Rev.1.2 2018/06/29

FL-net Control For Windows®（KE-SFL3WIN）
DLL 版 I/F 仕様書
（FA リンクプロトコル仕様 Ver.3 対応版）

株式会社日立ケーイーシステムズ

改訂来歴

Rev.	改訂日	改訂箇所	改訂内容
1.0	2013/07/01		新規作成。 ※FL_KE.DLL(Ver.2.0) I/F 仕様書（Rev.1.02,2007/02/20 発行版）を元に、FA リンクプロトコル仕様 Ver.3.0 に対応する。変更内容の主な項目を以下に示す。
		関連文献	文献を追加
		2.1	構成図を変更 ハードウェア仕様にハードディスク容量追加。USB ポートをオプションに変更。 動作対象 OS に Windows® Vista、7 を追加。XP の SP3 を追加。 ネットワーク設定に Packet ドライバの項目追加。 プロテクトキードライバ設定をオプションに変更。 実行時に必要なファイルに Packet ドライバ追加。 実行時に使用するレジストリ情報にライセンス認証データ追加。
		2.2	対応開発コンテナに Microsoft Visual C++® 2005 を追加。 開発時に必要なファイルに Packet ドライバ追加。
		3.1	機能一覧にコマンドサーバ、負荷測定、デバイスレベルネットワークを追加。
		3.3	ネットワーク管理にコンフィギュレーション用パラメータ、ノード番号、トークン監視時間、最小許容フレーム間隔、ログ情報(Ver3.00 版)を追加。
		3.5.1 3.5.2 3.5.3	実装メッセージ一覧のネットワークパラメータライト、停止指令、運転指令を変更。
		3.6	コマンドサーバ追加
		3.7	負荷測定追加
		3.8	デバイスレベルネットワーク追加
		4	I/F 仕様にコマンドサーバ関数、負荷測定関数、デバイスレベルネットワーク関数を追加。
		4.1 4.1.2 4.1.9 4.1.12 4.1.14 4.1.15 4.1.16 4.1.17 4.1.18 4.1.19 4.1.20 4.1.21 4.1.22 4.1.23 4.1.24	メモリアクセス関数に HFA_LinkInDefault 関数、 HFA_GetMyNodeLogV3 関数、 HFA_GetCommon 関数、 HFA_GetNodeName 関数、 HFA_SetNodeNo 関数、 HFA_SetNodeNo 関数、 HFA_SetTokenWatchTime 関数、 HFA_GetTokenWatchTime 関数、 HFA_SetMinFrameInterval 関数、 HFA_GetMinFrameInterval 関数、 HFA_SetIP 関数、 HFA_GetIP 関数、 HFA_SetConfigParam 関数、 HFA_GetConfigParam 関数を追加。

(次頁に続く)

Rev.	改訂日	改訂箇所	改訂内容
1.0	2013/07/01	4.2 4.2.16 4.2.17	メッセージ送信関数に HFA_RepWriteNetParam 関数、 HFA_RepControlEquipment 関数を追加。
		4.3 4.3.4	DLL 制御関数に HFA_SetCallbackV3 関数を追加。
		4.4 4.4.1 4.4.2 4.4.3 4.4.4 4.4.5 4.4.6	コマンドサーバ関数に HFA_StartCmdServerUdp 関数、 HFA_StopCmdServerUdp 関数、 HFA_StartCmdServerTcp 関数、 HFA_StopCmdServerTcp 関数、 HFA_GetCmdServerUdpStatus 関数、 HFA_GetCmdServerTcpStatus 関数を追加。
		4.5 4.5.1 4.5.2 4.5.3 4.5.4 4.5.5 4.5.6	負荷測定関数に HFA_StartTokenTimeMeasure 関数、 HFA_StopTokenTimeMeasure 関数、 HFA_GetTokenTimeMeasure 関数、 HFA_StartDataLogMeasure 関数、 HFA_StopDataLogMeasure 関数、 HFA_GetDataLogMeasure 関数を追加。

(次頁に続く)

Rev.	改訂日	改訂箇所	改訂内容
1.0	2013/07/01	4.6	デバイスレベルネットワーク関数に
		4.6.1	HFA_GetSlaveNodeLinkStatus 関数、
		4.6.2	HFA_SetIO 関数、
		4.6.3	HFA_GetIO 関数、
		4.6.4	HFA_SetOutputStatus 関数、
		4.6.5	HFA_GetOutputStatus 関数、
		4.6.6	HFA_GetInputStatus 関数、
		4.6.7	HFA_ReadInputData 関数、
		4.6.8	HFA_ReadInputBitData 関数、
		4.6.9	HFA_ReadInputWordData 関数、
		4.6.10	HFA_ReadInputRandomBitData 関数、
		4.6.11	HFA_ReadInputRandomWordData 関数、
		4.6.12	HFA_ReadOutputData 関数、
		4.6.13	HFA_ReadOutputBitData 関数、
		4.6.14	HFA_ReadOutputWordData 関数、
		4.6.15	HFA_ReadOutputRandomBitData 関数、
		4.6.16	HFA_ReadOutputRandomWordData 関数、
		4.6.17	HFA_WriteOutputBitData 関数、
		4.2.18	HFA_WriteOutputWordData 関数、
		4.6.19	HFA_WriteOutputRandomBitData 関数、
		4.6.20	HFA_WriteOutputRandomWordData 関数を追加。
		4.7	コールバック関数に
		4.7.28	RecvReqWriteNetParam 関数、
		4.7.29	RecvReqControlEquipment 関数、
		4.7.30	LinkInSlave 関数、
		4.7.31	LinkOutSlave 関数、
		4.7.32	InputDataRefresh 関数、
		4.7.33	InputStatusRefresh 関数、
		4.7.34	ChangeTokenTimeMeasureStatus 関数、
		4.7.35	ChangeDataLogMeasureStatus 関数、
		4.7.36	RecvReqReadByteBlockFromSetingTool 関数、
		4.7.37	RecvReqReadWordBlockFromSetingTool 関数、
		4.7.38	RecvReqWriteByteBlockFromSetingTool 関数、
		4.7.39	RecvReqWriteWordBlockFromSetingTool 関数、
		4.7.40	RecvReqWriteNetParamFromSetingTool 関数、
		4.7.41	RecvReqControlEquipmentFromSetingTool 関数、
		4.7.42	RecvReqSetIOFromSetingTool 関数、
		4.7.43	RecvReqConfigParamFromSetingTool 関数、
		4.7.44	RecvReqResetNodeFromSetingTool 関数を追加。

(次頁に続く)

Rev.	改訂日	改訂箇所	改訂内容
1.0	2013/07/01	4.8	I/F 構造体に
		4.8.14	HFA_CALLBACKFUNC_V3 構造体、
		4.8.15	HFA_ADDRESS 構造体、
		4.8.16	HFA_SLAVE_INFO 構造体、
		4.8.17	HFA_SLAVES_INFO 構造体、
		4.8.18	HFA_SLAVE_STATUS 構造体、
		4.8.19	HFA_SLAVE_INPUT_STATUS 構造体、
		4.8.20	HFA_SLAVE_OUTPUT_STATUS 構造体、
		4.8.21	HFA_LOG_TOKENTIME 構造体、
		4.8.22	HFA_COMMON_MEMORY 構造体、
		4.8.23	HFA_CONFIG_PARAM 構造体、
		4.8.24	HFA_REPLY_DATA 構造体、
		4.8.25	HFA_LOG_V3 構造体、
		4.8.26	HFA_LOG_Cyclic_V3 構造体、
		4.8.27	HFA_LOG_Message_V3 構造体、
		4.8.28	HFA_LOG_Token_V3 構造体、
		4.8.29	HFA_IP 構造体、
		4.8.30	HFA_LOG_IP_V3 構造体を追加。
		6.1	WinSock ポート番号に 55004 を追加。
		6.2	DLL のデフォルト値にコンフィギュレーション用パラメータを追加。
		7	制限事項の通信ポート番号の範囲を変更。
		付録 1	メッセージ番号に Ver3.00 で追加された TCD を追加。
		付録 5	プロファイル情報のシステムパラメータ改変番号、変更日付、製品形名を変更。
		付録 6	ログ情報に Ver3.00 で追加されたログ情報を追加。
		付録 7	モニタモードに Ver3.00 で追加した I/F 関数を追加。
		全般	製造業者形式を”KE-SFL120”から”KE-SFL3WIN”に変更
1.1	2018/03/01	2.1	動作環境を変更。
		2.2	開発環境を変更。
1.2	2018/06/29	1	64bit 版の DLL を追加。
		2.1	実行時に必要なファイルを変更。
		2.2	開発時に必要なファイルを変更。
		4.1.1 4.1.2 4.1.3 4.3.1 4.3.2 4.3.3 4.3.4	LinkID 引数の型を long から LONG_PTR に変更。
		付録 5	64bit 版のプロファイル情報を追加。

関連文献

本仕様書は、以下に示す文献を引用します。FA リンクプロトコルに関する用語（略号）や仕様につきましては、以下の文献をご参照ください。

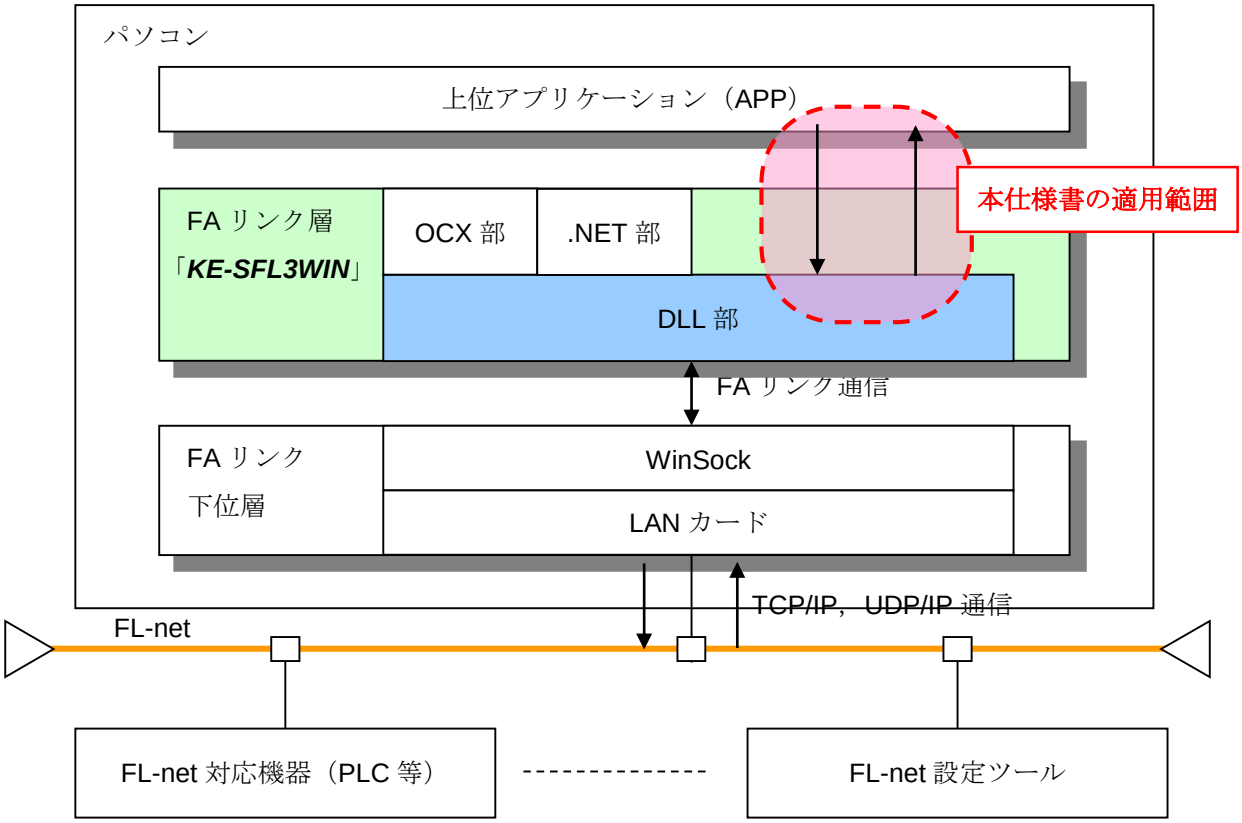
文献名	備　　考
FA コントロールネットワーク標準－プロトコル仕様	JISB3521 2004 年 02 月 20 日制定版
FA コントロールネットワーク標準－プロトコル仕様	日本電機工業会技術資料　JEM1479 2012 年 9 月 27 日改正（第 3 回）版
FA コントロールネットワーク［FL-net（OPCN-2）］－実装ガイドライン	日本電機工業会技術資料　JEM-TR213 2011 年 7 月 12 日改正（第 4 回）版
FA コントロールネットワーク［FL-net（OPCN-2）］－デバイスプロファイル共通仕様	日本電機工業会技術資料　JEM-TR214 2000 年 11 月 28 日制定版

目次

1. 適用範囲

本仕様書は、「FL-net Control For Windows®」（以降、「***KE-SFL3WIN***」と称します。）の DLL 部のインタフェース仕様を記述したものです。

「***KE-SFL3WIN***」は、FL-net（OPCN-2）ネットワーク内で適用される FA リンクプロトコルの Ver.3 版をサポートし、パソコン上で動作する通信エンジンです。「***KE-SFL3WIN***」の位置付けおよび本仕様書の適用範囲を下図に示します。



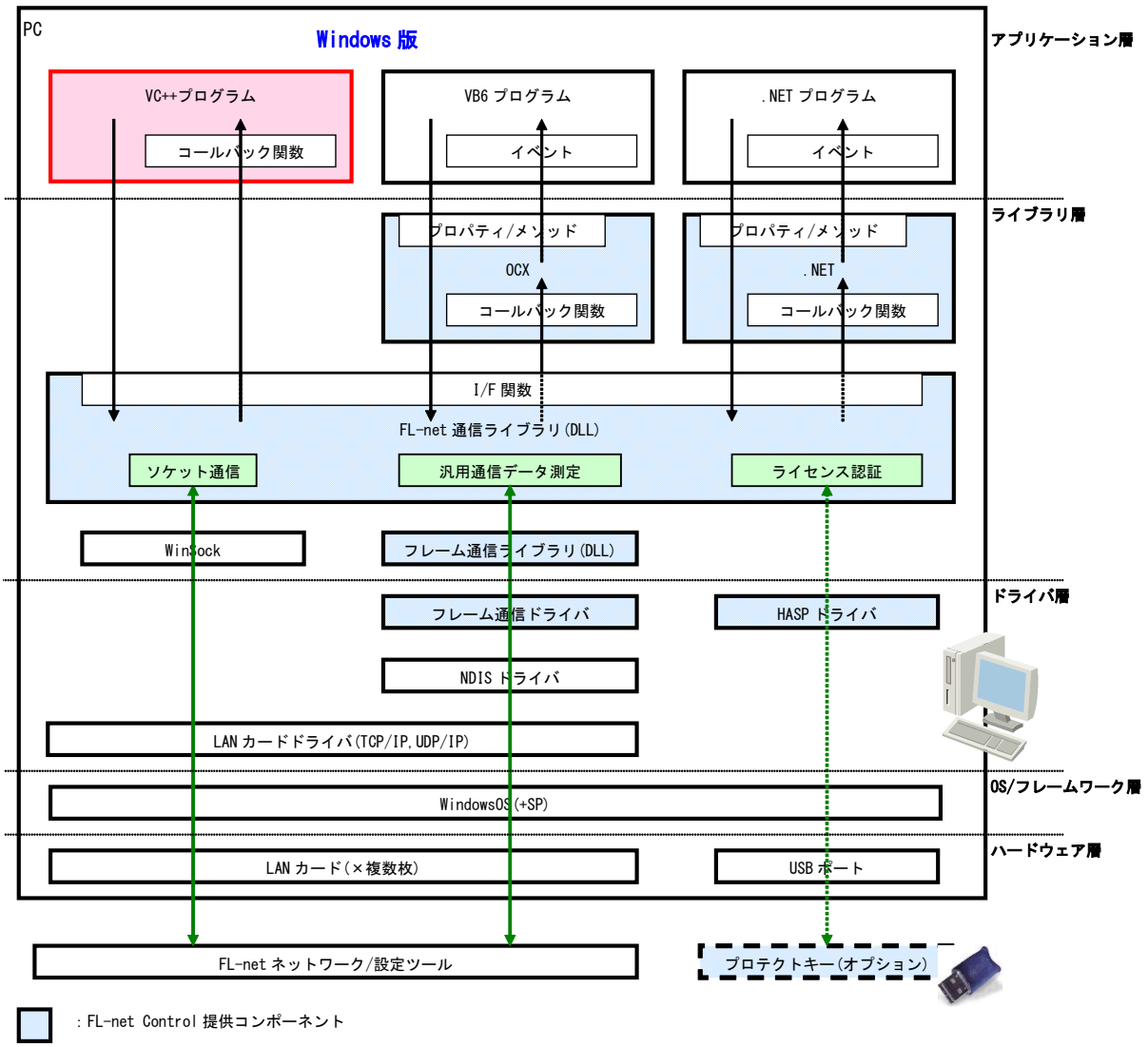
「***KE-SFL3WIN***」には 32bit 版と 64bit 版の DLL が存在します。アプリケーションのプラットフォームに応じて、以下の DLL を使用してください。

・32bit 版アプリケーションの場合：32bit 版の DLL(インストール先¥Bin¥x86¥FL_KE.dll)
・64bit 版アプリケーションの場合：64bit 版の DLL(インストール先¥Bin¥x64¥FL_KE.dll)

2. 動作環境

「***KE-SFL3WIN***」DLL 部の動作環境を以下に示します。

2.1. 実行環境



１） 動作環境

動作環境は弊社ホームページの FL-net Control For Windows®の動作環境をご確認ください。

２） ネットワーク設定

動作 OS 上で、以下に示すネットワーク設定を行う必要があります。（※設定手順については、本書では記述しません。OS のヘルプ等でご確認ください。）

① ネットワークサービスがインストールされていて、実行されていること。
② LAN カードを認識し、専用の LAN カードドライバがインストールされていること。
③ TCP/IP プロトコルがインストールされていて、LAN カードにバインドされていること。
④ TCP/IP のプロパティに IP アドレスが固定値で設定されていること。（複数の LAN カードを実装している場合は、それぞれ異なる IP アドレスを設定してください。）
⑤ フレーム通信ドライバがインストールされていること。詳細は別紙を参照してください。

３） プロテクトキードライバ設定(オプション)

プロテクトキーによる認証を行う場合、専用のドライバをインストールする必要があります。

４） イベントログ設定

実行中にエラー等を検知した場合、OS の機能であるイベントログにエラーの内容を出力します。イベントログの容量が小さいと、イベントログがオーバーフローしてしまう場合があります。イベントビューアにて、イベントログの容量を調整してください。（※手順については、OS のヘルプ等でご確認ください。）

実行時に必要なファイル

実行時には、以下のファイルが必要となります。

No.	項目	ファイル名	ファイル場所
1	DLL 本体ファイル	FL_KE.dll	上位アプリケーションから認識可能な場所。上位アプリケーションと同一フォルダ，OS のシステムフォルダ等。(※1)
2	イベントログ参照ファイル	FLnetMsg.dll	任意。ただし、レジストリ情報でイベントログ参照ファイル場所を特定すること。(※2)
3	フレーム通信ドライバ	FLpacket.dll	上位アプリケーションから認識可能な場所。上位アプリケーションと同一フォルダ，OS のシステムフォルダ等。(※1)
4	フレーム通信ログ参照ファイル	FLnetLog.dll	任意。ただし、レジストリ情報でイベントログ参照ファイル場所を特定すること。(※2)

