定義

```
class nn::http::CrlStore
{
    explicit CrlStore();
    virtual ~CrlStore (void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);
    nn::Result RegisterCrl(const u8* pCrlData, size_t crlDataSize,
                           nn::http::CrlId* pCrlIdCourier=NULL);
    nn::Result RegisterCrl(nn::http::InternalCrlId inCrlName,
                           nn::http::CrlId* pCrlIdCourier=NULL);
    nn::Result UnRegisterCrl(nn::http::CrlId crlId);
}
```

クライアント証明書の設定は SetClientCert() で行います。設定の方法には、証明書と秘密鍵を指定する方法と機器に内蔵されているクライアント証明書を指定する方法、nn::http::ClientCert クラスのインスタンスを利用して複数の接続で同じクライアント証明書を利用する方法があります。

nn::http::ClientCert クラスを複数の接続で利用する場合は、通信中にインスタンスを破棄しないでください。

コード 6-36. nn::http::ClientCert クラスの定義

```
class nn::http::ClientCert
{
    explicit ClientCert();
    virtual ~ClientCert (void);

    nn::Result Initialize(const u8* pCertData, size_t certDataSize,
                          const u8* pPrivateKeyData, size_t privateKeyDataSize);
    nn::Result Initialize(nn::http::InternalClientCertId inClientCertName);
    nn::Result Finalize(void);
}
```

nn::http:ClientCert クラスの初期化関数 Initialize() には、証明書と秘密鍵を指定するものと、機器に内蔵されているクライアント証明書を指定するものがあります。インスタンスが不要になったときは、必ず Finalize() で終了処理を行ってから破棄してください。

デフォルトでは有効となっていないサーバーの検証オプションを SetVerifyOption() で有効にすることができます。デフォルトでは、VERIFY_COMMON_NAME(CommonName 検証)、VERIFY_ROOT_CA(RootCA 検証)、VERIFY_SUBJECT_ALT_NAME(SubjectAlternativeName 検証)が有効となっています。

現在、追加で有効にすることができる検証オプションは、VERIFY_DATE(証明書の期限切れ検証)、USE_SESSION_CACHE(resumption の利用)、VERIFY_EV(EV 証明書検証)の 3 つです。

DisableVerifyOptionForDebug() で検証オプションを無効にすることができますが、デバッグ目的以外での使用はなるべく控えてください。

HTTPS 通信中に発生したエラーは GetSslError() の引数 *pResultCodeBuf* に格納される値で判断することができます。格納されるエラーコードは nn::ssl::ResultCode で定義されている値です。

6.3.4. 送信データの設定

接続を開始してデータを送信する前に、メッセージフィールドのヘッダの追加や、POST データの追加を行います。後述する POST データ遅延設定モードを利用すると、接続を開始したあとに POST データを追加することができます。

コード 6-37. 送信前に行うヘッダ・データの追加

```
class nn::http::Connection
{
    nn::Result AddHeaderField(const char* pLabel, const char* pValue);
    nn::Result SetPostDataEncoding(nn::http::EncodingType type);
    nn::Result AddPostDataAscii(const char* pLabel, const char* pValue);
    nn::Result AddPostDataBinary(const char* pLabel, const void* pValue,
                                 size_t valueSize);
    nn::Result AddPostDataRaw(const void* pValue, size_t valueSize);
}
```

メッセージフィールドにヘッダを追加するには AddHeaderField() を呼び出してください。たとえば、*pLabel* と *pValue* に "Connection" と "keep-alive" を指定すると、Connection ラベルに ″keep-alive″ を指定したヘッダが追加されます。

POST データを追加するには、最初に SetPostDataEncoding() でエンコードの方法を指定し、AddPostDataAscii() などで POST データを追加します。

type に渡すエンコードの方法は、nn::http::EncodingType 列挙子に定義された値から選択します。

表 6-22. POST データのエンコード方法

定義	説明
ENCODING_TYPE_AUTO	すべて AddPostDataAscii() で追加した場合は URL エンコード、1 つでも AddPostDataBinary() で追加した場合はマルチパートで送信します。(デフォルト)
ENCODING_TYPE_URL	データを URL エンコードして送信します。POST データを追加するときは、AddPostDataAscii() を呼び出します。
ENCODING_TYPE_MULTIPART	データをエンコードせずに送信します。AddPostDataAscii() で追加された POST データはエンコードされません。

AddPostDataAscii() で追加された POST データ(ASCII 文字列)は、URL エンコードされて送信される場合にはラベル、データともにエンコードされます。POST データにバイナリが含まれる場合や、エンコード方法にマルチパートを指定した場合はラベルもデータもエンコードされません。

AddPostDataBinary() で追加された POST データ(バイナリデータ)は、エンコードされずにそのまま送信されます。

AddPostDataRaw() は上記 2 つの関数とは異なり、複数の POST データを一括して追加する場合に呼び出します。POST データにはバイナリ形式のみ指定可能です。この関数のあとに AddPostDataAscii() や AddPostDataBinary() を呼び出した場合は Description に ER_POST_ADDED_ANOTHER が設定されたエラーが返されます。また、再度 AddPostDataRaw() を呼び出した場合は前に追加したデータが破棄され、新たに追加したデータが有効になります。

6.3.4.1. POST データ遅延設定モード

POST データ遅延設定モードを利用すると、接続を開始したあとに POST データを送信することができます。このモードで POST データを送信する場合は、SendPostData*() が同期処理のため、送信が完了するまで処理がブロックされること

に注意してください。ただし、引数 *timeout* を持つオーバーロードは指定された時間で処理をタイムアウトさせることができます。タイムアウトした場合、内部で Cancel() が呼び出され、接続はキャンセル状態になります。

コード 6-38. POST データ遅延設定モードでの送信

```
class nn::http::Connection
{
    nn::Result SetLazyPostDataSetting(nn::http::PostDataType dataType);
    nn::Result NotifyFinishSendPostData(void);
    nn::Result SendPostDataAscii(const char* pLabel, const char* pValue);
    nn::Result SendPostDataAscii(const char* pLabel, const char* pValue,
                                 const nn::fnd::TimeSpan& timeout);
    nn::Result SendPostDataBinary(
                 const char* pLabel, const void* pValue, size_t valueSize);
    nn::Result SendPostDataBinary(
                 const char* pLabel, const void* pValue, size_t valueSize,
                 const nn::fnd::TimeSpan& timeout);
    nn::Result SendPostDataRaw(const void* pValue, size_t valueSize);
    nn::Result SendPostDataRaw(const void* pValue, size_t valueSize,
                               const nn::fnd::TimeSpan& timeout);
}
```

POST データ遅延設定モードは SetLazyPostDataSetting() の呼び出しで行います。引数 *dataType* で指定する送信データのタイプによって、POST データ送信のために呼び出すことのできる関数とデータの送信方法が異なります。送信データのタイプは nn::http::PostDataType 列挙子に定義された値から選択します。

表 6-23. POST データ遅延設定モードの送信データのタイプ

定義	説明
POST_DATA_TYPE_URLENCODE	URL エンコード形式で送信します。
POST_DATA_TYPE_MULTIPART	マルチパート形式で送信します。
POST_DATA_TYPE_RAW	RAW 形式で送信します。