※1 「***KE-SFL 3WIN***」インストール時は以下に配布されます。

32bit 版：インストール先¥Bin¥x86

64bit 版：インストール先¥Bin¥x64

※2 「***KE-SFL 3WIN***」インストール時は以下に配布されます。

32bit 版/64bit 版共通：インストール先¥Bin

5） 実行時に使用するレジストリ情報

実行時に、以下のレジストリ情報を使用します。レジストリ情報は、「***KE-SFL 3WIN***」インストール時に自動的に作成されます。

＜フォルダ＞

HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥EventLog¥Application¥FL_KE

＜項目＞

No.	内容	名前	種類	値
1	イベントログ参照ファイル場所	EventMessageFile	REG_SZ	「***KE-SFL 3WIN***」インストール先¥Bin¥FLnetMsg.dll
2	イベントログタイプ	TypesSupported	REG_DWORD	7

＜フォルダ＞

HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥Services¥EventLog¥Application¥FLpacketDLL

＜項目＞

No.	内容	名前	種類	値
1	イベントログ参照ファイル場所	EventMessageFile	REG_SZ	「***KE-SFL 3WIN***」インストール先¥Bin¥FLnetLog.dll
2	イベントログタイプ	TypesSupported	REG_DWORD	7

2.2. 開発環境

上位アプリケーション開発時の環境を以下に示します。

1） 対応開発コンテナ

対応開発コンテナは弊社ホームページの FL-net Control For Windows®の動作環境をご確認ください。

2） 開発時に必要なファイル

開発時には、以下のファイルが必要となります。

No.	項目	ファイル名	ファイル場所
1	DLL 本体ファイル	FL_KE.dll	上位アプリケーションから認識可能な場所。上位アプリケーションと同一フォルダ，OS のシステムフォルダ等。(※1)
2	イベントログ参照ファイル	FLnetMsg.dll	任意。ただし、レジストリ情報でイベントログ参照ファイル場所を特定すること。(※2)
3	DLL 定義ファイル	FL_KE.h	上位アプリケーションから認識可能な場所(※3)
4	リンク用ライブラリファイル	FL_KE.lib	上位アプリケーションから認識可能な場所(※4)
5	フレーム通信ドライバ	FLpacket.dll	上位アプリケーションから認識可能な場所。上位アプリケーションと同一フォルダ，OS のシステムフォルダ等。(※1)
6	フレーム通信イベントログ参照ファイル	FLnetLog.dll	任意。ただし、レジストリ情報でイベントログ参照ファイル場所を特定すること。(※2)

※1 「***KE-SFL 3WIN***」インストール時は以下に配布されます。

32bit 版：インストール先¥Bin¥x86

64bit 版：インストール先¥Bin¥x64

※2 「***KE-SFL 3WIN***」インストール時は以下に配布されます。

32bit 版/64bit 版共通：インストール先¥Bin

※3 「***KE-SFL 3WIN***」インストール時は以下に配布されます。

32bit 版/64bit 版共通：インストール先¥Env

※4 「***KE-SFL 3WIN***」インストール時は以下に配布されます。

32bit 版：インストール先¥Env¥x86

64bit 版：インストール先¥Env¥x64

3. 機能

3.1. 機能一覧

「***KE-SFL3WIN***」が実装する FL-net 機能の一覧を以下に示します。

No.	機能	概要
1	参加・離脱	FL-net ネットワークに参加・離脱します。
2	ネットワーク管理	ネットワーク情報を管理します。
3	コモンメモリ管理	コモンメモリデータを管理します。
4	メッセージ送信	他ノードあてにメッセージを送信します。
5	メッセージ受信	自ノードあてのメッセージを受信します。
6	コマンドサーバ	設定ツールに対応したコマンドサーバを起動・停止し、汎用コマンドの送受信を行います（FL-net Ver.3 機能）。
7	負荷測定	FL-net ネットワークの負荷を測定します（FL-net Ver.3 機能）。
8	デバイスレベルネットワーク	任意マスタノードの設定を行うことで、マスタ・スレーブ方式によるデバイスレベルでの IO データ交換を行います（FL-net Ver.3 機能）。
9	モニタモード	FL-net ネットワークに参加せずに、ネットワークの情報を監視します。 詳細は、付録 7 をご参照ください。

3.2. 参加・離脱

1） 自ノード参加

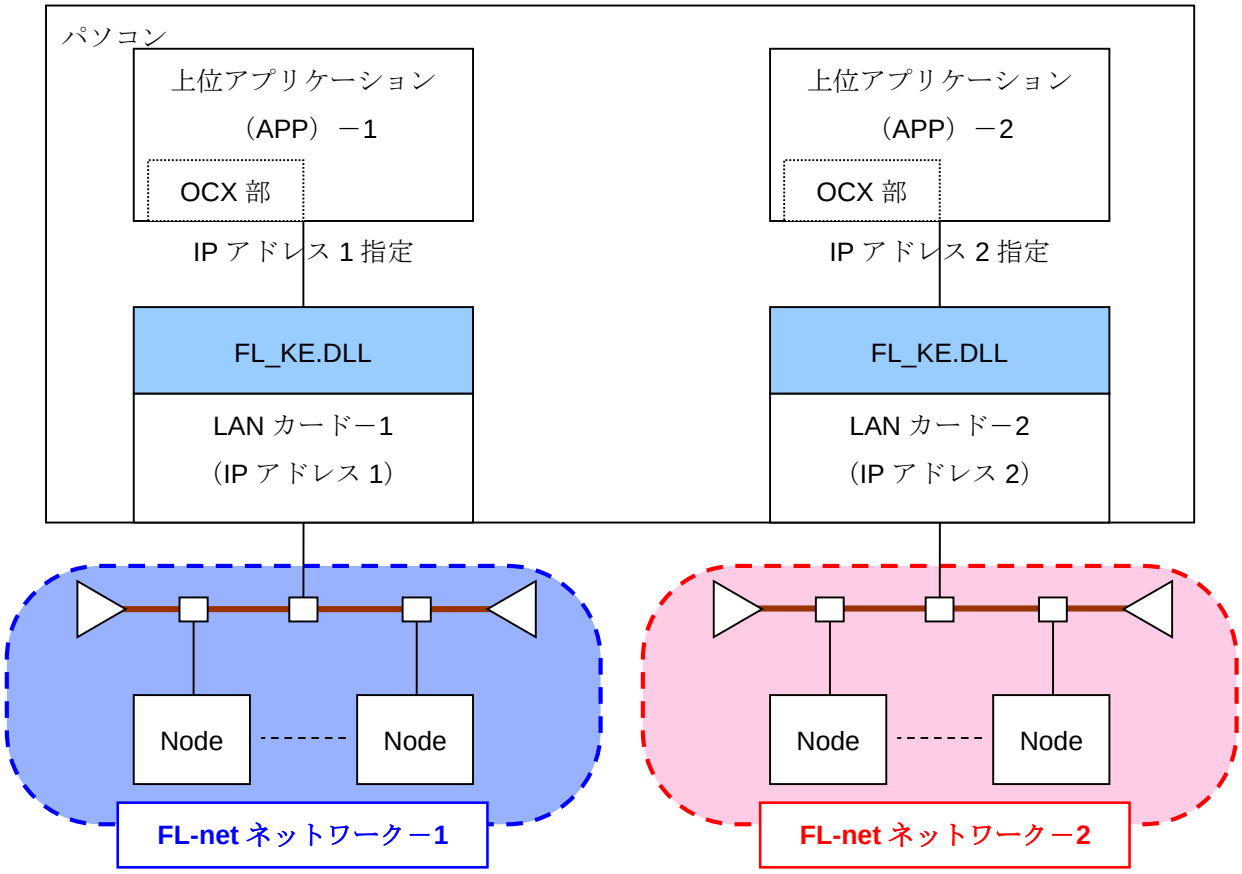
HFA_LinkIn 関数をコールすることで、FL-net ネットワークへの参加を開始します。ネットワーク参加時のパラメータを以下に示します。

No.	項目	概要
1	ノード番号	HFA_LinkIn 関数の NodeNo 引数で指定します。 FL-net ネットワーク上で重複しない番号を指定してください。
2	トークン監視時間	HFA_LinkIn 関数の TokenTimer 引数で指定します。 コモンメモリ送信領域および最小許容フレーム間隔の設定に応じて、値を調整する必要があります。
3	最小許容フレーム間隔	HFA_LinkIn 関数の MinFrameInterval 引数で指定します。 通常は、0 を指定します。FL-net ネットワーク上に通信性能が遅い機器が存在する場合は、値を調整する必要があります。
4	コモンメモリ送信領域	HFA_SetCommon 関数で指定します。送信領域を設定する必要がある場合は、HFA_LinkIn 関数より前にコールしてください。
5	使用する LAN カード	HFA_LinkIn 関数の LocalIP 引数で指定します。LAN カードが複数枚存在する場合に、LAN カードを特定することができます。

ネットワークに参加完了した時点で、上位アプリケーションに自ノード参加イベントを通知します。ただし、ネットワーク状態（ネットワーク上に他ノードが立ち上がっていない場合）によっては、ネットワーク参加完了まで時間が掛かる場合があります。

ネットワーク参加時にノード番号の重複やトークンモード不一致を検知した場合は、ネットワークへの参加が失敗します。

同一パソコン内に LAN カードが複数枚存在する場合は、同時にそれぞれの LAN カードから FL-net ネットワークに参加することが可能です（最大 2 枚まで動作保証）。複数 FL-net ネットワークに同時参加する場合は、上位アプリケーションをそれぞれ個別に実行し、HFA_LinkIn 関数の LocalIP 引数で、使用する LAN カードの IP アドレスをそれぞれ個別に指定してください。



2） 自ノード離脱

HFA_LinkOut 関数をコールすることで、FL-net ネットワークからの離脱を開始します。なお、ネットワークの状態によって、自ノードがネットワークから離脱してしまう場合もあります。

自ノードがネットワークから離脱した時点で、上位アプリケーションに自ノード離脱イベントを通知します。

3） 他ノード参加・離脱

自ノードが FL-net ネットワーク参加中に他ノードの参加・離脱を検知した場合は、上位アプリケーションに他ノード参加・離脱イベントを通知します。

4） FL-net 通信エラー

自ノードが FL-net ネットワーク参加中に通信エラーを検知した場合は、上位アプリケーションにエラーイベントを通知します。通知されるエラー番号から、エラー内容を確認することができます。

3.3. ネットワーク管理

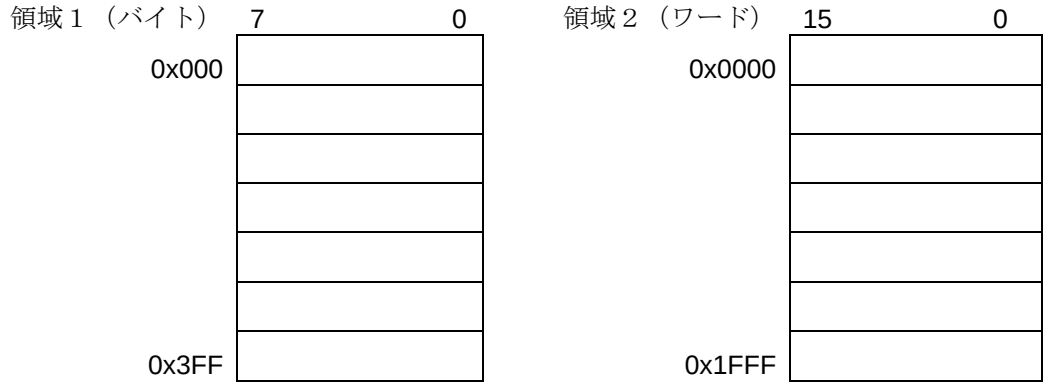
「***KE-SFL3WIN***」で管理する情報のうち以下の項目については、上位アプリケーションで読出しまたは設定が可能です。

No.	項目	概要
1	自ノード管理情報パラメータ	HFA_GetNodeStatus 関数で自ノードの状態情報を読出し可能です。読出し可能項目は、関数の I/F 仕様をご参照ください。
2	参加ノード管理情報パラメータ	HFA_GetNodeStatus 関数で参加ノードの状態情報を読出し可能です。読出し可能項目は、関数の I/F 仕様をご参照ください。
3	ネットワーク管理情報パラメータ	HFA_GetNetworkStatus 関数でネットワークの状態を読出し可能です。読出し可能項目は、関数の I/F 仕様をご参照ください。
4	ノード状態	ノード状態には、上位層の状態、FA リンクの状態、自ノードの状態があります。各状態に含まれる項目、構成および値は、6.3 節をご参照ください。 HFA_GetNodeStatus 関数でノード状態を読出し可能です。 HFA_SetControlEquipment 関数で、運転／停止状態の変更が可能です。
5	コンフィギュレーション用パラメータ	HFA_SetConfigParam 関数で設定が可能です。 HFA_GetConfigParam 関数で取得が可能です。
6	ノード名称	自ノードの名称を HFA_SetNodeName 関数で設定が可能です。 HFA_GetNodeName 関数で取得が可能です。
7	ノード番号	HFA_SetNodeNo 関数で設定が可能です。 HFA_GetNodeNo 関数で取得が可能です。
8	トークン監視時間	HFA_SetTokenWatchTime 関数で設定が可能です。 HFA_GetTokenWatchTime 関数で取得が可能です。
9	最小許容フレーム間隔	HFA_SetMinFrameInterval 関数で設定が可能です。 HFA_GetMinFrameInterval 関数で取得が可能です。
10	IP アドレス	HFA_SetIP 関数で設定が可能です。 HFA_GetIP 関数で取得が可能です。
11	ログ情報	HFA_GetMyNodeLog 関数で読出しが可能です。 自ノードが管理するログ情報の項目は、付録 6 をご参照ください。 HFA_ClearMyNodeLog 関数でクリアが可能です。
12	ログ情報(Ver3.0 版)	HFA_GetMyNodeLogV3 関数で読出しが可能です。 自ノードが管理するログ情報の項目は、付録 6 をご参照ください。

3.4. コモンメモリ管理

1）メモリマップ

コモンメモリデータは、「***KE-SFL3WIN***」内部のメモリで管理します。コモンメモリデータのメモリマップを以下に示します。



領域	単位	範囲	先頭アドレス	データサイズ
領域１	バイト	0x000～0x3FF	0x000	0x400
領域２	ワード	0x0000～0x1FFF	0x0000	0x2000

コモンメモリの送信領域を割り付ける場合は、HFA_SetCommon 関数をコールします。

ネットワーク参加時に他ノード割付との重複を検知した場合、コモンメモリの送信領域はゼロになります。

2）メモリアクセス

コモンメモリデータの値を読出す場合は、HFA_ReadCommon1 関数または HFA_ReadCommon2 関数をコールします。FL-net の参加状態に関わらず、読出しは可能です。

コモンメモリデータの値を書込む場合は、HFA_WriteCommon1 関数または HFA_WriteCommon2 関数をコールします。書込み可能な範囲は、送信領域に限られます。FL-net の参加状態に関わらず、書込みは可能です。

3）値の有効性

コモンメモリの値は、自ノードの状態および割付状況に応じて、以下の通りとなります。

<table>
<tr><td rowspan="2">自ノード状態</td><td colspan="3">割付状況</td></tr>
<tr><td>送信領域</td><td>他ノード送信領域</td><td>未使用領域</td></tr>
<tr><td>アプリケーション起動時</td><td colspan="3">オールゼロ</td></tr>
<tr><td>参加前／離脱</td><td rowspan="2">アプリケーションが書込みを行った値を保持</td><td>自ノードが最後にネットワークに参加していた時点の値を保持</td><td rowspan="3">前回の値を保持</td></tr>
<tr><td>参加中</td><td rowspan="2">他ノードから受信したサイクリックデータ。
※対象ノードが離脱した場合は、離脱した時点の値を保持</td></tr>
<tr><td>モニタモード</td><td>－（領域がゼロになる）</td></tr>
</table>

4） 変化検出機能

コモンメモリ変化検出領域内でコモンメモリの値に変化が発生した場合に、アプリケーションに変更イベントを通知します。コモンメモリ変化検出領域は、HFA_SetCommonRefreshDegree 関数で設定することが可能です。

3.5. メッセージ伝送

3.5.1. 実装メッセージ一覧

「***KE-SFL3WIN***」が実装するメッセージの一覧を以下に示します。

メッセージ種類	サーバ機能※1		クライアント機能※1	
	1 対 1	1 対 n	1 対 1	1 対 n
バイトブロック※2 リード	△	―	○	―
バイトブロック※2 ライト	△	―	○	―
ワードブロック※2 リード	△	―	○	―
ワードブロック※2 ライト	△	―	○	―
ネットワークパラメータリード	◎	―	○	―
ネットワークパラメータライト	△	―	○	―
停止指令	△	―	○	―
運転指令	△	―	○	―
プロファイルリード	◎	―	○	―
ログデータリード	◎	―	○	―
ログデータクリア	◎	◎	○	○
メッセージ折り返し	◎	―	○	―
ベンダ固有メッセージ	△	○	○	○
透過形メッセージ	○	○	○	○

※1.表内記号の意味は以下の通りです。

・ ○：実装

・ ◎：実装　（「***KE-SFL3WIN***」で正常応答を送信します。）

・ △：実装　（上位アプリケーションで応答する必要があります。）

・ ×：非実装（「***KE-SFL3WIN***」で非実装応答を送信します。）

・ ―：なし　（プロトコル仕様で定義されていません。）

※2.バイトブロックおよびワードブロックの実データは、「***KE-SFL3WIN***」DLL 部では管理しません。ブロックデータを使用する場合は、上位アプリケーションにて実データを管理する必要があります。

3.5.2. メッセージ送信

上位アプリケーションから送信可能なメッセージの一覧を以下に示します。

送信メッセージ種類	要求メッセージ		応答メッセージ	
	1 対 1	1 対 n	1 対 1	1 対 n
バイトブロックリード	○	－	○	－
バイトブロックライト	○	－	○	－
ワードブロックリード	○	－	○	－
ワードブロックライト	○	－	○	－
ネットワークパラメータリード	○	－	×	－
ネットワークパラメータライト	○	－	○	－
停止指令	○	－	○	－
運転指令	○	－	○	－
プロファイルリード	○	－	×	－
ログデータリード	○	－	×	－
ログデータクリア	○	○	×	－
メッセージ折り返し	○	－	×	－
ベンダ固有メッセージ	○	○	○	－
透過形メッセージ	○	○		

※表内記号の意味は以下の通りです。

・ ○：関数コールで送信可能です。

・ ×：なし。（「***KE-SFL3WIN***」で送信します。）

・ －：なし。（プロトコル仕様で定義されていません。）

メッセージを送信する場合は、メッセージ種類に対応した関数をコールします。詳細は、4.2 節をご参照ください。

3.5.3. メッセージ受信

上位アプリケーションで受信可能なメッセージの一覧を以下に示します。

受信メッセージ種類	要求メッセージ		応答メッセージ	
	1 対 1	1 対 n	1 対 1	1 対 n
バイトブロックリード	◎	－	○	－
バイトブロックライト	◎	－	○	－
ワードブロックリード	◎	－	○	－
ワードブロックライト	◎	－	○	－
ネットワークパラメータリード	×	－	○	－
ネットワークパラメータライト	◎	－	○	－
停止指令	◎	－	○	－
運転指令	◎	－	○	－
プロファイルリード	×	－	○	－
ログデータリード	×	－	○	－
ログデータクリア	×	×	○	－
メッセージ折り返し	×	－	○	－
ベンダ固有メッセージ	◎	○	○	－
透過形メッセージ	○	○		

※表内記号の意味は以下の通りです。

・ ◎：受信イベントを通知します。受信イベントを元に、上位アプリケーションで応答メッセージを送信する必要があります。詳細は、4.7 節をご参照ください。

・ ○：受信イベントを通知します。詳細は、4.7 節をご参照ください。

・ ×：なし。（「***KE-SFL3WIN***」で自動応答します。）詳細は、6.4 節をご参照ください。

・ －：なし。（プロトコル仕様で定義されていません。）

3.6. コマンドサーバ

1）コマンドサーバの起動・停止

コマンドサーバを起動することで、設定ツールより各種パラメータの設定または取得を行うことが可能です。TCP/IP および UDP/IP の 2 種類のプロトコルをサポートします。以下の関数で起動・停止が可能です。

No.	項目	概要
1	UDP/IP プロトコル	HFA_StartCmdServerUdp 関数で UDP/IP プロトコルのコマンドサーバを起動します。 HFA_StopCmdServerUdp 関数で停止します。 HFA_GetCmdServerUdpStatus 関数で、起動状態を取得します。
2	TCP/IP プロトコル	HFA_StartCmdServerTcp 関数で TCP/IP プロトコルのコマンドサーバを起動します。 HFA_StopCmdServerTcp 関数で停止します。 HFA_GetCmdServerTcpStatus 関数で、起動状態を取得します。

2）コマンドサーバ実装メッセージ一覧

「***KE-SFL3WIN***」が実装するコマンドサーバメッセージの一覧を以下に示します。

メッセージ種類	サーバ機能※1
	1 対 1
バイトブロック※2 リード	△
バイトブロック※2 ライト	△
ワードブロック※2 リード	△
ワードブロック※2 ライト	△
ネットワークパラメータリード	◎
ネットワークパラメータライト	△
停止指令	△
運転指令	△
プロファイルリード	◎
ログデータリード	◎
ログデータクリア	◎
メッセージ折り返し	◎
IO 割付設定	△
IO 割付読出し	◎
トークン保持時間測定開始	◎
トークン保持時間測定終了	◎
汎用通信データ送信元ログ測定開始	◎
汎用通信データ送信元ログ測定終了	◎
コンフィギュレーション用パラメータ設定	△
参加ノード管理情報パラメータ読出し	◎
自ノード設定情報パラメータ読出し	◎
自ノード管理情報パラメータ読出し	◎
ノードリセット	△

※1.表内記号の意味は以下の通りです。

・ ◎：実装（「***KE-SFL3WIN***」で正常応答を送信します。）

・ △：実装（受信イベントを通知します。受信イベントを元に、上位アプリケーションで応答する必要があります。詳細は、4.7 節をご参照ください。）

※2.バイトブロックおよびワードブロックの実データは、「***KE-SFL3WIN***」DLL 部では管理しません。ブロックデータを使用する場合は、上位アプリケーションにて実データを管理する必要があります。

3.7. 負荷測定

以下の関数でトークン保持時間の測定および汎用通信データ送信元ログの測定が可能です。

No.	項目	概要
1	トークン保持時間の測定	HFA_StartTokenTimeMeasure 関数で測定を開始します。 HFA_StopTokenTimeMeasure 関数で測定を停止します。 HFA_GetTokenTimeMeasureStatus 関数で測定状態を取得します。
2	汎用通信データ送信元ログの測定	HFA_StartDataLogMeasure 関数で測定を開始します。 HFA_StopDataLogMeasure 関数で測定を停止します。 HFA_GetDataLogMeasureStatus 関数で測定状態を取得します。

3.8. デバイスレベルネットワーク

任意マスタノードの設定を行うことで、マスタ・スレーブ方式によるデバイスレベルでの IO データ交換を行います。

1）上位アプリケーションからの任意マスタノード設定

制御スレーブ個数に 1 以上の値を指定して HFA_SetIO 関数をコールすることで、任意マスタノードの設定を行います。なお、制御スレーブ個数に 0 を指定した場合は、任意マスタ機能が無しになります。デフォルトは任意マスタ機能無しです。

HFA_SetIO 関数では、制御スレーブ単位に IO 割付情報を設定しますが、IO 出力データ領域および出力ステータス領域を、コモンメモリ送信領域内に指定する必要があります。コモンメモリ送信領域は、HFA_SetCommon 関数で設定可能です。また、制御スレーブ単位の IO 割付情報が重複しないように設定してください。

2）設定ツールからの任意マスタノード設定

コマンドサーバを起動しておき、設定ツールから IO 割付設定を行うと、*RecvReqSetIOFromSettingTool* イベントを通知します。上位アプリケーションが正常応答を返すことで、任意マスタノードの設定を更新できます。

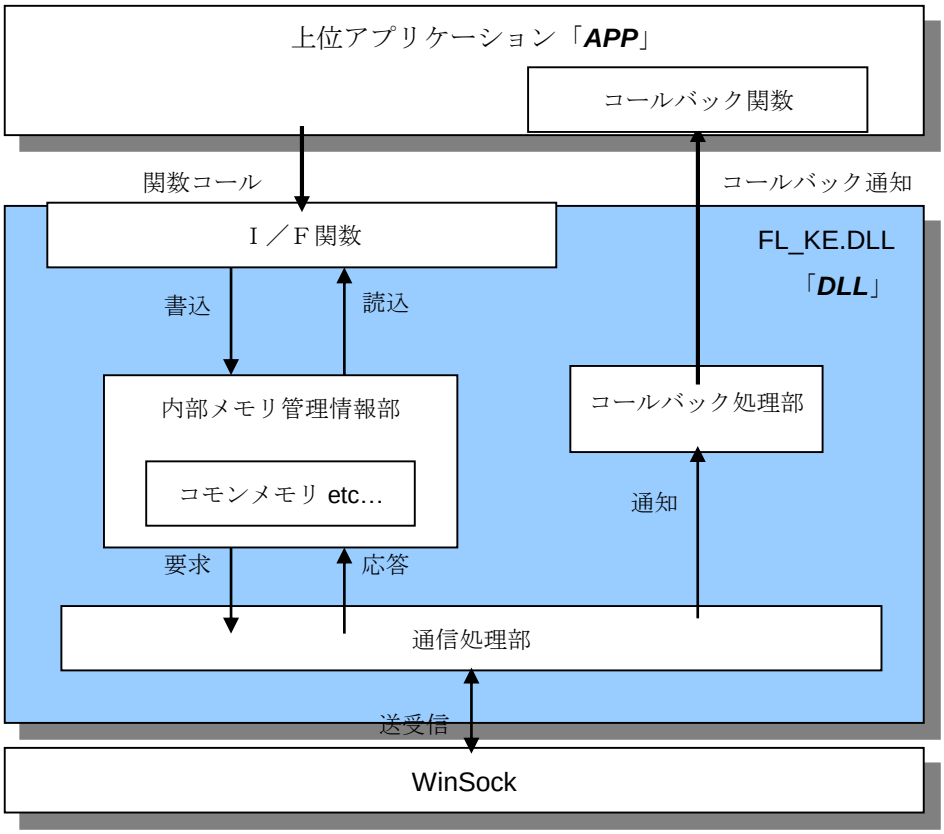
3）パラメータ

「***KE-SFL3WIN***」で管理する以下のパラメータを、上位アプリケーションで取得または設定が可能です。

No.	パラメータ	概要
1	IO 割付情報	HFA_SetIO 関数で設定が可能です。 HFA_GetIO 関数で取得が可能です。
2	入力ステータス	HFA_GetInputStatus 関数で取得が可能です。 取得可能項目は、関数の I/F 仕様をご参照ください。
3	出力ステータス	HFA_SetOutputStatus 関数で設定が可能です。 HFA_GetOutputStatus 関数で取得が可能です。 設定・取得可能項目は、関数の I/F 仕様をご参照ください。
4	入力データ	以下の関数で読出しが可能です。 ・HFA_ReadInputData 関数　：全データ読出し ・HFA_ReadInputBitData 関数　：連続ビット読出し ・HFA_ReadInputWordData 関数　：連続ワード読出し ・HFA_ReadInputRandomBitData 関数　：ランダムビット読出し ・HFA_ReadInputRandomWordData 関数　：ランダムワード読出し
5	出力データ	以下の関数で読出しが可能です。 ・HFA_ReadOutputData 関数　：全データ読出し ・HFA_ReadOutputBitData 関数　：連続ビット読出し ・HFA_ReadOutputWordData 関数　：連続ワード読出し ・HFA_ReadOutputRandomBitData 関数　：ランダムビット読出し ・HFA_ReadOutputRandomWordData 関数　：ランダムワード読出し 以下の関数で書込みが可能です。 ・HFA_WriteOutputBitData 関数　：連続ビット書込み ・HFA_WriteOutputWordData 関数　：連続ワード書込み ・HFA_WriteOutputRandomBitData 関数　：ランダムビット書込み ・HFA_WriteOutputRandomWordData 関数　：ランダムワード書込み
6	スレーブノード参加状態	HFA_GetSlaveNodeLinkStatus 関数で取得可能です。

4. I/F 仕様

FL_KE.DLL（以降、「***DLL***」と称します。）および上位アプリケーション（以降、「***APP***」と称します。）のインタフェース仕様を下図に示します。



「***APP***」が「***DLL***」に機能を要求する場合は、「***DLL***」の提供する I/F 関数をコールします。I/F 関数は、機能に応じて次の 6 種類に分類されます。なお、I/F 関数の呼び出し規約は、stdcall 呼び出しとします。

１） メモリアクセス関数
２） メッセージ送信関数
３） DLL 制御関数
４） コマンドサーバ関数
５） 負荷測定関数
６） デバイスレベルネットワーク関数

「***DLL***」が「***APP***」に非同期でイベントを通知する場合は、「***DLL***」が「***APP***」内に用意されたコールバック関数をコールします。非同期イベントの通知を有効にするためには、「***APP***」側でコールバック関数を実装し、関数のアドレスを「***DLL***」に登録する必要があります。

4.1. メモリアクセス関数

メモリアクセス関数とは、「***DLL***」内部で管理している情報にアクセスするための I/F 関数です。メモリアクセス関数の一覧を以下に示します。

No.	関数名	概要
1	HFA_LinkIn	FL-net ネットワークへの参加（モニタモードの開始）
2	HFA_LinkInDefault	FL-net ネットワークへの参加簡易版
3	HFA_LinkOut	FL-net ネットワークからの離脱（モニタモードの終了）
4	HFA_WriteCommon1	コモンメモリ（領域 1）書込み
5	HFA_WriteCommon2	コモンメモリ（領域 2）書込み
6	HFA_ReadCommon1	コモンメモリ（領域 1）読出し
7	HFA_ReadCommon2	コモンメモリ（領域 2）読出し
8	HFA_GetNodeStatus	ノード管理情報パラメータ読出し
9	HFA_GetNetworkStatus	ネットワークステータス読出し
10	HFA_GetMyNodeLog	自ノードログ情報読出し（FL-net Ver.2 専用）
11	HFA_GetMyNodeLogV3	自ノードログ情報読出し（FL-net Ver.3 対応版）
12	HFA_ClearMyNodeLog	自ノードログ情報クリア
13	HFA_SetCommon	コモンメモリ送信領域割り当て設定
14	HFA_GetCommon	コモンメモリ送信領域割り当て取得
15	HFA_SetNodeName	ノード名（設備名）設定
16	HFA_GetNodeName	ノード名（設備名）取得
17	HFA_SetNodeNo	ノード番号設定
18	HFA_GetNodeNo	ノード番号取得
19	HFA_SetTokenWatchTime	トークン監視時間設定
20	HFA_GetTokenWatchTime	トークン監視時間取得
21	HFA_SetMinFrameInterval	最小許容フレーム間隔設定
22	HFA_GetMinFrameInterval	最小許容フレーム間隔取得
23	HFA_SetIP	IP アドレス設定
24	HFA_GetIP	IP アドレス取得
25	HFA_SetConfigParam	コンフィギュレーション用パラメータ設定
26	HFA_GetConfigParam	コンフィギュレーション用パラメータ取得
27	HFA_SetCommonRefreshDegree	コモンメモリ更新イベントの検知範囲設定
28	HFA_SetControlEquipment	運転／停止状態設定

4.1.1. HFA_LinkIn

FL-net ネットワークへの参加（モニタモードの開始）

＜関数 I/F＞

long HFA_LinkIn(LONG_PTR LinkID, long *NodeNo, long *TokenTimer, long *MinFrameInterval,char *LocalIP)

＜引数＞

型	変 数	I/O	内 容
LONG_PTR	LinkID	IN	「***DLL***」内部メモリをアクセスするための識別子。 HFA_AttachLink 関数で取得した値を指定してください。
long *	NodeNo	IN/ OUT	FL-net ネットワークに参加する自ノード番号。 ・1～254 を指定した場合は、参加モードとして FL-net ネットワークに参加します。 ・0 を指定した場合は、モニタモード（付録 7 参照）を開始します。 既にネットワークへ参加中（参加待機中）の状態に本関数がコールされた場合は、入力値は無視され、現在の自ノード番号が格納されます。
long *	TokenTimer	IN/ OUT	トークン監視時間（ms 単位）。 既にネットワークへ参加中（参加待機中）の状態に本関数がコールされた場合は、入力値は無視され、現在値が格納されます。
long *	MinFrameInterval	IN/ OUT	最小許容フレーム間隔（100μs 単位）。 既にネットワークへ参加中（参加待機中）の状態に本関数がコールされた場合は、入力値は無視され、現在値が格納されます。
char *	LocalIP	IN/ OUT	ローカル IP アドレス。 複数の LAN カードを実装している場合に、特定の LAN カードを指定することが可能です。 ・LAN カードを特定しない場合は、””（空文字）を指定します。この場合、OS が管理するデフォルトの LAN カードが選択されます。 ・LAN カードを特定する場合は、10 進数のゼロサプレスで”XXX.XXX.XXX.XXX”形式で指定します。 本関数が正常終了した場合、使用カードの IP アドレスが格納されます。 **注）呼び出し元「*APP*」では、必ず 20 バイト以上の領域を確保してください。**

＜引数範囲＞

引 数	参加モード	モニタモード	備 考
NodeNo	1≦（値）≦254	0	
TokenTimer	1≦（値）≦255	－（値は無効）	ms 単位
MinFrameInterval	0≦（値）≦50	－（値は無効）	100μs 単位

＜詳細＞

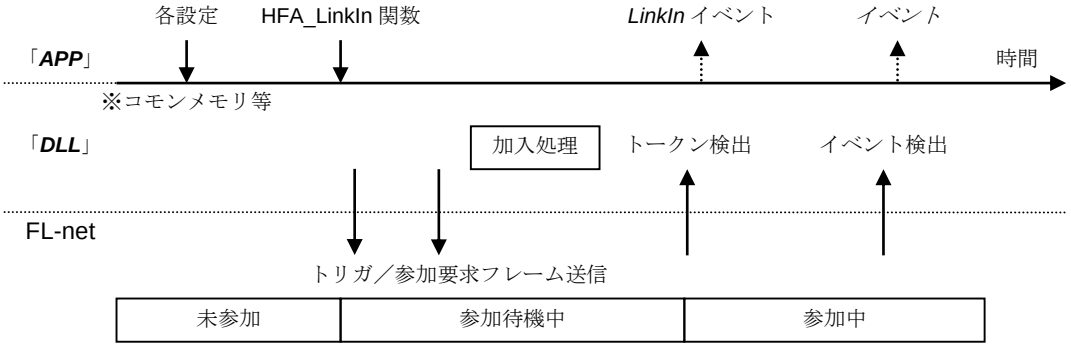
引数で指定された条件で FL-net ネットワークへ参加します。本関数が正常終了するとネットワークへの参加待機中となり、参加完了時にノード参加（*LinkIn*）イベントが発生します。

なお、一定時間（HFA_SetTimeout の LinkIn 引数で設定されたタイムアウト値）を経過してもネットワークに参加できない場合は、タイムアウト（*LinkInTimeout*）イベントが発生します。ただし、本関数をコールした時点におけるネットワーク参加状態に応じて、引数のチェック範囲およびイベントの発生状況が異なります。

ネットワーク参加後に設定不可の項目については、ネットワーク参加前に各種設定を行う必要があります。コモンメモリのアドレスとサイズのデフォルト値は 0（受信専用）です。コモンメモリの割付を行う場合は、HFA_SetCommon 関数にて設定する必要があります。

モニタモードを開始する場合は、NodeNo 引数に 0 を指定します。モニタモードの機能については、付録 7 をご参照ください。

ネットワーク参加状態	引数チェック	*LinkIn* イベント	*LinkInTimeout* イベント
未参加	○	○	○
既に参加待機中	×（既に設定された値を継続する）	○	○（既に設定されたタイマが発生する）
既に参加中	×（既に設定された値を継続する）	×（発生しない）	×（発生しない）



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	既に参加中(モニタ中)
7	既に参加待機中
11	IO 割付設定異常
-10	セキュリティエラー。以下の原因が考えられます。 ・プロテクトキーが正しくセットされていない ・ライセンス認証データが登録されていない ・試用版の制限時間を超過している
-100	システムエラー。以下の原因が考えられます。 ・メモリ不足 ・ネットワークポートが既に使用中（IP ポート：55000～55004） ・TCP/IP のプロパティが正しく設定されていない

＜使用例＞

１）ノード番号=1, トークン監視時間=255, 最小許容フレーム間隔=0, LAN カード=指定しないの条件で、FL-net に参加する。

```
void SampleLinkIn1( ){
    long lNodeNo=1, lTW=255, lMFT=0, lRet;                    // 参加条件を設定
    char cIp[20];
    memset(cIp, 0, sizeof(cIp));                              // LAN カード指定しない(=NULL 指定)
    lRet = HFA_LinkIn(G_LinkID, &lNodeNo, &lTW, &lMFT, cIp);  // FL-net ネットワークに参加
}
```

２）”192.168.250.250”の LAN カードを指定して、モニタモードを実行する。

```
void SampleLinkIn2( ){
    long lNodeNo=0, lTW, lMFT, lRet;                          //ノード番号=0（モニタモード）
    char cIp[20];
    strcpy(cIp, “192.168.250.250”);                           // LAN カード指定
    lRet = HFA_LinkIn(G_LinkID, &lNodeNo, &lTW, &lMFT, cIp);  // モニタモードを開始
}
```

＜関連事項＞

・ HFA_AttachLink 関数，HFA_LinkOut 関数，HFA_SetTimeout 関数，HFA_SetCommon 関数

・ *LinkIn* イベント，*LinkInTimeout* イベント

4.1.2. HFA_LinkInDefault

FL-net ネットワークへの参加（モニタモードの開始）の引数省略版

＜関数 I/F＞

long HFA_LinkInDefault(LONG_PTR LinkID)

＜引数＞

型	変 数	I/O	内 容
LONG_PTR	LinkID	IN	「***DLL***」内部メモリをアクセスするための識別子。 HFA_AttachLink 関数で取得した値を指定してください。