URL エンコード形式で送信する場合、送信のために呼び出すことのできる関数は SendPostDataAscii() と SendPostDataBinary() です。SendPostDataAscii() を呼び出した場合は URL エンコードされた「(ラベル名)=(データ内容)」という内容のデータが送信され、SendPostDataBinary() を呼び出した場合はエンコードされていない同じ内容のデータが送信されます。

マルチパート形式で送信する場合、送信のために呼び出すことのできる関数は SendPostDataAscii() と SendPostDataBinary() です。1 回の呼び出しで送信されるデータは、boundary データとヘッダフィールドを先頭に付与した Chunked データとしてまとめて送信されます。ラベル名が "Content-Disposition: form-data; name=(ラベル名)" というヘッダフィールドとして boundary データのあとに付与されるのはどちらの関数でも同じですが、SendPostDataBinary() を呼び出した場合は "Content-Type: application/octet-stream¥r¥nContent-Transfer-Encoding: binary¥r¥n" というヘッダフィールドがその前に追加で付与されます。

RAW 形式で送信する場合。送信のために呼び出すことのできる関数は SendPostDataRaw() のみです。1 回の呼び出しで送信されるデータは、そのまま Chunked データとしてまとめて送信されます。

すべての POST データを送信し終えたあとは必ず、NotifyFinishSendPostData() を呼び出して、POST データの送信完了を通知してください。呼び出したあとは、処理が HTTP レスポンスの受信へと移行します。

6.3.5. 接続の開始

初期化で指定した URL に対して HTTP 接続を開始します。接続のために呼び出す関数には、接続が開始されるまで処理をブロックする Connect() と、すぐに処理を返す ConnectAsync() の 2 種類があります。また、接続をキャンセル状態にする Cancel() も用意されています。

コード 6-39. 接続の開始とキャンセル

```
class nn::http::Connection
{
    nn::Result Connect(void);
    nn::Result ConnectAsync(void);
    nn::Result Cancel(void);
}
```

Connect() はシステム全体で同時に接続可能な数の HTTP 接続がすでに行われていた場合に処理をブロックしますが、ConnectAsync() はすぐにエラーとして処理を返します。

Cancel() を呼び出すと HTTP 接続はキャンセル状態となり、以降その接続クラスでは通信が行えなくなります。

> **注意:** nn::http::Connection::Connect() もしくは nn::http::Connection::ConnectAsync() を使用して接続を開始した際、ヘッダ受信中にアクセスポイントの WAN 側のケーブルを抜いた場合無期限に受信を待ち続けてしまいます。
>
> これを防ぐには、一定時間経過後に通信を切断(nn::http::Connection::Cancel())する処理を実装する必要があります。

6.3.6. レスポンスの受信

Connect() などで接続が完了したあと、HTTP 通信の結果(レスポンス)を受信するには、メッセージボディならば Read() を、レスポンスステータスならば GetStatusCode() を、メッセージヘッダならば GetHeaderField() または GetHeaderAll() を呼び出します。引数 *timeout* を持つオーバーロードは指定された時間で処理をタイムアウトさせることができます。タイムアウトした場合、内部で Cancel() が呼び出され、接続はキャンセル状態になります。

コード 6-40. レスポンスの受信

```
class nn::http::Connection
{
    nn::Result Read(u8* pBodyBuf, size_t bufLen);
    nn::Result Read(u8* pBodyBuf, size_t bufLen,
                    const nn::fnd::TimeSpan& timeout);
    nn::Result GetStatusCode(s32* pStatusCodeCourier) const;
    nn::Result GetStatusCode(s32* pStatusCodeCourier,
                             const nn::fnd::TimeSpan& timeout) const;
    nn::Result GetHeaderField(
                const char* pLabel, char* pFieldBuf, size_t bufSize,
                size_t* pFieldLengthCourier = NULL) const;
    nn::Result GetHeaderField(
                const char* pLabel, char* pFieldBuf, size_t bufSize,
                const nn::fnd::TimeSpan& timeout,
                size_t* pFieldLengthCourier = NULL) const;
    nn::Result GetHeaderAll(
                char* pHeaderBuf, size_t bufSize,
```

```
                size_t* pLengthCourier = NULL) const;
    nn::Result GetHeaderAll(
                char* pHeaderBuf, size_t bufSize,
                const nn::fnd::TimeSpan& timeout,
                size_t* pLengthCourier = NULL) const;
}
```

Read() は、*pBodyBuf* に指定された *bufLen* バイトのバッファにメッセージボディのデータを格納します。指定されたバッファサイズよりもメッセージボディのサイズが大きい場合は、バッファにできる限りのデータを格納して、バッファ不足を表す返り値(nn::http::ResultBodyBufShortage)を返しますが、再度 Read() を呼び出すことで続くデータの読み取りを継続することができます。それ以外でエラーが返された場合は、レスポンスの受信自体に失敗しています。ただし認証失敗などの HTTP ステータスエラーの場合は、エラーメッセージを示すメッセージボディの受信に成功しているため、通信成功時と同様の動作になります(つまりバッファ不足のときは nn::http::ResultBodyBufShortage が、受信完了ならば IsSuccess() が true を返すインスタンスが返ります)。

GetStatusCode() は、引数 *pStatusCodeCourier* に指定された s32 型の変数にレスポンスステータスを格納します。通信エラーが発生したかどうかは、この関数で取得するレスポンスステータスで確認することができます。メッセージヘッダの受信が完了していない状態でこの関数の呼び出したときは、ヘッダ部の受信が完了するまで処理がブロックされます。先に Read() を呼び出していた場合はヘッダ部の受信は完了していますので、この関数は処理をすぐに返します。

GetHeaderField() は、引数 *pFieldBuf* に指定された *bufSize* バイトのバッファに *pLabel* で指定されたラベルのフィールドデータを格納します。*pFieldBuf* に NULL、*bufSize* に 0、*pFieldLengthCourier* に NULL 以外を指定した場合は、指定されたラベルのフィールドデータのバイトサイズを *pFieldLengthCourier* に格納します。

GetHeaderAll() はすべてのメッセージヘッダを引数 *pHeaderBuf* に指定された *bufSize* バイトのバッファに格納します。*pHeaderBuf* に NULL、*bufSize* に 0、*pLengthCourier* に NULL 以外を指定した場合は、*pLengthCourier* にメッセージヘッダの取得に必要なバッファサイズをバイト単位で格納します。メッセージヘッダの受信が完了していない状態でこれらの関数の呼び出したときは、ヘッダ部の受信が完了するまで処理がブロックされます。先に Read() を呼び出していた場合はヘッダ部の受信は完了していますので、これらの関数は処理をすぐに返します。

6.3.7. 接続状況などの取得

HTTP 接続クラス自身のステータスや発生したエラーのエラーコード、レスポンス受信の進捗を取得するための関数が用意されています。

コード 6-41. 接続状況などの取得

```
class nn::http::Connection
{
    nn::Result GetStatus(nn::http::Status* pStatusBuf) const;
    nn::Result GetError(nn::http::ResultCode* pResultCodeBuf) const;
    nn::Result GetProgress(size_t* pReceivedLen, size_t* pContentLen) const;
}
```

GetStatus() は、引数 *pStatusBuf* に指定されたバッファに接続状況を格納します。格納される接続状況は nn::http::Status 列挙子の値です。

GetError() は、引数 *pResultCodeBuf* に指定されたバッファにエラーコードを格納します。格納されるエラーコードは nn::http::ResultCode 列挙子の値です。