＜詳細＞

HFA_LinkIn 関数の引数省略版です。現在設定されている設定値で FL-net ネットワークへ参加します。設定値を変更する場合は、以下の関数で設定します。

・自ノード番号･･･HFA_SetNodeNo 関数
・トークン監視時間･･･HFA_SetTokenWatchTime 関数
・最小許容フレーム間隔･･･HFA_SetMinFrameInterval 関数
・ローカル IP アドレス･･･HFA_SetIP 関数
・設定値一括変更･･･HFA_SetConfigParam 関数

＜戻り値＞

値	内 容
0	正常終了
4	既に参加中
7	既に参加待機中
11	IO 割付設定異常
-1	引数異常
-10	セキュリティエラー。以下の原因が考えられます。 ・プロテクトキーが正しくセットされていない ・ライセンス認証データが登録されていない ・試用版の制限時間を超過している
-100	システムエラー。以下の原因が考えられます。 ・メモリ不足 ・ネットワークポートが既に使用中(IP ポート：55000～55004) ・TCP/IP のプロパティが正しく設定されていない

＜使用例＞

１）FL-net に参加する。

```
void SampleLinkInDefault(){
      long lRet;
      HFA_SetNodeNo(10);                    // ノード番号を 10 に設定
      lRet = HFA_LinkInDefault(G_LinkID);   // FL-net ネットワークに参加
}
```

＜関連事項＞

・HFA_AttachLink 関数、HFA_LinkOut 関数、HFA_SetTimeout 関数、HFA_SetCommon 関数

・HFA_SetNodeNo 関数、HFA_SetTokenWatchTime 関数、HFA_SetMinFrameInterval 関数

・HFA_SetIP 関数、HFA_SetConfigParam 関数

・*LinkIn* イベント、*LinkInTimeout* イベント

4.1.3. HFA_LinkOut

FL-net ネットワークからの離脱（モニタモードの終了）

＜関数 I/F＞

long HFA_LinkOut(LONG_PTR LinkID)

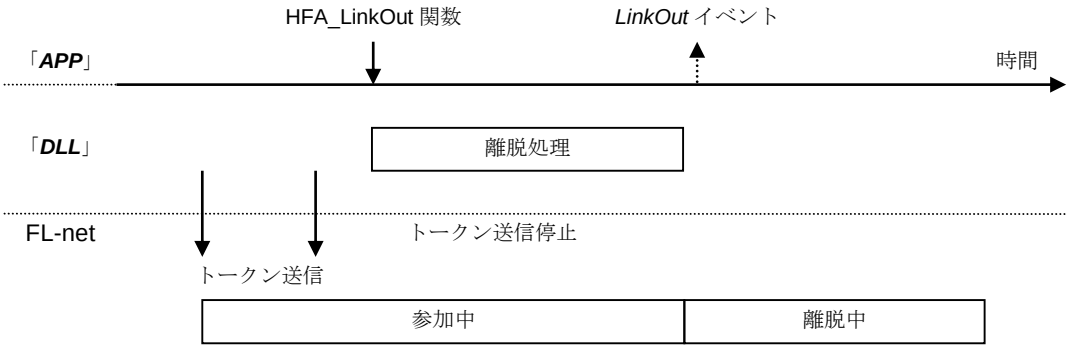
＜引数＞

型	変 数	I/O	内 容
LONG_PTR	LinkID	IN	「***DLL***」内部メモリをアクセスするための識別子。 HFA_AttachLink 関数で取得した値を指定してください。

＜詳細＞

FL-net ネットワークからの離脱を行います。本関数をコールすると、「***DLL***」はネットワーク離脱処理を開始し、離脱が成功した時点でノード離脱（*LinkOut*）イベントが発生します。ノード離脱イベント以降のイベントは通知されなくなります。本関数コール後、離脱をキャンセルすることはできません。

モニタモードを実行していた場合は、モニタモードを終了します。



＜戻り値＞

値	内 容
0	正常終了（リンク離脱待機中）
-1	引数異常（未登録の LinkID）
2	自ノード未参加（参加待機中、モニタモード中の場合は含まない）

＜使用例＞

１）FL-net から離脱する。

```
void SampleLinkOut1( ){
    long lRet = HFA_LinkOut(G_LinkID);        // FL-net ネットワークから離脱
}
```

＜関連事項＞

・ HFA_AttachLink 関数，HFA_LinkIn 関数
・ *LinkOut* イベント

4.1.4. HFA_WriteCommon

コモンメモリ書込み

＜関数 I/F＞

long HFA_WriteCommon1(long StartAddr, long Bytes, unsigned char *Data, long AddrFlg)
long HFA_WriteCommon2(long StartAddr, long Words, unsigned char * Data, long AddrFlg)

＜引数＞

型	変 数	I/O	内 容
long	StartAddr	IN	先頭アドレス。 ・HFA_WriteCommon1 はバイト単位 ・HFA_WriteCommon2 はワード単位
long	Bytes／Words	IN	書込みサイズ。 ・HFA_WriteCommon1 はバイト単位 ・HFA_WriteCommon2 はワード単位
unsigned char *	Data	IN	書込みデータの先頭ポインタ。 データのアクセス方法につきましては、使用例をご参照ください。 注）呼び出し元「***APP***」では、必ず書込みサイズ以上の領域を確保してください。
long	AddrFlg	IN	アドレス指定方法。 ・0＝相対アドレス指定 ・0≠絶対アドレス指定

＜引数範囲＞

引 数		HFA_WriteCommon1	HFA_WriteCommon2	備 考
StartAddr		0≦（値）＜0x400	0≦（値）＜0x2000	
Size	Bytes	0＜（値）≦0x400	－	
	Words	－	0＜（値）≦0x2000	
StartAddr+Size		（値）≦0x400	（値）≦0x2000	

＜詳細＞

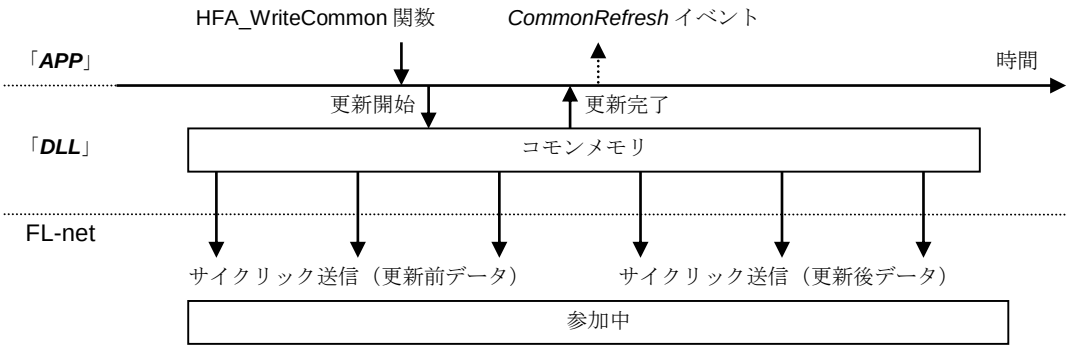
コモンメモリ領域[1]（領域１，領域２）にデータを書込みます。書込み可能な範囲は、自ノード送信領域（HFA_SetCommon 関数で設定）のみです。自ノード送信領域以外に書込みを指定した場合、戻り値=引数異常となり、コモンメモリは更新されません。

領域のアドレス指定は、相対アドレスまたは絶対アドレスの選択が可能です（AddrFlag 引数）。絶対アドレス指定の場合は、コモンメモリ全体のアドレス（領域１の場合：0x0～0x3FF、領域２の場合：0x0～0x1FFF）を指定します。相対アドレス指定の場合は、自ノード送信領域の先頭を 0 とした相対アドレスを指定します。

コモンメモリの書込みは、FL-net ネットワーク未参加状態でも可能です。この場合、「***DLL***」内部メモリの値が更新されます。モニタモードの場合は、戻り値=モニタモード中となり、コモンメモリは更新されません。

コモンメモリの書込みでコモンメモリの値が変化した場合、コモンメモリ更新（*CommonRefresh*）イベントが発生します。ただし、ネットワーク未参加時は、コモンメモリ更新イベントは発生しません。

[1] 領域のメモリマップは、付録 4 をご参照ください。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	モニタモード中

＜使用例＞

１）コモンメモリ領域 1 の指定アドレス（10 バイト目）ー指定ビット（下位 0 ビット目）の値を ON にする。

```
void SampleWriteCommon1( ){
    unsigned char ucData;
    // 指定アドレスのコモンメモリ値を読出す（バイト単位）
    long lRet = HFA_ReadCommon1(10, 1, &ucData, 1, 0);
    ucData |= 0x01;                    // 0 ビット目を ON にする（ビット演算）
    // 指定位置のコモンメモリ値を書込む（バイト単位）
    lRet = HFA_WriteCommon1(10, 1, &ucData, 1);
}
```

２）コモンメモリ領域 2 の指定アドレス（100 ワード目）から 2 ワード分の値を同時に更新する。

```
void SampleWriteCommon2( ){
    unsigned short usData[2];
    usData[0] = 0x1234;                // アドレス=100 の値をセット
    usData[1] = 0xABCD;                // アドレス=101 の値をセット
    // 2 ワード分のコモンメモリ値を書込む。
    long lRet = HFA_WriteCommon2(100, 2, (unsigned char *)usData, 1);
}
```

＜関連事項＞

・ HFA_SetCommon 関数，HFA_ReadCommon1 関数，HFA_ReadCommon2 関数
・ *CommonRefresh* イベント

4.1.5. HFA_ReadCommon

コモンメモリ読出し

＜関数 I/F＞

long HFA_ReadCommon1 (long StartAddr, long Bytes, unsigned char *Data, long AddrFlg, long NodeNo)
long HFA_ReadCommon2 (long StartAddr, long Words,unsigned char *Data, long AddrFlg, long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	StartAddr	IN	読出し開始アドレス ・HFA_ReadCommon1 はバイト単位 ・HFA_ReadCommon2 はワード単位
long	Bytes／Words	IN	読出しサイズ ・HFA_ReadCommon1 はバイト単位 ・HFA_ReadCommon2 はワード単位
unsigned char *	Data	OUT	読出し先バッファの先頭ポインタ データのアクセス方法につきましては、使用例をご参照ください。 **注）呼び出し元「*APP*」では、必ず読出しサイズ以上の領域を確保してください。**
long	AddrFlg	IN	アドレス指定方法 ・0＝相対アドレス指定 ・0≠絶対アドレス指定
long	NodeNo	IN	相対アドレス指定の対象ノード番号。自ノードの指定も可能です。 絶対アドレス指定の場合、値は無視されます。

＜引数範囲＞

引 数		HFA_ReadCommon1	HFA_ReadCommon2	備 考
StartAddr		0≦（値）＜0x400	0≦（値）＜0x2000	
Size	Bytes	0＜（値）≦0x400	－	
	Words	－	0＜（値）≦0x2000	
StartAddr+Size		（値）≦0x400	（値）≦0x2000	
NodeNo		1≦（値）≦254	1≦（値）≦254	自ノード可。

＜詳細＞

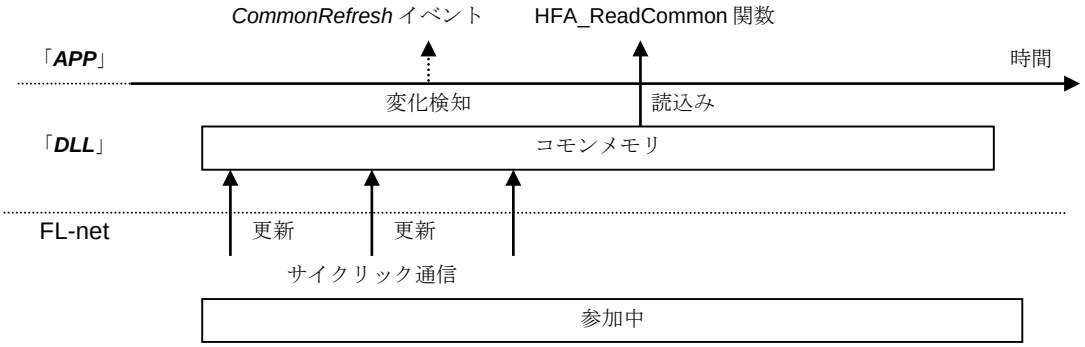
コモンメモリ領域（領域１，領域２）のデータを読出します。

領域のアドレス指定は、相対アドレスまたは絶対アドレスの選択が可能です（AddrFlag 引数）。絶対アドレス指定の場合は、コモンメモリ全体のアドレス（領域１の場合：0x0～0x3FF、領域２の場合：0x0～0x1FFF）を指定します。相対アドレス指定の場合は、対象ノード送信領域の先頭を 0 とした相対アドレスを指定します。

コモンメモリ読出しは、FL-net ネットワーク未参加状態でも可能です。この場合、戻り値=自ノード未参加となり、「***DLL***」内部メモリの値を読出します。（ネットワーク上の値とは異なります。）ただし、アドレス指定方法が相対アドレス指定で、対象ノードがネットワークに未参加の場合、戻り値=対象ノード未参加となり、コモンメモリは読出されません。

アドレス指定方法が相対アドレス指定の場合、対象ノードの FA リンク状態によりコモンメモリの有効／無効チェックを行います。コモンメモリが無効の場合は、戻り値=コモンメモリ無効となりますが、コモンメモリの読出しは行います。なお、指定したアドレスおよびサイズが対象ノードの送信領域外の場合、戻り値=引数異常となり、コモンメモリは読出されません。

アドレス指定方法が絶対アドレス指定の場合は、コモンメモリの有効／無効チェックは行いません。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加
5	対象ノード未参加
8	コモンメモリ無効（対象ノードの FA リンク状態=コモンメモリ無効の場合）

＜使用例＞

１）コモンメモリ領域 1 の指定アドレス（10 バイト目）の指定ビット（下位 0 ビット目）の値を参照する。

```
void SampleReadCommon1( ){
    unsigned char ucData, ucBit;
    // 指定位置のコモンメモリ値を読出す。
    long lRet = HFA_ReadCommon1(10, 1, &ucData, 1, 0);
    ucBit = (ucData & 0x01) ? 1 : 0;    // 0 ビット目の値を参照する（ビット演算）
}
```

２）コモンメモリ領域 2 の指定位置（アドレス=100 ワード）から 2 ワード分の値を同時に参照する。

```
void SampleReadCommon2( ){
    long lRet;
    unsigned short usData[2];
    // 2 ワード分のコモンメモリ値を読出す。
    long lRet = HFA_ReadCommon2(100, 2, (unsigned char *)usData, 1, 0);
}
```

＜関連事項＞

・ HFA_WriteCommon1 関数，HFA_WriteCommon2 関数
・ *CommonRefresh* イベント

4.1.6. HFA_GetNodeStatus

ノード管理情報パラメータ読出し

＜関数 I/F＞

long HFA_GetNodeStatus(long *NodeNo, NODE *Node)

＜引数＞

型	変 数	I/O	内 容
long *	NodeNo	IN/ OUT	読出し対象ノード番号。（0～254） 自ノード番号の指定も可能です。0 を指定することで、自ノードの情報を読出すことができます。この場合、自ノード番号が格納されます。
NODE *	Node	OUT	ノード情報[2] ※詳細は、以下をご参照ください。

＜詳細＞

ノード毎の管理情報パラメータを読出します。読出しを行う対象ノードに応じて、取得可能な項目が異なります。取得可能な項目および関数戻り値の一覧を以下に示します。

1）NodeNo=自ノード指定の場合

自ノードの参加状態に応じて、取得可能な項目が異なります。

項目	構造体メンバ名	自ノード参加状態による取得有無※			
		未参加	参加待機中	参加中	モニタ中
ベンダ名称	VendorName	○	○	○	○
製造業者形式	MakerType	○	○	○	○
ノード名称	NodeName	○	○	○	○
コモンメモリ 1 アドレス [3]	Common1Addr	○	○	○	×
コモンメモリ 1 サイズ [3]	Common1Bytes	○	○	○	×
コモンメモリ 2 アドレス [4]	Common2Addr	○	○	○	×
コモンメモリ 2 サイズ [4]	Common2Words	○	○	○	×
トークン監視時間	TokenTimeout	×	○	○	×
リフレッシュサイクル許容時間	MaxRefreshCycle	×	×	○	×
リフレッシュサイクル実測値	RefreshCycle	×	×	○	×
最小許容フレーム間隔	MinFrameInterval	×	○	○	×
上位層の状態 [5]	UpperStatus	○	○	○	○
FA リンクの状態 [5]	LinkStatus	○	○	○	○
プロトコルタイプ	ProtocolVersion	○	○	○	○
自ノードのステータス [6]	MyNodeStatus	○	○	○	○
関数戻り値		2	2	0	2

※取得有無欄の記号の意味は以下の通りです。

・○：取得値有効

・×：取得値無効

[2] ノード情報の構造体は、4.8.1 項をご参照ください。

２）NodeNo=他ノード指定の場合

自ノードおよび他ノードの参加状態に応じて、取得可能な項目が異なります。他ノードのベンダ名称, 製造業者形式, ノード名称の最新情報を取得する場合は、HFA_ReqReadNetParam 関数で他ノードのネットワークパラメータ読出しを行い、*RecvRepReadNetParam* イベント発生後に、本関数をコールする必要があります。

項目	構造体メンバ名	参加状態による取得有無※					
		自ノード	未参加	参加中		モニタ中	
		他ノード	参加中/未参加	参加中	未参加	参加中	未参加
ベンダ名称	VendorName		×	△	×	×	×
製造業者形式	MakerType		×	△	×	×	×
ノード名称	NodeName		×	△	×	×	×
コモンメモリ 1 アドレス[3]	Common1Addr		×	○	×	○	×
コモンメモリ 1 サイズ [3]	Common1Bytes		×	○	×	○	×
コモンメモリ 2 アドレス[4]	Common2Addr		×	○	×	○	×
コモンメモリ 2 サイズ [4]	Common2Words		×	○	×	○	×
トークン監視時間	TokenTimeout		×	○	×	○	×
リフレッシュサイクル許容時間	MaxRefreshCycle		×	○	×	○	×
リフレッシュサイクル実測値	RefreshCycle		×	○	×	×	×
最小許容フレーム間隔	MinFrameInterval		×	○	×	○	×
上位層の状態[5]	UpperStatus		×	○	×	○	×
FA リンクの状態 [5]	LinkStatus		◇1	○	◇2	○	◇2
プロトコルタイプ	ProtocolVersion		×	○	×	○	×
自ノードのステータス[6]	MyNodeStatus		○	○	○	○	○
関数の戻り値			2	0	5	2	5

※取得有無欄の記号の意味は以下の通りです。

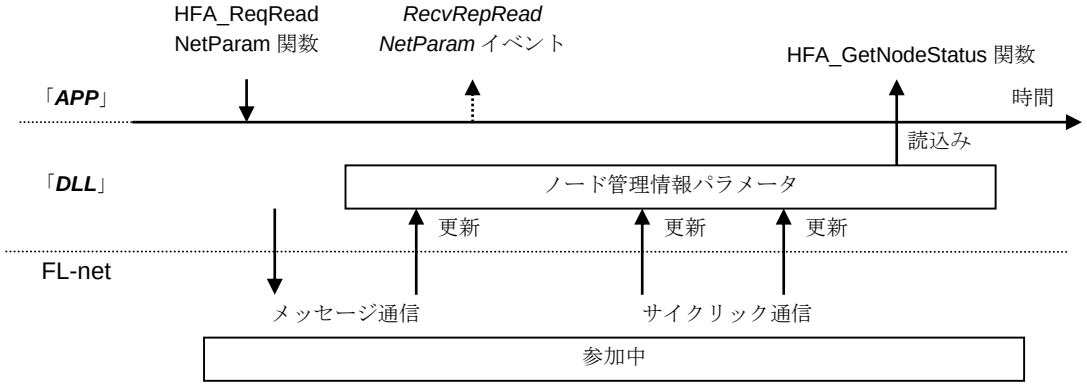
・○ ：取得値有効

・× ：取得値無効

・△ ：ネットワークパラメータリード応答受信後、最新の情報が取得可能になります。

・◇1：自ノードが最後にネットワークに参加していた時点の通信無効検知フラグのみ有効です。

・◇2：対象ノードが最後にネットワークに参加していた時点の通信無効検知フラグのみ有効です。



[3] コモンメモリ１のアドレスおよびサイズは、バイト単位。

[4] コモンメモリ２のアドレスおよびサイズは、ワード単位。

[5] 詳細は、6.3 節をご参照ください。

[6] 詳細は、6.3 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加
5	対象ノード未参加

＜使用例＞

１）指定ノード（ノード番号=10）のノード管理情報を読出す。

```
void SampleNodeStatus1( ){
    long lNodeNo=10;
    NODE xNode;
    memset(&xNode, sizeof(xNode));                          // ノード管理情報初期化
    if(HFA_GetNodeStatus(&lNodeNo, &xNode) == 0){           // 指定ノードのノード管理情報読出し
        printf("ノード名称=%s¥n", xNode.NodeName);          // ノード名称を参照
        // 上位層の動作情報を参照（ビット演算）
        printf("上位層の動作情報=%s¥n", (xNode.UpperStatus & 0x8000) ? "RUN" : "STOP");
    }
}
```

＜関連事項＞

・ HFA_ReqReadNetParam 関数

・ *RecvRepReadNetParam* イベント

4.1.7. HFA_GetNetworkStatus

ネットワークステータス読出し

＜関数 I/F＞

long HFA_GetNetworkStatus(NETWORK *Network, unsigned char *Node)

＜引数＞

型	変 数	I/O	内 容
NETWORK *	Network	OUT	ネットワーク管理情報パラメータ[7]。※詳細は、以下をご参照ください。
unsigned char *	Node	OUT	ノード参加状態。FL-net ネットワークへのノード参加状態を、ノード番号の昇順（1～254）で 1 バイト毎に格納します。unsigned char 型配列の添え字がノード番号と対応し、参加状態の値は以下となります。 ・ '0'=未参加 ・ '1'=参加中 ・ '2'=通信無効検知（未参加） **注）呼び出し元「*APP*」では、必ず 254 バイト以上の領域を確保してください。**

＜詳細＞

ネットワークステータスを読出します。取得可能な項目は以下の通りです。FL-net ネットワーク未参加（モニタモード除く）の状態で本関数をコールした場合、戻り値=自ノード未参加となり、ステータスの読出しは行いません。

取得可能項目	変数名
トークン保持ノード	TokenOwner
最小フレーム間隔	MinFrameInterval
リフレッシュサイクル許容時間	MaxRefreshCycle
リフレッシュサイクル実測値	RefreshCycle
リフレッシュサイクル最大値	RefreshCycleHigh
リフレッシュサイクル最小値	RefreshCycleLow

＜戻り値＞

値	内 容
0	正常終了（モニタモード含む）
2	自ノード未参加

[7] ネットワーク情報の構造体は、4.8.2 項をご参照ください。

＜使用例＞

１）ネットワーク管理情報を参照する。

```
void SampleNetworkStatus1( ){
      NETWORK xNetwork;
      unsigned char ucNodeNo[256];
      if(HFA_GetNetworkStatus(&xNetwork, ucNodeNo) == 0){      // ネットワーク管理情報読出し
            // リフレッシュサイクル実測値を参照
            printf(“リフレッシュサイクル実測値=%ld¥n”, xNetwork.RefreshCycle);
            printf(“ノード 100 の参加状態=%d¥n”, ucNodeNo[100]);    // ノード 100 の参加状態を参照
      }
}
```

＜関連事項＞

なし

4.1.8. HFA_GetMyNodeLog

自ノードログ情報読出し（FL-net　Ver.2 専用）

＜関数 I/F＞

long HFA_GetMyNodeLog(LOG *Log)

＜引数＞

型	変 数	I/O	内 容
LOG *	Log	OUT	ログ情報[8]

＜詳細＞

自ノードのログ情報（FL-net Ver.2 専用）を読出します。FL-net Ver.3 対応のログ情報を取得する場合は、HFA_GetMyNodeLogV3 関数を使用してください。

他ノードのログ情報を読出す場合は、HFA_ReqReadLog 関数を使用してください。

＜戻り値＞

値	内 容
0	正常終了

＜使用例＞

1）自ノードログ情報を参照する。

```
void SampleGetMyLog1( ){
      LOG xLog;
      HFA_GetMyNodeLog(&xLog);    // 自ノードログ情報読出し
      // 1 対 1 メッセージ送信回数を参照
      printf("1 対 1 メッセージ送信回数=%lu¥n", xLog.Frame.SendPeerToPeer);
}
```

＜関連事項＞

・ HFA_GetMyNodeLogV3 関数
・ HFA_ReqReadLog 関数

[8] ログ情報の構造体は、0 項をご参照ください。

4.1.9. HFA_GetMyNodeLogV3

自ノードログ情報読出し（FL-net Ver.3 対応版）

＜関数 I/F＞

```
long HFA_GetMyNodeLogV3(HFA_LOG_V3 *Log)
```

＜引数＞

型	変 数	I/O	内 容
HFA_LOG_V3 *	Log	OUT	ログ情報[9]

＜詳細＞

自ノードのログ情報を読出します。HFA_GetMyNodeLog 関数の拡張版で、FL-net Ver.2 および Ver.3 のログ情報を読み出す場合に使用します。

他ノードのログ情報を読出す場合は、HFA_ReqReadLog 関数を使用してください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

1）自ノードログ情報を取得する。

```
void SampleGetMyNodeLogV3(){
    HFA_LOG_V3 xLog;
    HFA_GetMyNodeLogV3(&xLog);        // 自ノードログ情報取得
    printf("汎用通信データ送信元ログ測定時間=%d", xLog.IP.Time);
}
```

＜関連事項＞

- HFA_GetMyNodeLog 関数
- HFA_ReqReadLog 関数

[9] ログ情報の構造体は、4.8.25 項をご参照ください。

4.1.10. HFA_ClearMyNodeLog

自ノードログ情報クリア

＜関数 I/F＞

long HFA_ClearMyNodeLog(void)

＜引数＞

なし

＜詳細＞

自ノードのログ情報をクリアします。自ノードが FL-net ネットワークに参加している場合は、ログクリア（*LogClear*）イベントが発生します。

他ノードのログ情報をクリアする場合は、HFA_ReqClearLog 関数を使用してください。

＜戻り値＞

値	内 容
0	正常終了

＜使用例＞

1）自ノードログ情報をクリアする。

```
void SampleClearMyLog1( ){
    long lRet = HFA_ClearMyNodeLog( );      // 自ノードログ情報クリア
}
```

＜関連事項＞

・ HFA_ReqClearLog 関数
・ *LogClear* イベント

4.1.11. HFA_SetCommon

コモンメモリ送信領域割り当て設定

＜関数 I/F＞

long HFA_SetCommon(long Common1Addr, long Common1Bytes,
long Common2Addr, long Common2Words)

＜引数＞

型	変 数	I/O	内 容
long	Common1Addr	IN	コモンメモリ領域１先頭アドレス（バイト単位）
long	Common1Bytes	IN	コモンメモリ領域１サイズ（バイト単位）
long	Common2Addr	IN	コモンメモリ領域２先頭アドレス（ワード単位）
long	Common2Words	IN	コモンメモリ領域２サイズ（ワード単位）

＜引数範囲＞

項 目（式）	値正常範囲	備 考
Common1Addr	0≦（値）≦0x3FF	偶数のみ
Common1Bytes	0≦（値）≦0x400	偶数のみ
Common2Addr	0≦（値）≦0x1FFF	
Common2Words	0≦（値）≦0x2000	
Common1Addr+Common1Bytes	（値）≦0x400	
Common2Addr+Common2Words	（値）≦0x2000	

＜詳細＞

コモンメモリ送信領域（領域１，領域２）の範囲（アドレスおよびサイズ）を設定します。コモンメモリ送信領域の設定は、FL-net ネットワーク未参加時に設定可能です。ネットワーク参加中（モニタモード含む）の状態で本関数をコールした場合、戻り値=自ノード参加中となり、設定の変更は行いません。

Common1Bytes=0 および Common2Words=0 を指定した場合、コモンメモリ送信領域無し（受信専用）として設定されます。領域 1（Common1Addr，Common1Bytes）は、バイト単位で設定してください。

FL-net ネットワーク内の他ノードと重複しないように領域を設定してください。FL-net ネットワークに自ノードが参加する時点で、コモンメモリ送信領域が他ノードと重複するかのチェックを行います。領域の重複を検知した場合は参加（*LinkIn*）イベントの引数で異常を通知します。この場合、自ノードのコモンメモリのアドレスおよびサイズが 0 に設定されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	自ノード参加中（モニタモード含）

＜使用例＞

１）コモンメモリ送信領域を無しに設定する。

```
void SampleSetCommon1( ){
      long lRet = HFA_SetCommon(0, 0, 0, 0);              // コモンメモリ送信領域=無し設定
}
```

２）コモンメモリ送信領域を全領域（領域 1=0～1023[バイト]，領域 2=0～8191[ワード]）に設定する。

```
void SampleSetCommon2( ){
      long lRet = HFA_SetCommon(0, 1024, 0, 8192);      // コモンメモリ送信領域=全領域設定
}
```

＜関連事項＞

・ HFA_LinkIn 関数，HFA_WriteCommon1 関数，HFA_WriteCommon2 関数

・ *LinkIn* イベント

4.1.12. HFA_GetCommon

コモンメモリ送信領域取得

＜関数 I/F＞

```
long HFA_GetCommon(long *Common1Addr, long *Common1Bytes,
                   long *Common2Addr, long *Common2Words)
```

＜引数＞

型	変 数	I/O	内 容
long *	Common1Addr	OUT	コモンメモリ領域１先頭アドレスのポインタ（バイト単位）
long *	Common1Bytes	OUT	コモンメモリ領域１サイズのポインタ（バイト単位）
long *	Common2Addr	OUT	コモンメモリ領域２先頭アドレスのポインタ（ワード単位）
long *	Common2Words	OUT	コモンメモリ領域２サイズのポインタ（ワード単位）

＜詳細＞

コモンメモリ送信領域(領域 1，領域 2)の設定値を取得します。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１） コモンメモリ送信領域を取得する。

```
void SampleGetCommon(){
    long C1Addr, C1Size, C2Addr, C2Size;
    if(HFA_GetCommon(&C1Addr, &C1Size, &C2Addr, &C2Size) == 0){
        printf("コモンメモリ領域 1 アドレス=%d¥n",C1Addr);
        printf("コモンメモリ領域 1 サイズ=%d¥n",C1Size);
    }
}
```

＜関連事項＞

・HFA_SetCommon 関数

4.1.13. HFA_SetNodeName

ノード名（設備名）設定

＜関数 I/F＞

long HFA_SetNodeName (char *NodeName)

＜引数＞

型	変 数	I/O	内 容
char *	NodeName	IN	ノード名（設備名）。 ※10 バイト目以降のデータは無視し、10 バイトに満たないデータ部分は NULL（0）で埋めた値とします。

＜詳細＞

自ノードのノード名（設備名）を設定します。本関数がコールされない場合のノード名のデフォルト値は、6.2 節をご参照ください。FL-net ネットワーク参加中の場合も、ノード名の設定は可能です。

＜戻り値＞

値	内 容
0	正常終了

＜使用例＞

１）ノード名称を”Sample”に設定する。

```
void SampleSetNodeName( ){
    long lRet = HFA_SetNodeName(“Sample”);          // ノード名称=”Sample”設定
}
```

＜関連事項＞

・HFA_GetNodeName 関数

4.1.14. HFA_GetNodeName

ノード名（設備名）取得

＜関数 I/F＞

long HFA_GetNodeName (char *NodeName)

＜引数＞

型	変 数	I/O	内 容
char *	NodeName	OUT	ノード名（設備名） **注) 呼び出し元「APP」で必ず 10 バイト以上の領域を確保してください。**

＜詳細＞

自ノードのノード名（設備名）を取得します。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）ノード名(設備名)を取得する。

```
void SampleGetNodeName(){
    char cName[20];
    if(HFA_GetNodeName(cName) == 0){
        printf("ノード名(設備名)=%s¥n", cName);
    }
}
```

＜関連事項＞

・HFA_SetNodeName 関数

4.1.15. HFA_SetNodeNo

ノード番号設定

＜関数 I/F＞

long HFA_SetNodeNo(long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	FL-net ネットワークに参加する自ノード番号 ・1～254 を指定した場合は、参加モードとして FL-net ネットワークに参加します。 ・0 を指定した場合は、モニタモードを開始します。

＜引数範囲＞

項目(式)	値正常範囲	備 考
NodeNo	0≦（値）≦254	0 はモニタモード

＜詳細＞

自ノード番号を設定します。自ノード番号の設定は FL-net ネットワーク未参加時に設定可能です。FL-net ネットワークに参加する場合は、HFA_LinkInDefault 関数をコールしてください。HFA_LinkIn 関数をコールすると、ノード番号が上書きされます。HFA_SetNodeNo 関数および HFA_LinkIn 関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	既に参加中(モニタ中)
7	既に参加待機中

＜使用例＞

１）ノード番号=100 を設定して、参加する。

```
void SampleSetNodeNo(){
    long lRet = HFA_SetNodeNo(100);
}
```

＜関連事項＞

・HFA_GetNodeNo 関数

・HFA_LinkInDefault 関数，HFA_LinkIn 関数

4.1.16. HFA_GetNodeNo

ノード番号取得

＜関数 I/F＞

long HFA_GetNodeNo(long *NodeNo)

＜引数＞

型	変 数	I/O	内 容
long *	NodeNo	OUT	ノード番号のポインタ

＜詳細＞

自ノード番号を取得します。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

1）自ノード番号を取得する。

```
void SampleGetNodeNo(){
      long NodeNo;
      if(HFA_GetNodeNo (&NodeNo) == 0){
            printf(“ノード番号=%d¥n”, NodeNo);
      }
}
```

＜関連事項＞

・HFA_SetNodeNo 関数

4.1.17. HFA_SetTokenWatchTime

トークン監視時間設定

＜関数 I/F＞

long HFA_SetTokenWatchTime(long TokenTimer)

＜引数＞

型	変 数	I/O	内 容
long	TokenTimer	IN	トークン監視時間（ms 単位）

＜引数範囲＞

項目(式)	値正常範囲	備 考
TokenTimer	1≦（値）≦255	ms 単位

＜詳細＞

トークン監視時間を[ms]単位で設定します。リンク参加中は、値の設定はできません。値を設定する場合は、リンク参加前に行う必要があります。FL-net ネットワークに参加する場合は、HFA_LinkInDefault 関数をコールしてください。HFA_LinkIn 関数をコールすると、値が上書きされます。HFA_SetTokenWatchTime 関数および HFA_LinkIn 関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	既に参加中(参加待機中，モニタ中含む)

＜使用例＞

１）トークン監視時間=100ms を設定する。

```
void SampleSetTokenWatchTime(){
    long lRet = HFA_SetTokenWatchTime(100);
}
```

＜関連事項＞

・HFA_GetTokenWatchTime 関数

・HFA_LinkInDefault 関数，HFA_LinkIn 関数

4.1.18. HFA_GetTokenWatchTime

トークン監視時間取得

＜関数 I/F＞

long HFA_GetTokenWatchTime(long *TokenTimer)

＜引数＞

型	変 数	I/O	内 容
long *	TokenTimer	OUT	トークン監視時間（ms 単位）のポインタ。

＜詳細＞

トークン監視時間を取得します。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）トークン監視時間を取得する。

```
void SampleGetTokenWatchTime(){
    long lTokenTimer;
    if(HFA_GetTokenWatchTime(&lTokenTimer) == 0){
        printf(“トークン監視時間=%d¥n”,lTokenTimer);
    }
}
```

＜関連事項＞

・HFA_SetTokenWatchTime 関数

4.1.19. HFA_SetMinFrameInterval

最小許容フレーム間隔設定

＜関数 I/F＞

long HFA_SetMinFrameInterval(long MinFrameInterval)

＜引数＞

型	変 数	I/O	内 容
long	MinFrameInterval	IN	最小許容フレーム間隔（100μs 単位）。 既にネットワークへ参加中（参加待機中）の状態に本関数がコールされた場合は、入力値は無視されます。

＜引数範囲＞

項目(式)	値正常範囲	備 考
MinFrameInterval	0≦（値）≦50	100μs 単位

＜詳細＞

最小許容フレーム間隔を[100μs]単位で設定します。リンク参加中は、値の設定はできません。値を設定する場合は、リンク参加前に行う必要があります。FL-net ネットワークに参加する場合は、HFA_LinkInDefault 関数をコールしてください。HFA_LinkIn 関数をコールすると、値が上書きされます。HFA_SetMinFrameInterval 関数および HFA_LinkIn 関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	既に参加中(参加待機中，モニタ中含む)

＜使用例＞

1）最小許容フレーム間隔=1ms を設定する。

```
void SampleSetMinFrameInterval (){
    long lRet = HFA_SetMinFrameInterval (10);
}
```

＜関連事項＞

・HFA_GetMinFrameInterval 関数

・HFA_LinkInDefault 関数，HFA_LinkIn 関数

4.1.20. HFA_GetMinFrameInterval

最小許容フレーム間隔取得

＜関数 I/F＞

long HFA_GetMinFrameInterval(long *MinFrameInterval)

＜引数＞

型	変 数	I/O	内 容
long *	MinFrameInterval	OUT	最小許容フレーム間隔（100μs 単位）のポインタ。

＜詳細＞

最小許容フレーム間隔を取得します。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）最小許容フレーム間隔を取得する。

```
void SampleGetMinFrameInterval(){
    long lMinFrameInterval;
    if(HFA_GetMinFrameInterval(&lMinFrameInterval) == 0){
        printf(“最小許容フレーム間隔=%d¥n”, lMinFrameInterval);
    }
}
```

＜関連事項＞

・HFA_SetMinFrameInterval 関数

4.1.21. HFA_SetIP

IP アドレス設定

＜関数 I/F＞

long HFA_SetIP(char *LocalIP)

＜引数＞

型	変 数	I/O	内 容
char *	LocalIP	IN	ローカル IP アドレス。 複数の LAN カードを実装している場合に、特定の LAN カードを指定することが可能です。 ・LAN カードを特定しない場合は、""（空文字）を指定します。この場合、OS が管理するデフォルトの LAN カードが選択されます。 ・LAN カードを特定する場合は、10 進数のゼロサプレスで"XXX.XXX.XXX.XXX"形式で指定します。

＜詳細＞

複数の LAN カードを実装している場合に、特定の LAN カードの IP アドレスを設定します。リンク参加中は、値の変更はできません。値を設定する場合は、リンク参加前に行う必要があります。FL-net ネットワークに参加する場合は、HFA_LinkInDefault 関数をコールしてください。HFA_LinkIn 関数をコールすると、値が上書きされます。HFA_SetIP 関数および HFA_LinkIn 関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	既に参加中(参加待機中，モニタ中含む)