GetProgress() は、引数 *pReceivedLen* と *pContentLen* に指定された変数に受信済みメッセージボディのバイトサイズと予定されるメッセージボディの総サイズを格納します。総サイズは HTTP レスポンスのメッセージヘッダ "Content-Length" の値ですので、メッセージヘッダに "Content-Length" が存在しなかった場合、*pContentLen* には 0 が格納さ

れます。

6.3.8. 接続の終了

レスポンスの受信がすべて完了し、インスタンスが不要になったときは必ず Finalize() を呼び出してから破棄してください。接続先 URL を指定していない(Initialize() を呼び出していない)インスタンスに対して呼び出す必要はありません。

コード 6-42. 接続の終了

```
class nn::http::Connection
{
    nn::Result Finalize(void);
}
```

6.3.9. 終了

HTTP ライブラリの使用を終了するときは nn::http::Finalize() を呼び出して終了処理を行ってください。初期化が行われていない状態でこの関数を呼び出したときは nn::http::ResultNotInitializedErr が返されます。

コード 6-43. HTTP ライブラリの終了

```
nn::Result nn::http::Finalize(void);
```

7. 付録:ローカル通信の通信性能

本章では、ローカル通信のスループットの測定結果を記します。開発中のタイトルのスループットを評価する際の目安にしてください。なお、グラフィック表示等の処理負荷が大きいとき、電波状況が悪いときにはスループットが低下します。

7.1. 測定環境および測定結果

シールドボックス内に設置した 3 台の開発実機のうち、1 台を Master に設定、そこに 2 台の Client を接続しました。なお、Client のうち 1 台は送信のみを行い、もう 1 台は受信のみを行いました。

測定には、SampleDemos/uds/simple を測定用に改変して CTR-SDK 4.2.1 でコンパイルしたプログラムを使用しています。

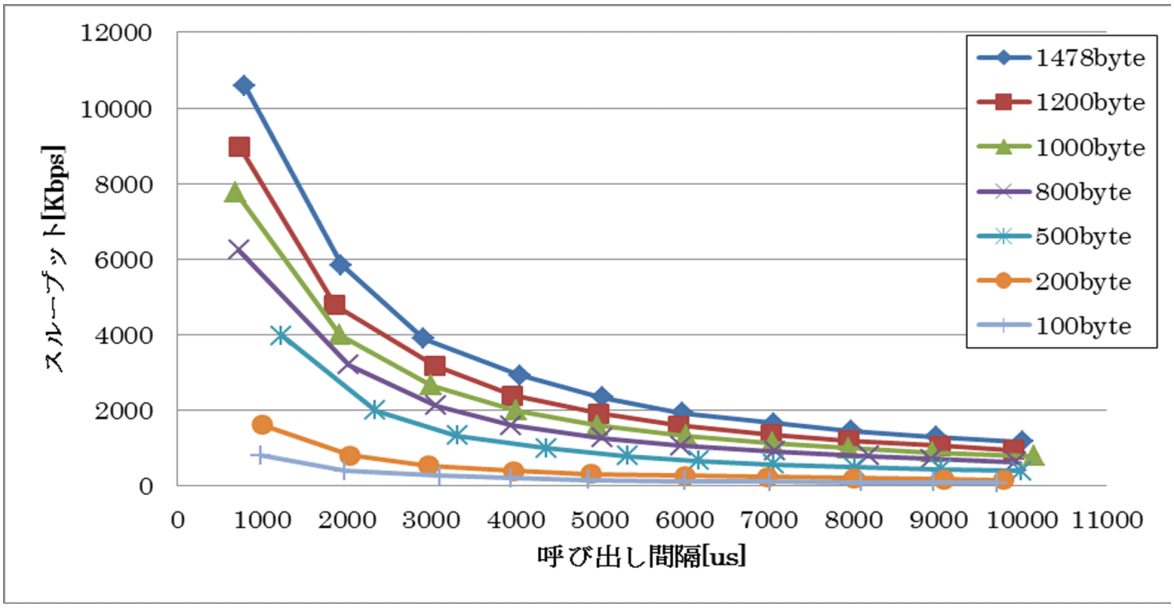
送信のみを行う Client からは、以下の要素を組み合わせて `SendTo` 関数を呼び出しました。

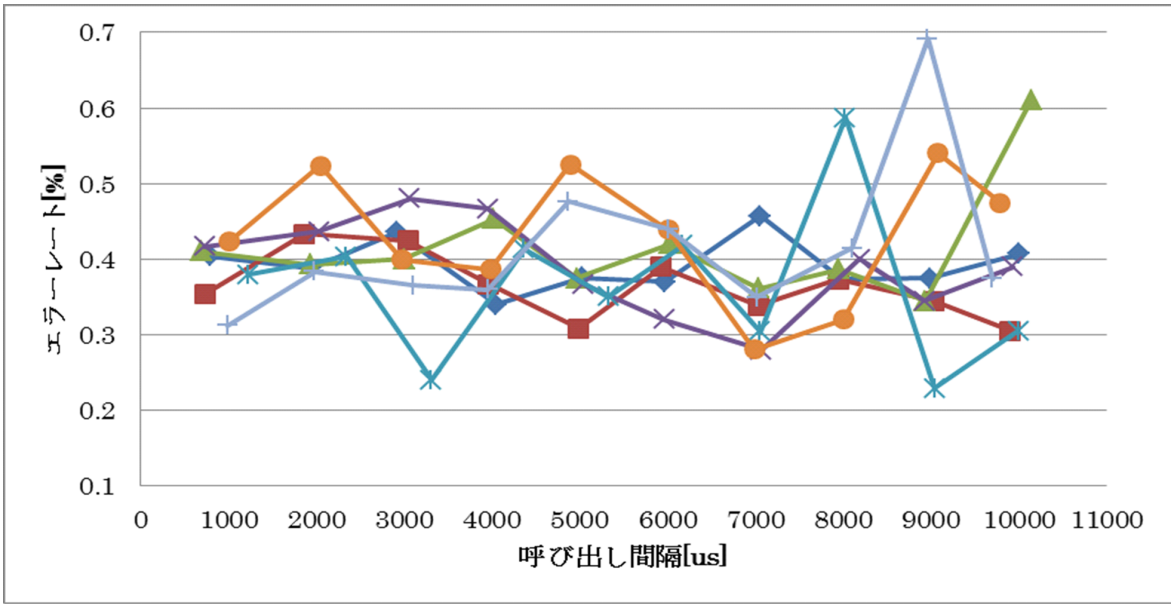
表 7-1. ローカル通信の性能測定で使用した測定要素

要素	条件
データサイズ	100 / 200 / 500 / 800 / 1000 / 1200 / 1478 byte
呼び出し間隔	1000 / 2000 / 3000 / 4000 / 5000 / 6000 / 7000 / 8000 / 9000 / 10000 us
送信オプション	0x00 / NO_WAIT / FORCE_DIRECT_BC / NO_WAIT | FORCE_DIRECT_BC