＜使用例＞

１）"192.168.250.25"の LAN カードを指定する。

```
void SampleSetIP(){
    long lRet = HFA_SetIP("192.168.250.25");
}
```

＜関連事項＞

・HFA_GetIP 関数

・HFA_LinkInDefault 関数，HFA_LinkIn 関数

4.1.22. HFA_GetIP

IP アドレス取得

＜関数 I/F＞

long HFA_GetIP(char *LocalIP)

＜引数＞

型	変 数	I/O	内 容
char *	LocalIP	OUT	ローカル IP アドレスの先頭ポインタ。 **注）呼び出し元「APP」では、必ず 20 バイト以上の領域を確保してください。**

＜詳細＞

現在設定されている IP アドレスを取得します。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

1）IP アドレスを取得する。

```
void SampleGetIP(){
      char clp[20];
      if(HFA_GetIP(clp) == 0){
            printf("IP=%s¥n",clp);
      }
}
```

＜関連事項＞

・HFA_SetIP 関数

4.1.23. HFA_SetConfigParam

コンフィギュレーション用パラメータ設定

＜関数 I/F＞

long HFA_SetConfigParam(HFA_CONFIG_PARAM *Param)

＜引数＞

型	変 数	I/O	内 容
HFA_CONFIG_PARAM *	Param	IN	コンフィギュレーション用パラメータ[10] 。

＜詳細＞

コンフィギュレーション用パラメータを一括で設定します。コンフィギュレーション用パラメータの設定は FL-net ネットワーク未参加時に設定可能です。設定する項目は省略できませんので、全てのメンバに適切な値を設定してください。ひとつでも設定に失敗すると引数異常になり、コンフィギュレーション用パラメータの設定は更新されません。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	既に参加中
7	既に参加待機中

＜使用例＞

1）コンフィギュレーション用パラメータを設定する。

```
void SampleSetConfigParam(){
    HFA_CONFIG_PARAM xConfig;
    memset(&xConfig,0,sizeof(xConfig));          // 初期化
    xConfig.NodeNo = 1;                          // ノード番号
    xConfig.Common1Addr = 0;                     // 領域 1 アドレス(バイト単位)
    xConfig.Common1Bytes = 100;                  // 領域 1 サイズ(バイト単位)
    xConfig.Common2Addr = 0;                     // 領域 2 アドレス(ワード単位)
    xConfig.Common2Words = 200;                  // 領域 2 サイズ(ワード単位)
    xConfig.TokenWatchTime = 50;                 // トークン監視時間(ms 単位)
    xConfig.MinFrameInterval = 10;               // 最小許容フレーム間隔(100μs 単位)
    strcpy(xConfig.NodeName,”FLnet Ctrl”);       // ノード名
    strcpy(xConfig.LocalIP,”192.168.250.25”);    // ローカル IP アドレス

    long lRet = HFA_SetConfigParam(&xConfig);    // 設定
}
```

＜関連事項＞

・HFA_GetConfigParam 関数

---

[10] コンフィギュレーション用パラメータの構造体は、4.8.23 項をご参照ください。

4.1.24. HFA_GetConfigParam

コンフィギュレーション用パラメータ取得

＜関数 I/F＞

long HFA_GetConfigParam(HFA_CONFIG_PARAM *Param)

＜引数＞

型	変 数	I/O	内 容
HFA_CONFIG_PARAM *	Param	OUT	コンフィギュレーション用パラメータ[11] 。

＜詳細＞

コンフィギュレーション用パラメータを一括で取得します。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

1）コンフィギュレーション用パラメータを取得する。

```
void SampleGetConfigParam(){
    HFA_CONFIG_PARAM xConfig;
    if(HFA_GetConfigParam(&xConfig) == 0){
        printf("ノード番号=%d¥n", xConfig.NodeNo);
        printf("ノード名=%s¥n", xConfig.NodeName);
        printf("IP アドレス=%s¥n", xConfig.IP);
    }
}
```

＜関連事項＞

・HFA_SetConfigParam 関数

[11] コンフィギュレーション用パラメータの構造体は、4.8.23 項をご参照ください。

4.1.25. HFA_SetCommonRefreshDegree

コモンメモリ更新イベントの検知範囲設定

＜関数 I/F＞

long HFA_SetCommonRefreshDegree(long Common1Mode, long Common1Addr, long Common1Bytes,
long Common2Mode, long Common2Addr, long Common2Words)

＜引数＞

型	変 数	I/O	内 容
long	Common1Mode	IN	コモンエリア１設定モード。 ・0 =全領域による範囲設定 ・-1=範囲設定無し（既に設定された範囲を変更しない） ・1 =指定領域による範囲設定
long	Common1Addr	IN	コモンエリア１先頭アドレス（バイト単位）。
long	Common1Bytes	IN	コモンエリア１サイズ（バイト単位）。
long	Common2Mode	IN	コモンエリア２設定モード。 ・0 =全領域による範囲設定 ・-1=範囲設定無し（既に設定された範囲を変更しない） ・1 =指定領域による範囲設定
long	Common2Addr	IN	コモンエリア２先頭アドレス（ワード単位）。
long	Common2Words	IN	コモンエリア２サイズ（ワード単位）。

＜引数範囲＞

項 目（式）	値正常範囲	備 考
Common1Addr	0≦（値）≦0x3FF	
Common1Bytes	0≦（値）≦0x400	
Common2Addr	0≦（値）≦0x1FFF	
Common2Words	0≦（値）≦0x2000	
Common1Addr+Common1Bytes	（値）≦0x400	
Common2Addr+Common2Words	（値）≦0x2000	

＜詳細＞

コモンメモリ（領域１，２）の更新（*CommonRefresh*）イベントを検知する範囲（アドレスおよびサイズ）を設定します。本関数をコールすることで、コモンメモリ更新イベントの範囲を限定することができます。

設定モード（Common1Mode および Common2Mode）の値に応じて、設定される範囲が異なります。本関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）コモンメモリ更新検知範囲を無しに設定する（※この場合、*CommonRefresh* イベントは発生しなくなる）。

```
void SampleSetCommonRefresh1( ){
    HFA_SetCommonRefreshDegree(1, 0, 0, 1, 0, 0);        // コモンメモリ更新検知範囲=無し設定
}
```

２）コモンメモリ更新検知範囲を全領域（領域 1=0～1023[バイト]，領域 2=0～8191[ワード]）に設定する。

```
void SampleSetCommonRefresh2( ){
    HFA_SetCommonRefreshDegree(0, 0, 0, 0, 0, 0);        // コモンメモリ更新検知範囲=全領域設定
}
```

３）コモンメモリ更新検知範囲の領域 1 を 0～10[バイト]に変更する（領域 2 は変更しない）。

```
void SampleSetCommonRefresh3( ){
    HFA_SetCommonRefreshDegree(1, 0, 10, -1, 0, 0);      // コモンメモリ更新検知範囲の領域 1 を変更
}
```

＜関連事項＞

・ *CommonRefresh* イベント

4.1.26. HFA_SetControlEquipment

運転／停止状態設定

＜関数 I/F＞

long HFA_SetControlEquipment (long RunMode)

＜引数＞

型	変 数	I/O	内 容
long	RunMode	IN	自ノードの運転／停止状態を設定する運転モード。 ・ 0＝停止状態 ・ 0≠運転状態

＜詳細＞

自ノードの運転／停止状態を設定します。本関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了。

＜使用例＞

１）自ノード運転／停止状態を”運転”に設定する。

```
void SampleSetControlEquipment( ){
    HFA_SetControlEquipment(1);  // 運転状態に設定
}
```

＜関連事項＞

なし

4.2. メッセージ送信関数

メッセージ送信関数とは、FL-net ネットワークにメッセージフレームを送信するための I/F 関数です。メッセージ送信関数の一覧を以下に示します。

No.	関数名	概要
1	HFA_ReqReadByteBlock	バイトブロックリード要求
2	HFA_ReqWriteByteBlock	バイトブロックライト要求
3	HFA_ReqReadWordBlock	ワードブロックリード要求
4	HFA_ReqWriteWordBlock	ワードブロックライト要求
5	HFA_ReqReadNetParam	ネットワークパラメータリード要求
6	HFA_ReqWriteNetParam	ネットワークパラメータライト要求
7	HFA_ReqControlEquipment	運転／停止指令要求
8	HFA_ReqReadProfile	プロファイルリード要求
9	HFA_ReqReadLog	ログデータリード要求
10	HFA_ReqClearLog	ログデータクリア要求
11	HFA_ReqEchoMessage	メッセージ折り返し要求
12	HFA_RepReadByteBlock	バイトブロックリード応答
13	HFA_RepWriteByteBlock	バイトブロックライト応答
14	HFA_RepReadWordBlock	ワードブロックリード応答
15	HFA_RepWriteWordBlock	ワードブロックライト応答
16	HFA_RepWriteNetParam	ネットワークパラメータライト応答
17	HFA_RepControlEquipment	運転／停止指令応答
18	HFA_SendTransparency	透過形メッセージ送信
19	HFA_ReqVendorMessage	ベンダ固有メッセージ要求
20	HFA_RepVendorMessage	ベンダ固有メッセージ応答

メッセージ送信関数は、非ブロッキング関数です。（関数がリターンした地点で、メッセージの送信が完了しているとは限りません。）呼び出し元「***APP***」では、送信完了（*SendComplete*）またはタイムアウト（*SendTimeout*）時に発生するイベントにて送信結果を確認する必要があります。

メッセージの送信タイミングは、FL-net ネットワーク状況に影響されます。（ネットワーク状況によっては、送信遅延が発生する場合があります。）なお、メッセージ送信中の場合に、メッセージ送信関数をコールすると、戻り値=送信ビジーとなります。

4.2.1. HFA_ReqReadByteBlock

バイトブロックリード要求

＜関数 I/F＞

long HFA_ReqReadByteBlock (long NodeNo, unsigned long StartAddr, unsigned long Bytes)

＜引数＞

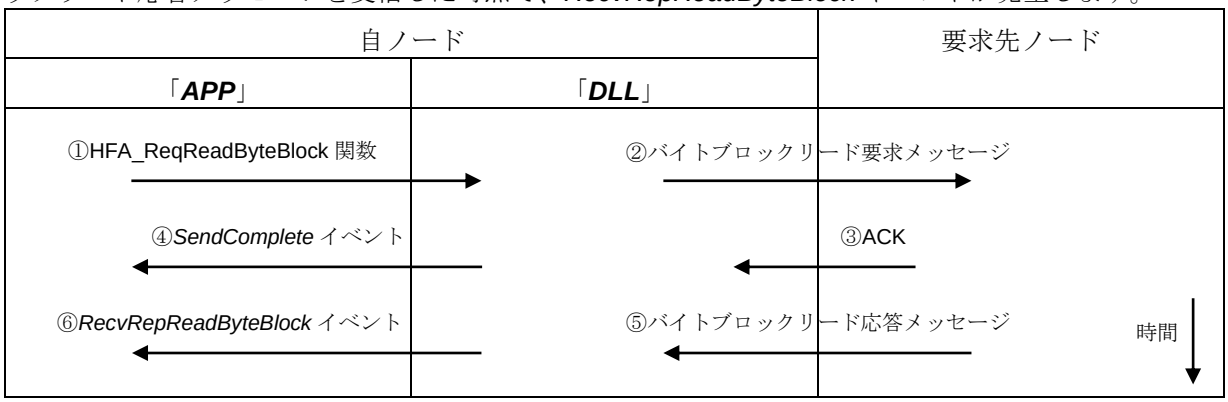
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
unsigned long	StartAddr	IN	バイトブロック開始アドレス（バイト単位）。
unsigned long	Bytes	IN	バイト数。

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Bytes	1≦（値）≦1024	
StartAddr+Bytes	（値）≦0x100000000	

＜詳細＞

他ノードからバイトブロックデータの読出しを要求します。引数で指定された条件に基づき、バイトブロックリード要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからバイトブロックリード応答メッセージを受信した時点で、*RecvRepReadByteBlock* イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 のバイトブロックデータ（先頭アドレス=16#10000 から 1000 バイト）の読出し要求を行う。

```
void SampleReqReadByteBlock( ){
    long lRet = HFA_ReqReadByteBlock(1, 0x10000, 1000);      // バイトブロックリード要求
}

// バイトブロックリード応答受信コールバック関数
void CALLBACK RecvRepReadByteBlock(long NodeNo, long Result, unsigned long Addr, long Bytes,
        unsigned char *Data, long ErrBytes, unsigned char *Error){
    if(Result == 0){            // 正常応答の場合
        memcpy(&G_ByteBlock[NodeNo][Addr], Data, Bytes);      // リード結果を待避
    }
    else if(Result == 1){       // 異常応答の場合
        char cError[1025];
        memset(cError, 0, sizeof(cError));
        memcpy(cError, Error, ErrBytes);                                    // エラーコード待避
        printf("バイトブロックリード異常発生。[エラー内容=%s]¥n", cError);          // エラー表示
    }
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepReadByteBlock* イベント

4.2.2. HFA_ReqWriteByteBlock

バイトブロックライト要求

＜関数 I/F＞

long HFA_ReqWriteByteBlock (long NodeNo, unsigned long StartAddr, unsigned long Bytes,
unsigned char *Data);

＜引数＞

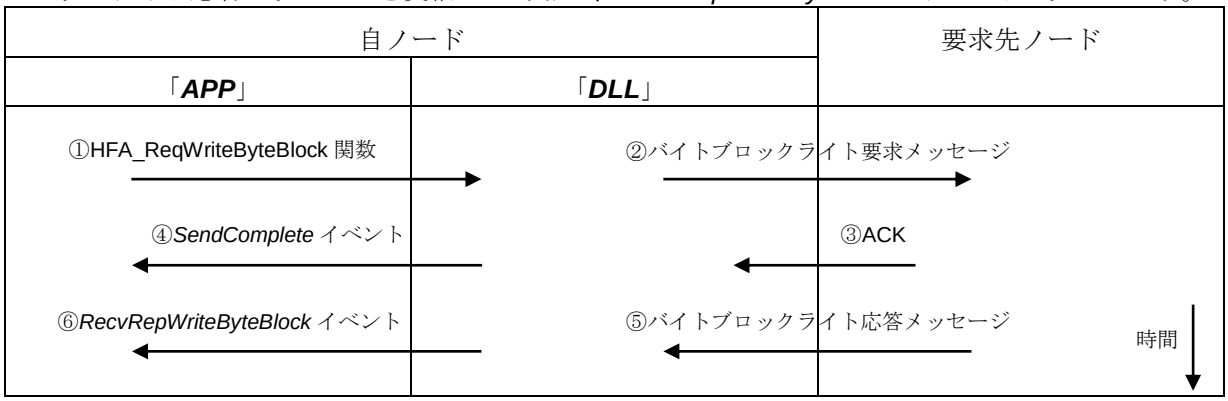
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
unsigned long	StartAddr	IN	バイトブロック開始アドレス（バイト単位）。
unsigned long	Bytes	IN	書込みバイト数。
unsigned char *	Data	IN	書込みデータの先頭アドレス。 **注）呼び出し元「*APP*」では、必ず書込みサイズ以上の領域を確保する必要があります。**

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Bytes	1≦（値）≦1024	
StartAddr+Bytes	（値）≦0x100000000	

＜詳細＞

他ノードに対し、バイトブロックデータの書込みを要求します。引数で指定された条件に基づき、バイトブロックライト要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからバイトブロックライト応答メッセージを受信した時点で、*RecvRepWriteByteBlock* イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
-10	セキュリティエラー（プロテクトキーが正しくセットされていない。または、ライセンス認証データが登録されていない）
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 のバイトブロックデータ（先頭アドレス=16#10000 から 2 バイト）の書込み要求を行う。

```
void SampleReqWriteByteBlock( ){
    unsigned char ucData[2];
    ucData[0] = 0x12;        // アドレス=16#10000 の値をセット
    ucData[1] = 0x34;        // アドレス=16#10001 の値をセット
    long lRet = HFA_ReqWriteByteBlock(1, 0x10000, 2, ucData);          // バイトブロックライト要求
}

// バイトブロックライト応答受信コールバック関数
void CALLBACK RecvRepWriteByteBlock(long NodeNo, long Result, unsigned long Addr, long Bytes,
        long ErrBytes, unsigned char *Error){
    if(Result == 0){                    // 正常応答の場合
        printf("バイトブロックライト完了¥n");
    }
    else if(Result == 1){               // 異常応答の場合
        char cError[1025];
        memset(cError, 0, sizeof(cError));
        memcpy(cError, Error, ErrBytes);                    // エラーコード待避
        printf("バイトブロックライト異常発生。[エラー内容=%s]¥n", cError);        // エラー表示
    }
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepWriteByteBlock* イベント

4.2.3. HFA_ReqReadWordBlock

ワードブロックリード要求

＜関数 I/F＞

long HFA_ReqReadWordBlock (long NodeNo, unsigned long StartAddr, unsigned long Words)

＜引数＞

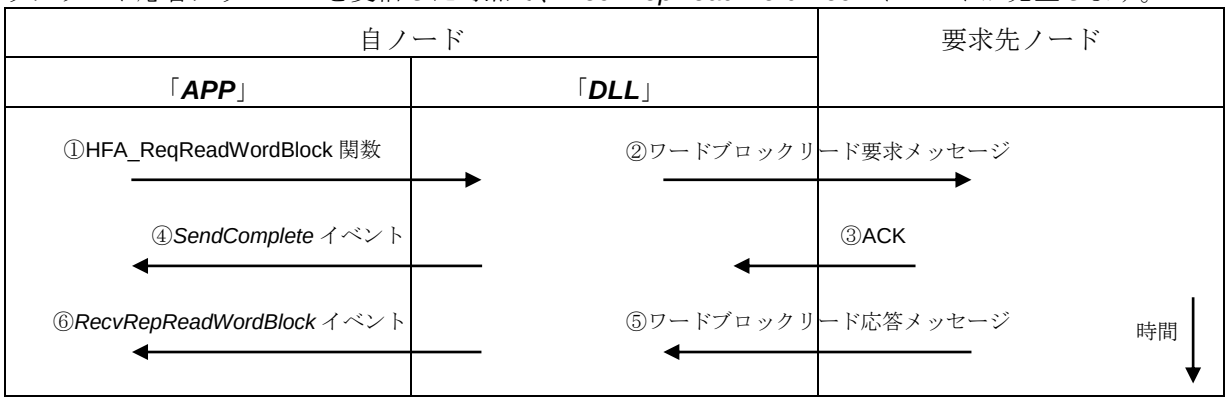
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
unsigned long	StartAddr	IN	ワードブロック開始アドレス（ワード単位）。
unsigned long	Words	IN	読出しワード数。

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Words	1≦（値）≦512	
StartAddr+Words	（値）≦0x100000000	

＜詳細＞

他ノードからワードブロックデータの読出しを要求します。引数で指定された条件に基づき、ワードブロックリード要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからワードブロックリード応答メッセージを受信した時点で、*RecvRepReadWordBlock* イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 のワードブロックデータ（先頭アドレス=16#10000 から 512 ワード）の読出し要求を行う。

```
void SampleReqReadWordBlock( ){
    long lRet = HFA_ReqReadWordBlock(1, 0x10000, 512);        // ワードブロックリード要求
}

// ワードブロックリード応答受信コールバック関数
void CALLBACK RecvRepReadWordBlock(long NodeNo, long Result, unsigned long Addr, long Words,
        unsigned char *Data, long ErrBytes, unsigned char *Error){
    if(Result == 0){            // 正常応答の場合
        memcpy(&G_WordBlock[NodeNo][Addr], Data, Words * 2);            // リード結果を待避
    }
    else if(Result == 1){       // 異常応答の場合
        char cError[1025];
        memset(cError, 0, sizeof(cError));
        memcpy(cError, Error, ErrBytes);                                // エラーコード待避
        printf("ワードブロックリード異常発生。[エラー内容=%s]¥n", cError);          // エラー表示
    }
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepReadWordBlock* イベント

4.2.4. HFA_ReqWriteWordBlock

ワードブロックライト要求

＜関数 I/F＞

long HFA_ReqWriteWordBlock (long NodeNo, unsigned long StartAddr, unsigned long Words,
unsigned char *Data)

＜引数＞

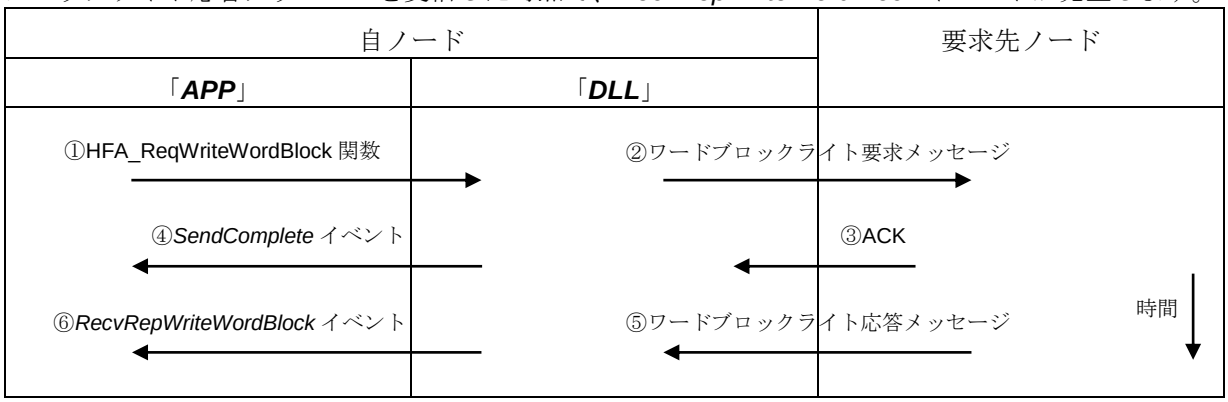
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
unsigned long	StartAddr	IN	ワードブロック開始アドレス（ワード単位）。
unsigned long	Words	IN	書込みワード数。
unsigned char *	Data	IN	書込みデータの先頭アドレス。 **注）呼び出し元「*APP*」では、必ず書込みサイズ以上の領域を確保する必要があります。「*APP*」側で管理しているワード単位（=2バイト単位）のデータを(unsigned char *)型に型変換して指定してください。**

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Words	1≦（値）≦512	
StartAddr+Bytes	（値）≦0x100000000	

＜詳細＞

他ノードに対し、ワードブロックデータの書込みを要求します。引数で指定された条件に基づき、ワードブロックライト要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからワードブロックライト応答メッセージを受信した時点で、*RecvRepWriteWordBlock* イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 のワードブロックデータ（先頭アドレス=16#10000 から 2 ワード）の書込み要求を行う。

```
void SampleReqWriteWordBlock( ){
    unsigned long ulData[2];
    ulData[0] = 0x1234;
    ulData[1] = 0x5678;
    // ワードブロックライト要求
    long lRet = HFA_ReqWriteWordBlock(1, 0x10000, 2, (unsigned char *)ulData);
}

// ワードブロックライト応答受信コールバック関数
void CALLBACK RecvRepWriteWordBlock(long NodeNo, long Result, unsigned long Addr, long Words,
        long ErrBytes, unsigned char *Error){
    if(Result == 0){                    // 正常応答の場合
        printf("ワードブロックライト完了¥n");
    }
    else if(Result == 1){               // 異常応答の場合
        char cError[1025];
        memset(cError, 0, sizeof(cError));
        memcpy(cError, Error, ErrBytes);                // エラーコード待避
        printf("ワードブロックライト異常発生。[エラー内容=%s]¥n", cError);      // エラー表示
    }
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepWriteWordBlock* イベント

4.2.5. HFA_ReqReadNetParam

ネットワークパラメータリード要求

＜関数 I/F＞

long HFA_ReqReadNetParam (long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。

＜引数範囲＞

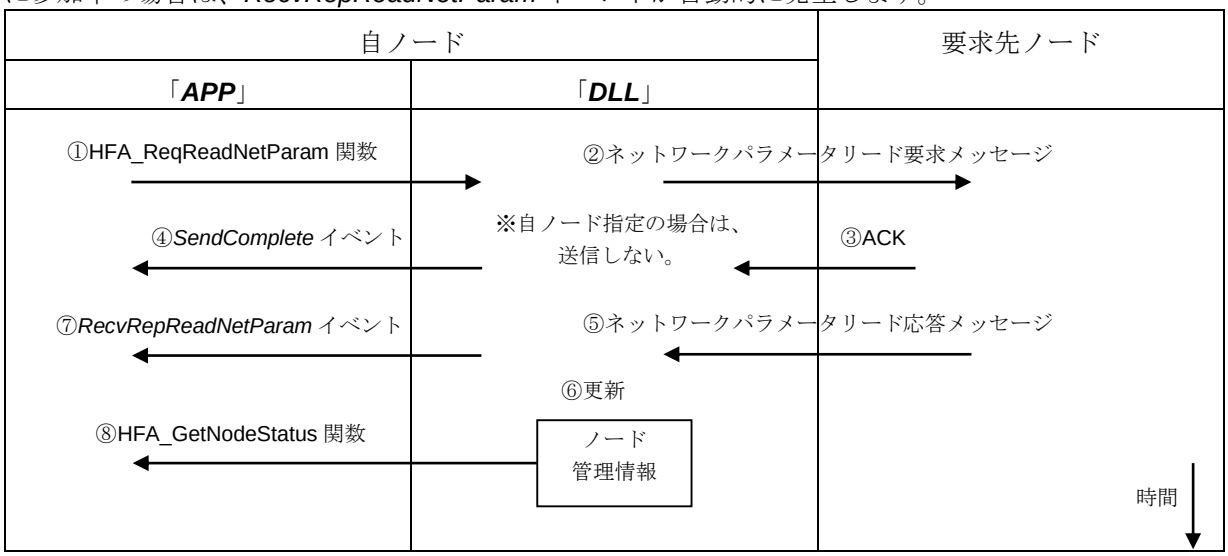
項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード可，255（=1 対 n メッセージ）不可

＜詳細＞

他ノードからネットワークパラメータの読出しを要求します。引数で指定された条件に基づき、ネットワークパラメータリード要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。

要求先ノードからネットワークパラメータリード応答メッセージを受信した時点で、*RecvRepReadNetParam* イベントが発生します。イベント発生後に、HFA_GetNodeStatus 関数をコールすることで、ネットワークパラメータの最新情報を取得することが可能となります。

NodeNo=自ノードを指定した場合、メッセージの送信は行いません。また、自ノードが FL-net ネットワークに参加中の場合は、*RecvRepReadNetParam* イベントが自動的に発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 のネットワークパラメータの読出し要求を行う。

```
void SampleReqReadNetParam( ){
    long lRet = HFA_ReqReadNetParam(1);              // ネットワークパラメータリード要求
}
```

２）ネットワークパラメータのリード応答受信時に、応答ノードのノード情報を取得する。

```
void CALLBACK RecvRepReadNetParam(long NodeNo, long Result, long ErrBytes, unsigned char *Error){
    if(Result == 0){
        long lNodeNo = NodeNo;
        NODE xNode;
        if(HFA_GetNodeStatus(&lNodeNo, &xNode) == 0){          // 応答ノードの管理情報パラメータ読出し
            printf("ノード番号=%d のノード名称=%s¥n", xNode.NodeName);          // ノード名称を参照
        }
    }
}
```

＜関連事項＞

・HFA_GetNodeStatus 関数

・*SendComplete* イベント，*RecvRepReadNetParam* イベント

4.2.6. HFA_ReqWriteNetParam

ネットワークパラメータライト要求

＜関数 I/F＞

long HFA_ReqWriteNetParam (long NodeNo, long Common1Addr, long Common1Bytes,
long Common2Addr, long Common2Words, char *NodeName, long SetMask)

＜引数＞

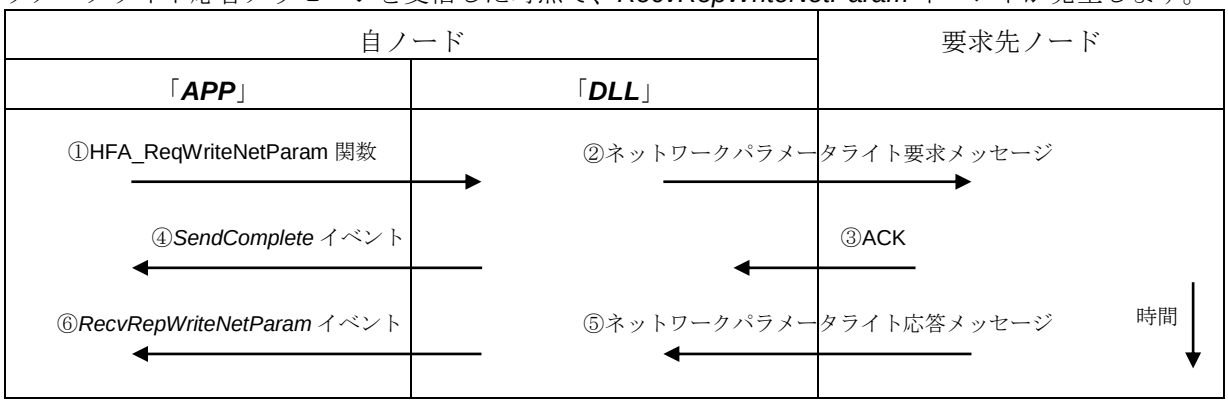
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
long	Common1Addr	IN	コモンメモリ１アドレス（バイト単位）。
long	Common1Bytes	IN	コモンメモリ１バイト数（バイト単位）。
long	Common2Addr	IN	コモンメモリ２アドレス（ワード単位）。
long	Common2Words	IN	コモンメモリ２ワード数（ワード単位）。
char *	NodeName	IN	ノード名称。10 バイト以内で NULL 終端とします。10 バイト以上を指定した場合は 10 バイトのみとし、11 バイト目以降は無視します。
long	SetMask	IN	設定マスク。設定する項目に応じて以下の値を論理和演算で指定します。 ・ 0x00000001：ノード名称の設定 ・ 0x00000002：アドレスの設定

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
Common1Addr	0≦（値）≦0x3FF	偶数のみ。
Common1Bytes	0≦（値）≦0x400	偶数のみ。
Common2Addr	0≦（値）≦0x1FFF	
Common2Words	0≦（値）≦0x2000	
Common1Addr+Common1Bytes	（値）≦0x400	
Common2Addr+Common2Words	（値）≦0x2000	

＜詳細＞

他ノードに対し、ネットワークパラメータの書込みを要求します。引数に基づき、ネットワークパラメータライト要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからネットワークパラメータライト応答メッセージを受信した時点で、*RecvRepWriteNetParam* イベントが発生します。



なお、他ノードよりネットワークパラメータライト要求を受信した場合は、「***DLL***」は自動的に非実装応答を返信し、ネットワークパラメータを変更しません。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 にネットワークパラメータ（ノード名称=”Sample”）の書込み要求を行う。

```
void SampleReqWriteNetParam1( ){
    // ネットワークパラメータライト要求
    long lRet = HFA_ReqWriteNetParam(1, 0, 0, 0, 0, “Sample”, 1);
}
```

２）ノード番号=1 にネットワークパラメータ（コモンメモリ領域［領域 1 先頭アドレス=0(バイト)，領域 1 サイズ=10(バイト)，領域 2 先頭アドレス=100(ワード)，領域 2 サイズ=50(ワード)］，ノード名称=”Sample”）の書込み要求を行う。

```
void SampleReqWriteNetParam2( ){
    // ネットワークパラメータライト要求
    long lRet = HFA_ReqWriteNetParam(1, 0, 10, 100, 50, “Sample”, 3);
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepWriteNetParam* イベント

4.2.7. HFA_ReqControlEquipment

運転／停止指令要求

＜関数 I/F＞

long HFA_ReqControlEquipment (long NodeNo, long Command)

＜引数＞

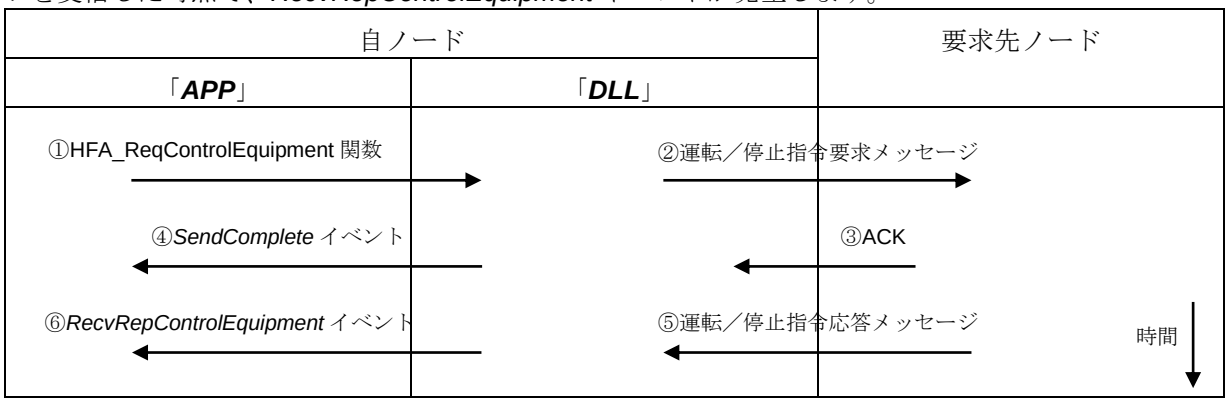
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
long	Command	IN	要求する制御内容。 ・ 0＝停止指令 ・ 0≠運転指令

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可

＜詳細＞

他ノードに対し、運転／停止指令を要求します。引数で指定された条件に基づき、運転／停止指令要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードから運転／停止指令応答メッセージを受信した時点で、*RecvRepControlEquipment* イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=100 に運転指令の要求を行う。

```
void SampleReqControlEquipment( ){
      long lRet = HFA_ReqControlEquipment(100, 1);                  // 運転指令要求
}

// 運転/停止指令応答受信コールバック関数
void CALLBACK RecvRepControlEquipment (long NodeNo, long Command, long Result, long ErrBytes,
            unsigned char *Error){
      if(Result == 0){              // 正常応答の場合
            printf(“運転指令正常¥n”);       // エラー表示
      }
      else if(Result == 1){       // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);                        // エラーコード待避
            printf(“運転指令異常[エラー内容=%s]¥n”, cError);          // エラー表示
      }
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepControlEquipment* イベント

4.2.8. HFA_ReqReadProfile

プロファイルリード要求

＜関数 I/F＞

long HFA_ReqReadProfile (long NodeNo)

＜引数＞

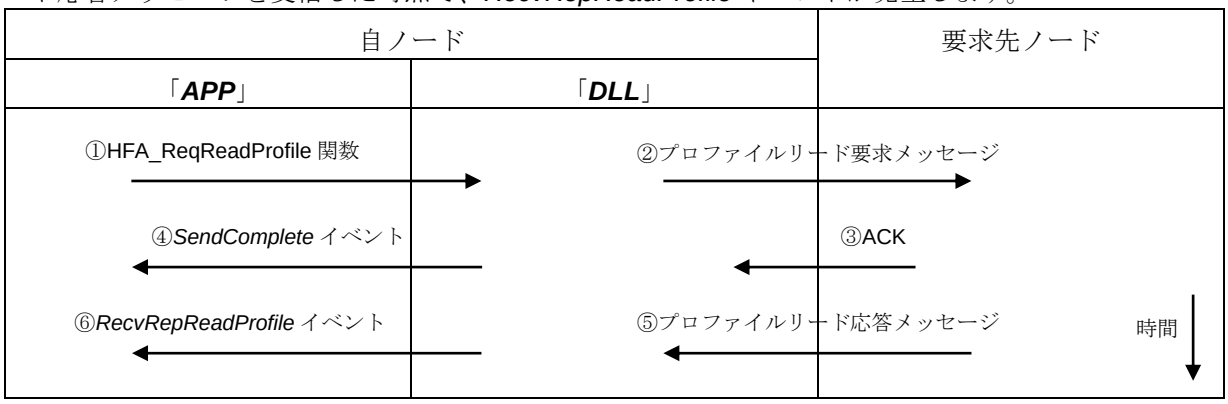
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可

＜詳細＞

他ノードからプロファイルデータの読出しを要求します。引数で指定された条件に基づき、プロファイルリード要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからプロファイルリード応答メッセージを受信した時点で、*RecvRepReadProfile* イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜関連事項＞

・*SendComplete* イベント，*RecvRepReadProfile* イベント

4.2.9. HFA_ReqReadLog

ログデータリード要求

＜関数 I/F＞

long HFA_ReqReadLog (long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。

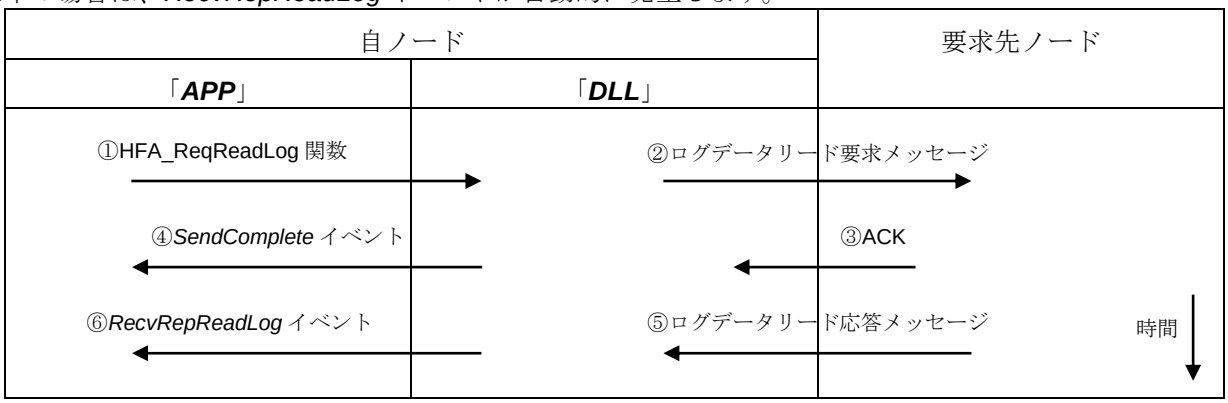
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード可，255（=1 対 n メッセージ）不可

＜詳細＞

他ノードからログデータの読出しを要求します。引数で指定された条件に基づき、ログデータリード要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからログデータリード応答メッセージを受信した時点で、*RecvRepReadLog* イベントが発生します。

NodeNo=自ノードを指定した場合、メッセージの送信は行いません。自ノードが FL-net ネットワークに参加中の場合は、*RecvRepReadLog* イベントが自動的に発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜関連事項＞