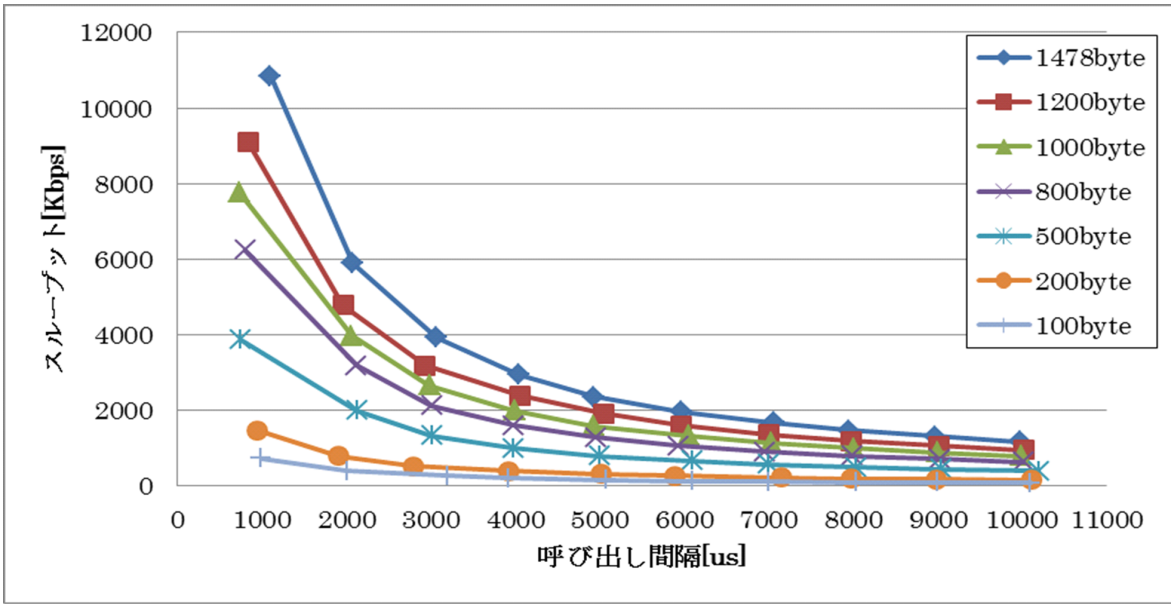
測定結果として、受信のみを行う Client で測定したスループットおよびエラーレートを送信オプションごとにグラフにしたものを以下に示します。

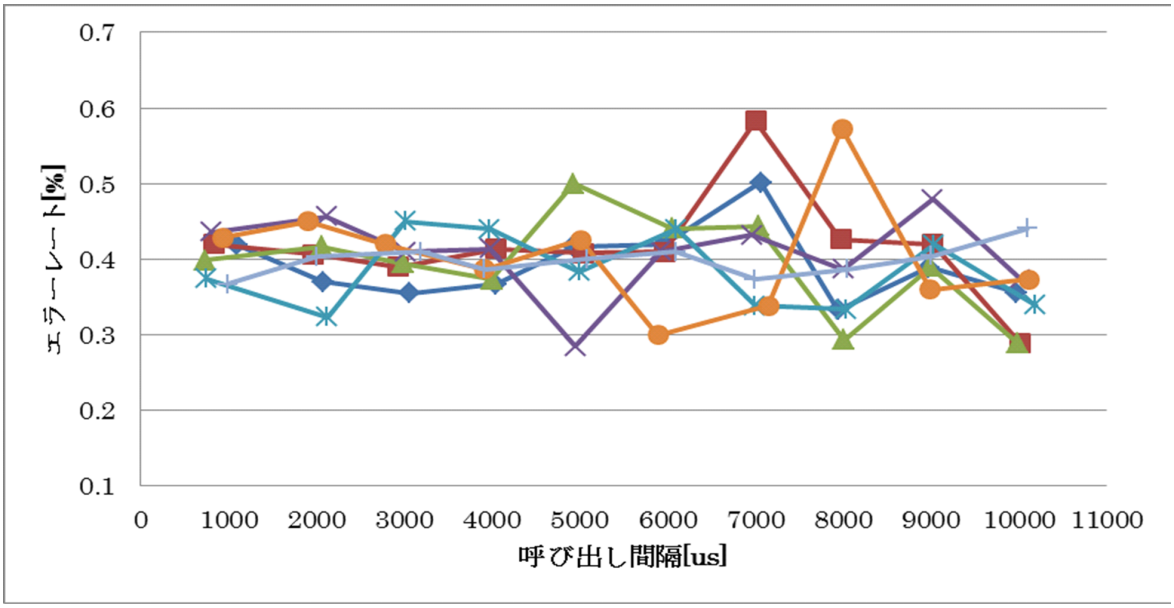
送信オプション = 0x00



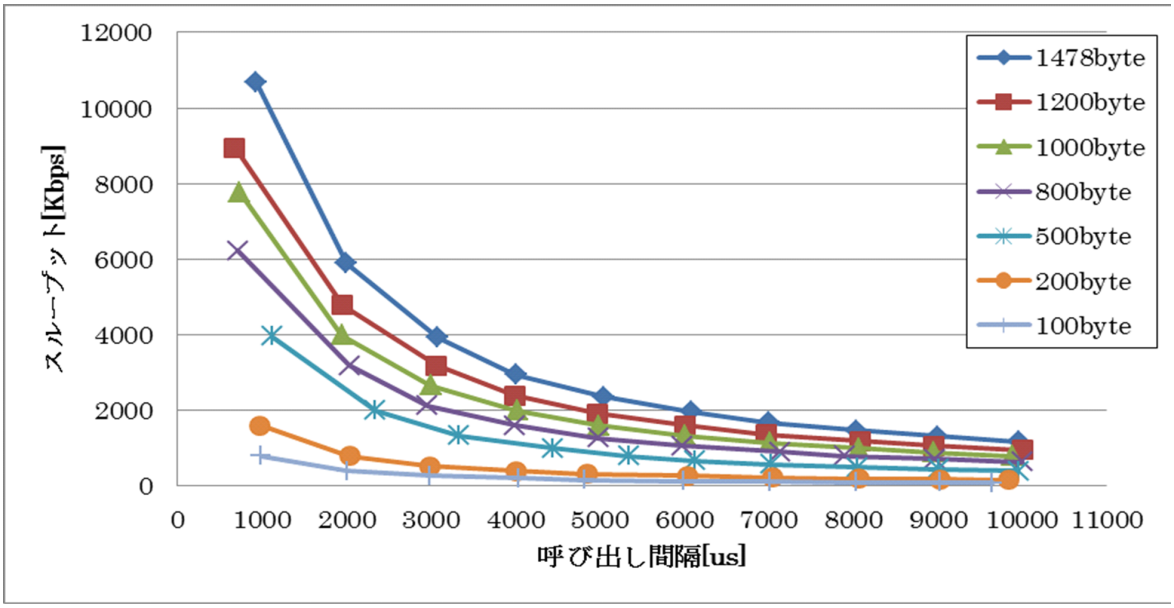


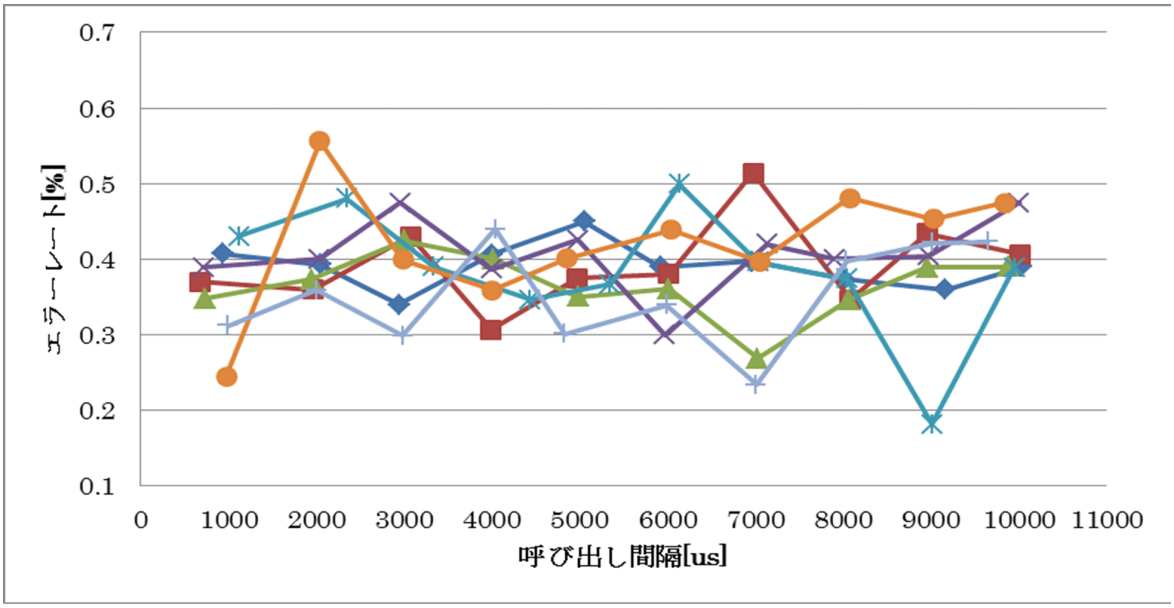
送信オプション = NO_WAIT



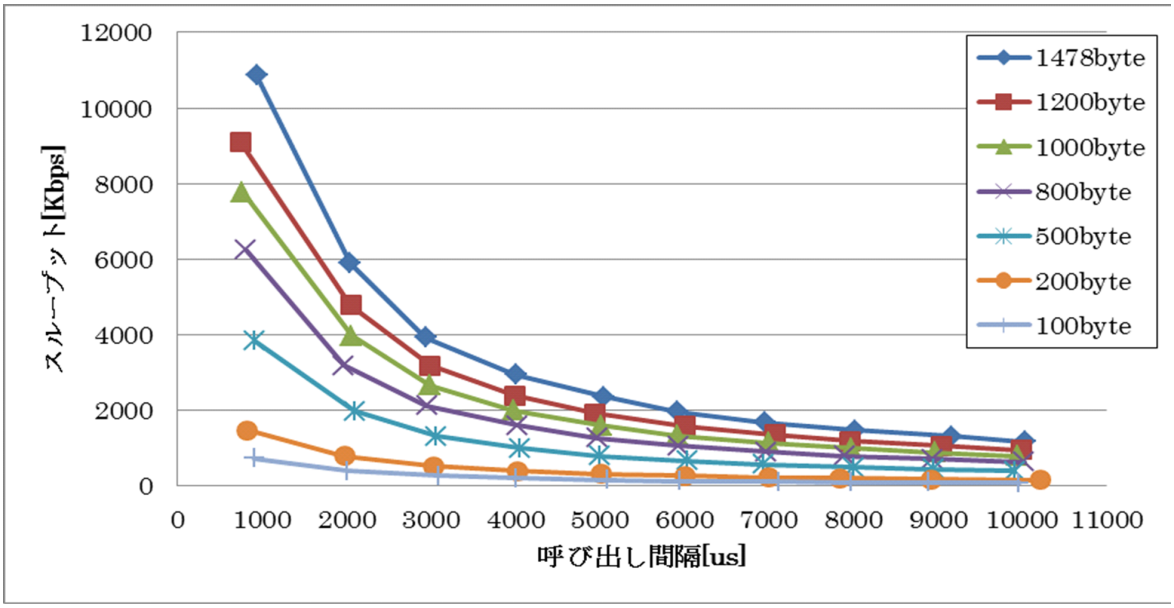


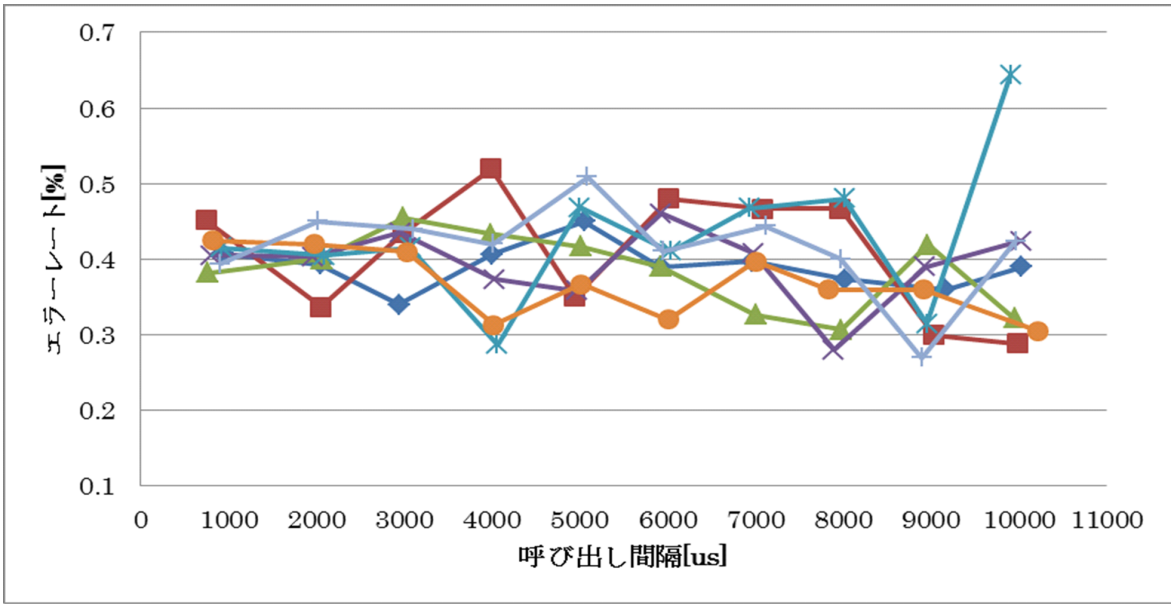
送信オプション = FORCE_DIRECT_BC





送信オプション = NO_WAIT | FORCE_DIRECT_BC





8. 付録:無線通信環境に関する諸注意

3DS の無線通信で用いられている 2.4 GHz 帯の無線周波数帯域は、Wi-Fi 機器をはじめとするさまざまな無線機器によって利用され、非常に混雑した無線周波数帯域になりつつあります。また、無線環境の悪化によって、時おり 3DS の無線通信が不安定になることは技術的に回避できない問題です。このため、「ほかの機器からの影響のないクリアな無線環境を用意し、無線通信を使った機能の開発を効率的に行いたい」という要望があっても、無線の特性により、完全にクリアな無線環境を作ることは容易ではありません。また、無線環境に対する理解不足により、実はクリアでない環境をクリアであると誤解し、通信に不具合が発生した際の問題の切り分けに多くの時間を費やしてしまうケースも発生しています。

この章では、無線環境に関する理解を深めていただくため、無線機器同士の干渉にまつわる情報をいくつか説明します。また、それを踏まえて、できうる限り、ほかの機器からの影響を受けない無線通信環境を用意する方法を説明します。

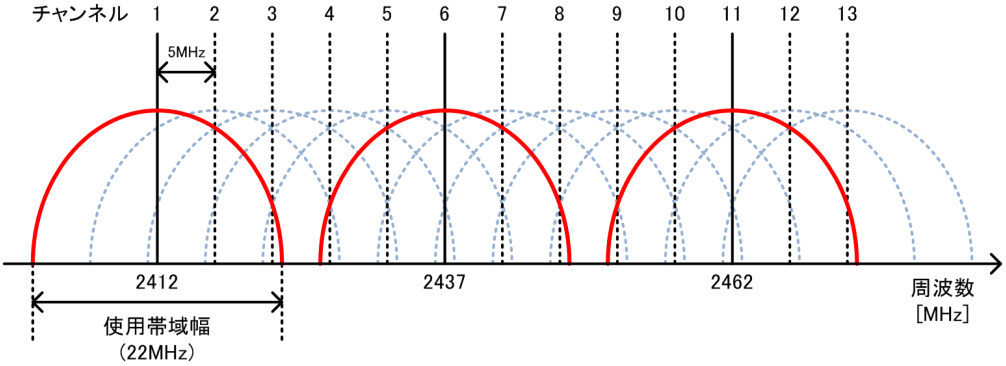
8.1. 無線通信環境を用意する際に注意すべきこと

ここでは、無線機器同士の干渉をなるべく発生させないようにするために知っておくべきこと、良好な無線環境が構築できているかを確認する際に注意すべきことを説明します。

8.1.1. ほかの Wi-Fi 機器のチャンネル設定

3DS の無線通信で使用する、2.4 GHz 帯の周波数チャンネルの割り当ての例を図 8-1 に示します。図のように、3DS の無線通信や一般的な Wi-Fi 通信は、ひとつのチャンネルとして割り当てられた帯域幅よりも広い帯域幅を使用します。よって、設定されたチャンネルが近い機器間では、両機器の通信が影響(干渉)し合うことになります。そのため、機器間で互いに影響しないようにする場合は、各機器のチャンネルを 5 つ以上離すことが推奨されています。

図 8-1. 2.4 GHz 帯のチャンネル割り当て



※ 使用できるチャンネル数は国によって異なります。

しかし、チャンネルを 5 つ以上離したとしても、互いの通信への影響が完全になくなるわけではありません。たとえば、チャンネル 1 とチャンネル 13 を使用した通信を行う機器同士であっても、機器同士の距離が非常に近ければ、互いの通信に影響を及ぼすことがあります。そのため、機器同士で影響を与えないようにするには、それぞれに 5 つ以上離れたチャンネルを設定し、かつ機器同士の間に十分な距離(少なくとも 2 m 以上)を保つ必要があります。

> **補足:** 機器の送信出力や利得、受信感度、電波伝搬の環境によって必要な距離は異なります。

8.1.2. Wi-Fi 機器以外の機器の影響

2.4 GHz 帯の周波数帯域は、3DS の無線通信や一般的な Wi-Fi 通信のような IEEE802.11 による通信のほかにも、さまざまな通信方式や電子機器で使用されることが許された周波数帯域です。代表的なものとしては、電子レンジ、コードレス電話、Bluetooth 機器(PC やスマートフォンの周辺機器等)などがあげられます。IEEE802.11 には、これらと効率的に共存する機能はありませんので、これらの信号の影響を受けることで通信が不安定になることがあります。

クリアな無線環境を得るためには、これらの機器は停止する必要があります。特に最近では、スマートフォンに Bluetooth が標準的に搭載されていますので、試験環境へスマートフォンを持ち込む際には注意が必要です。

8.1.3. パケットキャプチャ使用時の注意

AirPcap(Riverbed Technology 社製の無線 LAN プロトコルアナライザ)などに代表される市販のパケットキャプチャツールを使用することで、無線パケットをキャプチャすることができます。通信試験を円滑に行うために、このようなパケットキャプチャツールを導入している開発者の方もいらっしゃると思いますが、パケットキャプチャツールを使用しても無線環境の良し悪しは判断できないという点に注意いただく必要があります。つまり、キャプチャされる無線パケットの数が少ない、いわゆる空いている状況であっても通信が不安定になることがありうるということです。

パケットキャプチャツールでは、IEEE802.11 とは異なる通信方式の無線パケットはキャプチャされません。また、IEEE802.11 の無線パケットであっても、受信電力が弱いなどが原因で大部分を正しく復号できなかったものはキャプチャされません。キャプチャされない無線パケットであっても物理層や MAC 層のような無線通信の低層の処理においては影響が発生しますので、そのような無線パケットによって通信が不安定になることがあります。

> **補足:** 受信電力が弱いと MAC 層のプロトコルが正しく動作しないことがあり、キャプチャ可能な無線パケットよりも、キャプチャされない無線パケットが深刻な影響を通信に及ぼす場合もあります。

8.2. ほかの機器からの影響を受けない無線通信環境を用意する方法

2.4 GHz 帯を使用する無線通信機器の普及が著しい現状を考慮すると、高機能な電波暗箱や電波暗室を使用する以外に、確実にクリアな無線環境を得ることは難しいと言えます。ただ、完全にクリアとは言えないまでも、周辺装置のチャネル設定や機器間の距離に注意し、市販の電波シールドや同軸ケーブルを使用することで比較的クリアな無線環境を実現することは可能です。

> **補足:** ここで紹介する方法で用意した無線環境であっても、ほかの無線機器からの影響が完全にはなくならないという認識を持って開発や試験を運用してください。無線環境を原因とする事象が発生しうることを認識することは、不要な混乱を回避し、作業を効率的にする上で重要になります。

8.2.1. 電波シールドの使用

クリアな無線環境を得るために、電波を遮蔽するシールドを使用するという方法があります。簡易的な袋状のもの、箱状のもの、テント状のものなど、大小さまざまなシールド製品が市販されています。これらは外部からの干渉電波をある程度遮蔽できますので、通信が安定する効果が期待されます。しかし、市販のシールド製品の多くは、完全に外部からの影響を遮断できるほど十分な遮蔽性能を持たないという点に注意していただく必要があります。

あくまで目安ですが、シールド製品の遮蔽性能(減衰性能)が 40 dB 以下であれば、少なくとも同じ部屋の中にある無線機器からの影響は排除できないと考えていただく必要があります。クリアな通信環境を得るためには、60 dB 以上の減衰性能を持ったシールドを用い、シールドのすぐ近くには無線装置を置かないようにする必要があります。

> **補足:** 減衰した無線パケットによる悪影響を、シールドが助長することもありますので注意が必要です。

図 8-2. 各種電波シールド

8.2.2. 同軸ケーブルの使用

デバッガでは、無線アンテナの代わりに同軸ケーブルによって装置同士を結合し、同軸ケーブルを介した通信を行わせることもできます。電波シールドと同様に、ほかの無線通信の影響を抑制する効果が期待されます。ただし、ケーブルを介してやり取りされる信号は、無線通信の信号と同一のため、これも外部からの影響を完全になくすことができるものではありません。数メートル以内の無線装置からは影響を受けるという前提で使用していただく必要があります。

9. 付録:すれちがい通信中継

すれちがい通信中継は、すれちがい通信中継サーバーを利用して、特定のアクセスポイントに接続したユーザー間でバケツリレーのようにすれちがいデータを渡す仕組みです。この仕組みにより、同じ時間、同じ場所でなければ発生しなかったすれちがい通信の機会を、より多くのユーザーにもたらすことができます。

以下のアクセスポイントに接続すると、すれちがい通信中継が行われます。

- ニンテンドーゾーン
- ニンテンドー3DSステーション
- FREE SPOT(※)
- Wifine(※)

※ 一部の施設では利用できない場合があります。

補足: 本体のインターネット設定に登録されているアクセスポイントに接続した際には、すれちがい通信中継は行われません。

9.1. 通常のすれちがい通信との違い

すれちがい通信中継で行われるすれちがい通信の処理と通常のすれちがい通信で行われる処理には、以下のような点で違いがあります。

補足: すれちがい通信中継が行われるとアプリケーションの企画上問題が発生する場合には、非対応タイトルリストへの登録を弊社窓口までご連絡ください。

9.1.1. すべてのすれちがいデータが処理の対象となる

通常のすれちがい通信では通信する相手の本体に同じタイトルのすれちがいデータがなければ通信を行いませんが、すれちがい通信中継では本体に登録されているすべてのすれちがいデータで通信が行われます。そのため、各タイトルで 1 回ずつ、最大で 12 回のすれちがい通信が行われます。

すれちがい通信中継サーバーに同じタイトルのすれちがいデータがある場合は、通常のすれちがい通信と同様に、すれちがいデータの受信および送信が行われます。すれちがい通信中継サーバーに同じタイトルのすれちがいデータがない場合は、すれちがい通信中継サーバーへのすれちがいデータの送信(アップロード)だけが行われます。

9.1.2. フレンド向けのデータは送受信されない

すれちがい通信中継はフレンドではないユーザーとのすれちがい通信として処理されます。そのため、すれちがいデータ作成時の引数 *messageTypeFlag* に MESSAGE_TYPEFLAG_FRIEND のみが指定されているデータはすれちがい通信中継サーバーにアップロードされません。

このことにより、アプリケーションの仕様によっては、フレンド関係にあるユーザー間で、直接すれちがい通信が行われた場合とすれちがい通信中継が行われた場合とで渡されるすれちがいデータが異なるという事象が発生することに留意してください。

9.1.3. 交換の相手が異なる

すれちがい通信中継の仕様上、送受信モードが「交換」のすれちがいデータは、必ずしもすれ違ったユーザー同士で交換されるとは限りません。

たとえば、A → B → C の 3 人が順にすれちがい通信中継による交換を行った場合、以下のような挙動となります。

- A はすれちがい通信中継サーバーに自分のデータを送信する。(すれ違っていない状態となる)
- B は A のデータを受信し、すれちがい通信中継サーバーに自分のデータを送信する。
- C は B のデータを受信し、すれちがい通信中継サーバーに自分のデータを送信する。

つまり、B は A のデータを受信しているが、B のデータを A が受信していないことになります。

9.2. すれちがい通信中継のテスト

開発機でのすれちがい通信中継のテストは、BossLotcheckTool を用いて行います。詳しくは CTR のツール「BossLotcheckTool」のリファレンスを参照してください。なお、すれちがい通信中継を行うには、開発機を複数台(最低でも 2 台、3 台以上推奨)用意する必要があります。

> **補足:** 開発機では「9. 付録:すれちがい通信中継」に記載されているアクセスポイントと接続しても、すれちがい通信中継の処理は行われません。また、すれちがい通信中継サーバーは開発機と製品機で送受信するすれちがいデータを区別するため、開発機と製品機の間ですれちがい通信が行われることはありません。
>
> すれちがい通信中継のテスト機能は CTR-SDK に含まれる BossLotcheckTool で対応しています。

すれちがい通信中継のテストの目的は、「9.1. 通常のすれちがい通信との違い」で説明した違いによる挙動の変化が企画上許容可能なものかどうかの確認です。通常のすれちがい通信処理が正しく実装されていれば送受信されるすれちがいデータ自体の問題やエラーは発生しませんので、すれちがい通信中継でも正常に通信できるかを確認する必要はありません。

9.3. BossLotcheckTool を使用したテスト手順

以下の手順に沿って、すれちがい通信中継のテストを行ってください。

1. すれちがいデータを登録する。
2. 手順 1 の開発機で BossLotcheckTool を起動し、起動メニューから「Check functions」→「SPRelay」→「Forced Task Start」の順に選択する。
3. 上画面に表示されているメッセージが「Executing SPRelay...」から変化するまで待つ。
4. 変化後のメッセージが「Executed(HttpStatus:200)」であることを確認する。
   ほかのメッセージが表示された場合はすれちがい通信中継が正常に完了していません。表示されるメッセージの詳細については、「9.3.1. Forced Task Start の実行結果」を参照してください。
5. 別の開発機で手順 1～4 を実行する。

補足: このテスト手順で選択した「Forced Task Start」では、すれちがい通信中継処理が強制的に実行されます。また、同じ通信相手とは最大で 8 時間通信しないという制限も無視されます。なお、開発機が接続しているアクセスポイントを通して実行されるため、ニンテンドーゾーン等の特別な通信環境を用意しなくてもテストすることができます。本体にすれちがいデータが 1 つも登録されていない状態やペアレンタルコントロールで制限された状態であってもすれちがい通信中継処理は実行されますが、Forced Task Start の実行結果はエラーとなります。

このテストを初めて行ったとき(すれちがい通信中継サーバーに初めて接続したとき)には、すれちがいデータの送信のみが行われます。その後のテストで別の開発機を用いたときには、すれちがいデータの送信と受信の両方が行われ、受信した場合にはすれちがい通信と同様におしらせランプが緑色に点灯します。

注意: 「SPRelay」を選択したあとに表示されるメニューの「Enable」は選択しないでください。選択した場合、ニンテンドーゾーンのビーコンを検知したときにすれちがい通信中継の処理が行われるように設定されるため、通常のすれちがい通信を対象としたテストを妨げてしまう可能性があります。

9.3.1. Forced Task Start の実行結果

Forced Task Start の実行結果は以下のフォーマットで表示されます。

```
$TaskResult(HttpStatus:$HttpStatusCode)
```

$TaskResult の部分には、すれちがい通信中継のために行われている BOSS の処理(すれちがい通信中継タスク)の実行結果が表示されます。表示される実行結果とその意味については、以下の表を参照してください。

表 9-1. すれちがい通信中継タスクの実行結果

実行結果	実行結果の意味と詳細
Executed	すれちがい通信中継タスクが正常に実行されました。なお、$HttpStatusCode の部分には必ず 200 が表示されます。 正常に実行された場合でも、以下の理由ですれちがいデータの送受信が行われないことがあります。 ● 同じアクセスポイント、同じ開発機で連続してすれちがい通信中継を実行した。(アップロードしたすれちがいデータがほかの本体に渡されない限り、アップロード自体が行われません) ● 使用したアクセスポイントで初めてすれちがい通信中継を実行した。(すれちがいデータのアップロードのみが行われます)
Error	すれちがい通信中継タスクの実行でリトライ不可能なエラーが発生しました。 この実行結果が表示された場合には、$HttpStatusCode に表示された HTTP ステータスコードおよび以下の設定を確認してください。 ● ペアレンタルコントロールで、すれちがい通信が制限されていないか。 ● すれちがいデータが本体に登録されているか。
Retry	すれちがい通信中継タスクの実行でリトライ可能なエラーが発生しました。 この実行結果が表示された場合には、以下の設定を確認してください。 ● 通信が切断されていないか。 ● 無線通信の電波環境は悪くないか。 ● すれちがい通信中継の実行中に、本体の開閉や無線スイッチの操作を行っていないか。

$HttpStatusCode の部分には、すれちがい通信中継サーバーから返ってきた HTTP ステータスコードが表示されます。表示される HTTP ステータスコードとその意味については、以下の表を参照してください。

表 9-2. すれちがい通信中継サーバーから返される HTTP ステータスコードとその意味

HTTP ステータスコード	HTTP ステータスコードの意味
200	すれちがい中継処理が成功しました。
400	すれちがい中継処理が失敗しました。 クライアント側のリクエスト(開発機側の設定)に問題があります。
500	すれちがい中継処理が失敗しました。 すれちがい通信中継サーバー側のエラーです。
503	すれちがい中継処理が失敗しました。 すれちがい通信中継サーバーがメンテナンス中です。

補足: すれちがい通信中継サーバーが返す HTTP ステータスコードは、今後変更される可能性があります。

更新履歴

Version 1.6 2016-06-24

変更

- 3.3.3. 接続状態の取得
  - いつの間に通信拠点への接続検出方法を追記しました。

Version 1.5 2016-05-10

変更

- 4.2.1.2. タスクのプロパティ設定
  - bossタスクの消尽回数が減る条件について追記しました。
- 4.2.2.1. NS アーカイブ、NS データ
  - SetOptoutFlag()の説明に「おしらせを受け取らない」設定した場合と同じ挙動になることを追記しました。
- 4.2.2.4. BOSS 利用のための準備
  - nn::boss::UnregisterStorage() で BOSS ストレージの登録解除をした後、登録されたタスクが残っている場合について注意文を追記しました。

Version 1.4 2015-11-05

変更

- 全般
  - アライメント制約の定義名を追記しました。
- 3.2.10. 擬似クライアント
  - パッチや改訂版をリリースする場合の注意について追記しました。
- 4.1.1. 動作の概要
  - すれちがい通信が行われない場面についての補足を修正しました。
- 4.1.3.1. アクセスの開始(オープンと作成)
  - すれちがいボックスが作成可能かどうかを取得する機能について追記しました。
- 4.1.4.1. 新規作成
  - すれちがい通信で送信または受信のいずれかだけが成功する状況について追記しました。
- 4.2. いつの間に通信
  - ニンテンドーゾーンの地域別配信について記述を削除しました。
- 4.2.1.3. タスクの動作設定
  - ニンテンドーゾーンの地域別配信について記述を削除しました。
- 4.2.2. Nintendo アーカイブダウンロードタスク(NADL タスク)
  - BOSSデータサーバーの使い方についてはサーバーログイン後に閲覧できる「ヘルプ」を参照するように記述を修正しました。
- 4.2.4. データアップロードタスク
  - データアップロードタスクの再利用について情報を修正しました。
- 4.2.5. DataStore アップロードタスク
  - DataStore アップロードタスクの再利用について情報を修正しました。
- 6.1.1. 初期化
  - バッファのサイズ要件を追記しました。
- 6.1.2. ソケットの作成
  - ソケット作成時に発生するエラーについて修正しました。
- 6.1.3. アドレスとポート番号の結び付け
  - アドレス、ポート番号の結び付け時に発生するエラーについて修正しました。

- 6.1.4. 動作モード
  - 動作モードの設定・取得時に発生するエラーについて修正しました。
- 6.1.5. 接続の待ち受け
  - 接続の待ち受け時に発生するエラーについて修正しました。
- 6.1.6. リモートホストへの接続
  - リモートホストへの接続時に発生するエラーについて修正しました。
  - リモートホストへの接続の確認時に発生するエラーについて修正しました。
- 6.1.7. データの受信
  - データの受信時に発生するエラーについて修正しました。
- 6.1.8. データの送信
  - データの送信時に発生するエラーについて修正しました。
- 6.1.9. オプション設定
  - オプション設定時に発生するエラーについて修正しました。
- 6.1.10. ソケットの切断
  - ソケットの切断時に発生するエラーについて修正しました。
- 6.1.11. ソケットの破棄
  - ソケットの破棄時に発生するエラーについて修正しました。
- 6.1.13. ユーティリティ関数
  - ソケットアドレスの取得時に発生するエラーについて修正しました。
  - nn::socket::GetAddrInfo() によるホスト情報の取得時に発生するエラー について修正しました。
  - nn::socket::GetNameInfo() によるホスト情報の取得時に発生するエラーについて修正しました 。
- 6.2.5. ハンドシェイク
  - バッファはデバイスメモリ上に確保してはいけないことを追記しました。
- 6.3.1. 初期化
  - バッファはデバイスメモリ上に確保してはいけないことを追記しました。

Version 1.3 2015-04-28

追加

- 4.1.4.5. 情報の更新
- 4.2.1.10. BOSS ライブラリのエラーハンドリング

変更

- 3.1.3.2. ネットワークの新規構築
  - Config ツールのメニュー階層を現行のツールに合わせて修正しました。
- 3.3. 自動接続
  - Config ツールのメニュー階層を現行のツールに合わせて修正しました。
- 3.3.2. 自動接続
  - 自動接続の接続先の優先順位について追記しました。
- 4.2.1.4. タスクの登録と実行
  - 「タスクの削除」としていた文言を「タスクの登録解除」に表現に改めました。
- 4.2.2.4. BOSS 利用のための準備
  - 登録できる BOSS ストレージの最大数と自動登録解除について追記しました。
- 4.2.2.5. NADL タスクの登録
  - DNSサーバーのIPアドレスの確認方法について変更しました。
- 4.2.2.8. ダウンロードデータの読み込み
  - タスクを再登録した場合に重複受信防止機能が正しく働かなくなることについて記述を修正しました。
  - NS データの破損、改竄について返されるエラーごとの判定方法を追記しました。
- 5.4.4.1. 独自サービス向けサービストークン
  - アプリケーションの更新が必要な場合のシーケンスをニンテンドーeショップのパッチページへのジャンプすることと記載しました。

- 6.1.7. データの受信
  - MSG_PEEKN を MSG_PEEK に修正しました。
  - データグラムソケットでの受信時、1 回の呼び出しで受信できるデータの最大サイズを追記しました。
- 6.1.8. データの送信
  - データグラムソケットでの送信時、1 回の呼び出しで送信できるデータの最大サイズを追記しました。

Version 1.2 2015-01-15

変更

- 4.2.2.2. BOSS ストレージ
  - おしらせリストに関する補足内容を修正しました。
- 4.2.2.8. ダウンロードデータの読み込み
  - 重複ダウンロードに関する注意事項を修正しました。
- 6.1.7. データの受信
  - nn::socket::SockAddrIn 構造体ポインタを指定した際の説明を修正しました。
- 9.2. すれちがい通信中継のテスト
  - BossLotcheckTool 参照先を追記しました。

削除

- プラットフォームの表記について
  - プラットフォーム表記を Readme に転記したため、ページを削除しました。

Version 1.1 2014-11-10

変更

- 4.2.3.2. NSA リストの取得
  - NSA リストが本体言語設定を参照する事を追記しました。

Version 1.0 2014-09-04

追加/変更

- 初版