・*SendComplete* イベント，*RecvRepReadLog* イベント

4.2.10. HFA_ReqClearLog

ログデータクリア要求

＜関数 I/F＞

long HFA_ReqClearLog(long NodeNo);

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。

＜引数範囲＞

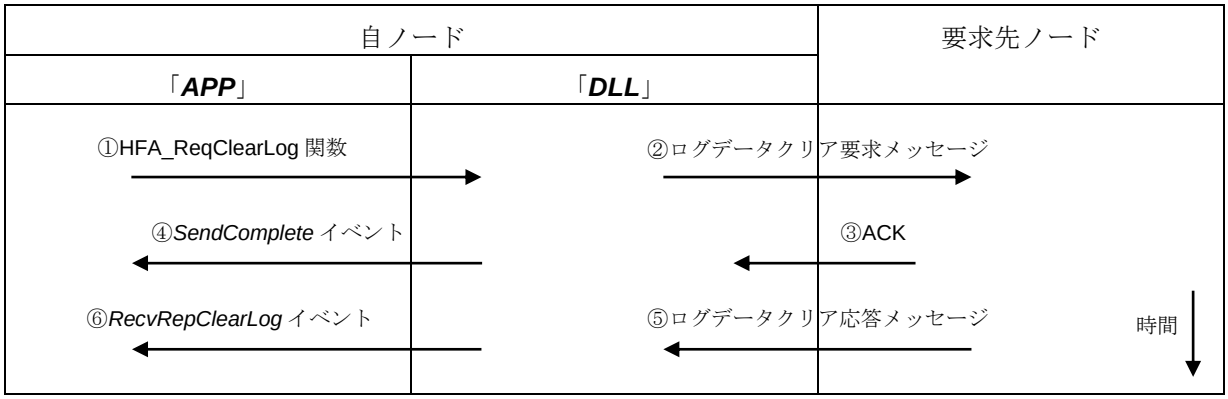
項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦255	自ノード可，255（=1 対 n メッセージ）指定可

＜詳細＞

他ノードに対し、ログデータのクリアを要求します。引数で指定された条件に基づき、ログデータクリア要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからログデータクリア応答メッセージを受信した時点で、*RecvRepClearLog* イベントが発生します。

NodeNo 引数に 255（1 対 n メッセージ）を指定した場合は、全ノードに対しログデータクリアを要求します。この場合、*SendComplete* イベントは発生しますが、*RecvRepClearLog* イベントは発生しません。

NodeNo 引数に自ノード番号または 255 を指定した場合、FL-net ネットワーク参加状態に関わらず自ノードのログ情報をクリアします。なお、自ノードが FL-net ネットワークに参加中の場合は、*RecvRepClearLog* イベントが自動的に発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜関連事項＞

・*SendComplete* イベント，*RecvRepClearLog* イベント

4.2.11. HFA_ReqEchoMessage

メッセージ折り返し要求

＜関数 I/F＞

long HFA_ReqEchoMessage (long NodeNo, long Bytes, unsigned char *SendData)

＜引数＞

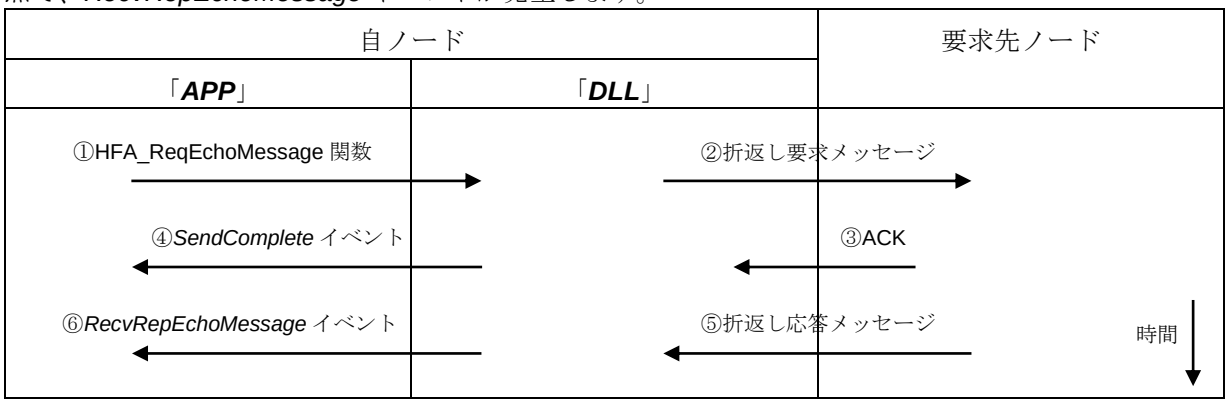
型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
long	Bytes	IN	送信データバイト数。
unsigned char *	SendData	IN	送信テストデータの先頭ポインタ。 **注）呼び出し元「*APP*」では、必ず送信データバイト数以上の領域を確保してください。**

＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
Bytes	1≦（値）≦1024	0 バイト不可

＜詳細＞

他ノードに対し、メッセージ折返しを要求します。引数で指定された条件に基づき、折返し要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードから折返し応答メッセージを受信した時点で、*RecvRepEchoMessage* イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 に折返しメッセージ要求（テキストデータ=”12345”）を送信する。

```
void SampleReqEchoMessage( ){
    long lRet = HFA_ReqEchoMessage(1, 5, “12345”);          // 折返しメッセージ要求
}

// 折返しメッセージ応答コールバック関数で、応答データのチェックを行う。
void CALLBACK RecvRepEchoMessage(long NodeNo, long Bytes, unsigned char *Message){
    if(NodeNo == 1 && Bytes == 5){              // 応答ノード番号，バイト数チェック
        if(strcmp(Message, “12345”) == 0){      // 応答データ内容チェック
            printf(“メッセージ折返し内容=OK”);          // 内容一致
        }
    }
}
```

２）ノード番号=100 に折返しメッセージ要求（バイナリデータ）を送信する。

```
void SampleReqEchoMessage2( ){
    unsigned char ucData[1024];
    ucData[0] = 0xFF;                   // バイナリデータ設定
     :
    long lRet = HFA_ReqEchoMessage(100, 1024, ucData);      // 折返しメッセージ要求
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepEchoMessage* イベント

4.2.12. HFA_RepReadByteBlock

バイトブロックリード応答

＜関数 I/F＞

long HFA_RepReadByteBlock (long NodeNo, long Result, unsigned long StartAddr, long Bytes,
unsigned char *Data, long ErrBytes, unsigned char *Error)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答先ノード番号。
long	Result	IN	応答結果。バイトブロックリード要求内容（開始アドレス，データサイズ）を「***APP***」で判断した結果を指定してください。 ・ 0＝正常 ・ 1＝異常 ・ 2＝非実装
unsigned long	StartAddr	IN	仮想ブロック空間の開始アドレス（バイト単位）。 **注）要求内容の開始アドレスと同一の値を指定してください。**
long	Bytes	IN	仮想ブロック空間のデータサイズ（バイト単位）。 **注）要求内容のデータサイズと同一の値を指定してください。**
unsigned char *	Data	IN	応答バイトデータの先頭ポインタ。 応答結果が正常の場合、値が有効となります。 **注）応答結果が正常の場合、呼び出し元「*APP*」では必ず仮想ブロック空間のデータサイズ以上の領域を確保してください。**
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答結果=異常の場合、値が有効となります。

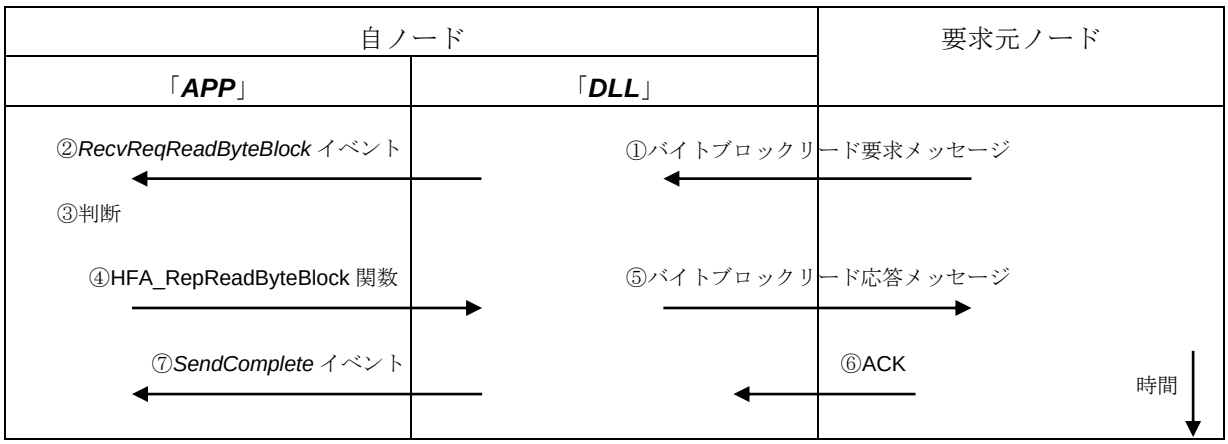
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Bytes	1≦（値）≦1024	
StartAddr+Bytes	（値）≦0x100000000	
ErrBytes	0≦（値）≦1024	応答結果=異常の場合のみ。

＜詳細＞

他ノードからのバイトブロックリード要求に対する応答を行います。バイトブロックリード要求メッセージを受信した時点で、*RecvReqReadByteBlock* イベントが発生します。要求内容（開始アドレスおよびバイト数など）を「***APP***」で判断し、本関数をコールしてください。

引数で指定された条件に基づき、バイトブロックリード応答メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	応答先ノード未参加
6	送信ビジー（「**DLL**」内に送信中のメッセージが残っている）

＜使用例＞

1）バイトブロックリード要求受信時の処理を行う。

```
void CALLBACK RecvReqReadByteBlock(long NodeNo, long Addr, long Bytes){
    long lRet;
    if (…) {          // 通知される引数情報を判断
        // バイトブロックリード応答（正常応答）
        lRet = HFA_RepReadByteBlock(NodeNo, 0, Addr, Bytes, &G_ByteBlock[Addr], 0, 0);
    }
    else{
        unsigned char ucError=1;     // エラーコード=1
        // バイトブロックリード応答（異常応答）
        lRet = HFA_RepReadByteBlock(NodeNo, 1, Addr, Bytes, 0, 1, &ucError);
    }
}
```

＜関連事項＞

・*RecvReqReadByteBlock* イベント，*SendComplete* イベント

4.2.13. HFA_RepWriteByteBlock

バイトブロックライト応答

＜関数 I/F＞

long HFA_RepWriteByteBlock (long NodeNo, long Result, unsigned long StartAddr, long Bytes,
long ErrBytes, unsigned char *Error)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答先ノード番号。
long	Result	IN	応答結果。バイトブロックライト要求内容（開始アドレス，データサイズ，データ内容）を「***APP***」側で判断した結果を指定してください。 ・ 0＝正常 ・ 1＝異常 ・ 2＝非実装
unsigned long	StartAddr	IN	仮想ブロック空間の開始アドレス（バイト単位）。 **注）要求内容の開始アドレスと同一の値を指定してください。**
long	Bytes	IN	仮想ブロック空間のデータサイズ（バイト単位）。 **注）要求内容のデータサイズと同一の値を指定してください。**
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答結果=異常の場合、値が有効です。

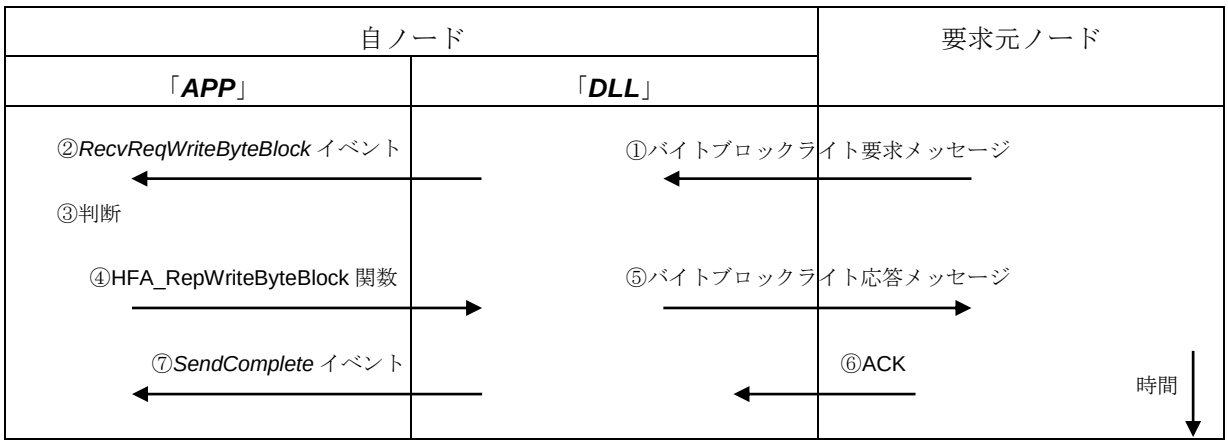
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Bytes	1≦（値）≦1024	
StartAddr+Bytes	（値）≦0x100000000	
ErrBytes	0≦（値）≦1024	応答結果=異常の場合のみ

＜詳細＞

他ノードからのバイトブロックライト要求に対する応答を行います。バイトブロックライト要求メッセージを受信した時点で、*RecvReqWriteByteBlock* イベントが発生します。要求内容（開始アドレスおよびバイト数など）を「***APP***」で判断し、本関数をコールしてください。

引数で指定された条件に基づき、バイトブロックライト応答メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	応答先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）バイトブロックライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteByteBlock(long NodeNo, long Addr, long Bytes, unsigned char *Data){
    long lRet;
    if (…) {          // 通知される引数情報を判断
        memcpy(&G_ByteBlock[Addr], Data, Bytes);                    // バイトブロックデータの更新
        // バイトブロックライト応答（正常応答）
        lRet = HFA_RepWriteByteBlock(NodeNo, 0, Addr, Bytes, 0, 0);
    }
    else{
        unsigned char ucError=1;      // エラーコード=1
        // バイトブロックライト応答（異常応答）
        lRet = HFA_RepWriteByteBlock(NodeNo, 1, Addr, Bytes, 1, &ucError);
    }
}
```

＜関連事項＞

・*RecvReqWriteByteBlock* イベント，*SendComplete* イベント

4.2.14. HFA_RepReadWordBlock

ワードブロックリード応答

＜関数 I/F＞

long HFA_RepReadWordBlock (long NodeNo, long Result, unsigned long StartAddr, long Words,
unsigned long *Data, long ErrBytes, unsigned char *Error)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答先ノード番号。
long	Result	IN	応答結果。ワードブロックリード要求内容（開始アドレス，データサイズ）を「***APP***」で判断した結果を指定してください。 ・ 0＝正常 ・ 1＝異常 ・ 2＝非実装
unsigned long	StartAddr	IN	仮想ブロック空間の開始アドレス（ワード単位）。 **注）要求内容の開始アドレスと同一の値を指定してください。**
long	Words	IN	仮想ブロック空間のデータサイズ（ワード単位）。 **注）要求内容のデータサイズと同一の値を指定してください。**
unsigned char *	Data	IN	応答ワードデータの先頭ポインタ。応答結果=正常の場合、値が有効です。 **注）応答結果が正常の場合、呼び出し元「*APP*」では必ず仮想ブロック空間のデータサイズ以上の領域を確保してください。「*APP*」側で管理しているワード単位（=2 バイト単位）のデータを(unsigned char *)型に型変換して指定してください。**
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答結果=異常の場合、値が有効です。

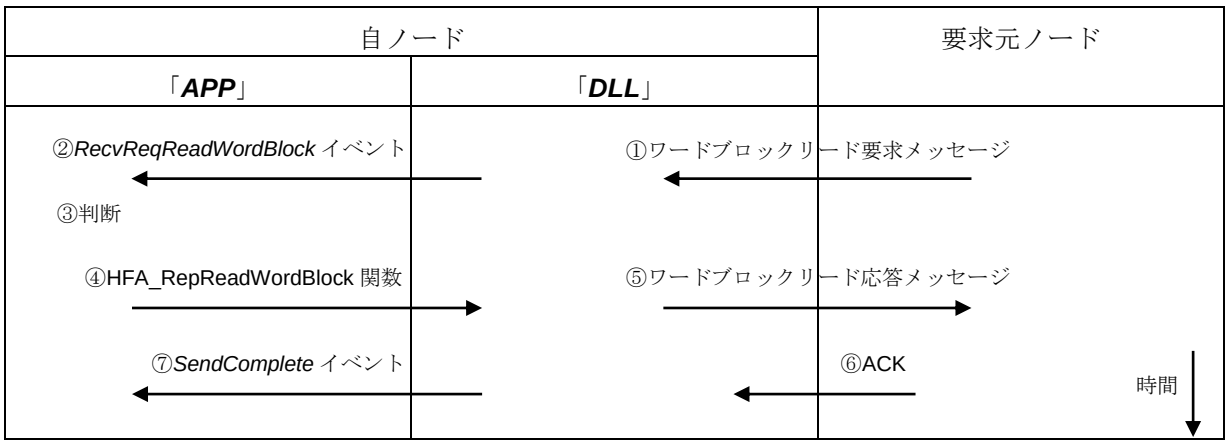
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Words	1≦（値）≦512	
StartAddr+Words	（値）≦0x100000000	
ErrBytes	0≦（値）≦1024	応答結果=異常の場合のみ

＜詳細＞

他ノードからのワードブロックリード要求に対する応答を行います。ワードブロックリード要求メッセージを受信した時点で、*RecvReqReadWordBlock* イベントが発生します。要求内容（開始アドレスおよびワード数など）を「***APP***」で判断し、本関数をコールしてください。

引数で指定された条件に基づき、ワードブロックリード応答メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	応答先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ワードブロックの読出し要求受信時の処理を行う。

```
void CALLBACK RecvReqReadWordBlock(long NodeNo, long Addr, long Words){
    long lRet;
    if (…) {          // 通知される引数情報を判断
        // ワードブロックリード応答（正常応答）
        lRet = HFA_RepReadWordBlock(NodeNo, 0, Addr, Words,
            (unsigned char *)&G_WordBlock[Addr], 0, 0);
    }
    else{
        unsigned char ucError=1;     // エラーコード=1
        // ワードブロックリード応答（異常応答）
        lRet = HFA_RepReadWordBlock(NodeNo, 1, Addr, Words, 0, 1, &ucError);
    }
}
```

＜関連事項＞

・*RecvReqReadWordBlock* イベント，*SendComplete* イベント

4.2.15. HFA_RepWriteWordBlock

ワードブロックライト応答

＜関数 I/F＞

long HFA_RepWriteWordBlock (long NodeNo, long Result, unsigned long StartAddr, long Words, long ErrBytes, unsigned char *Error)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答先ノード番号。
long	Result	IN	応答結果。ワードブロックライト要求内容（開始アドレス，データサイズ，データ内容）を「***APP***」で判断した結果を指定してください。 ・ 0＝正常 ・ 1＝異常 ・ 2＝非実装
unsigned long	StartAddr	IN	仮想ブロック空間の開始アドレス（ワード単位）。 **注）要求内容の開始アドレスと同一の値を指定してください。**
long	Bytes	IN	仮想ブロック空間のデータサイズ（ワード単位）。 **注）要求内容のデータサイズと同一の値を指定してください。**
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答結果=異常の場合、値が有効です。

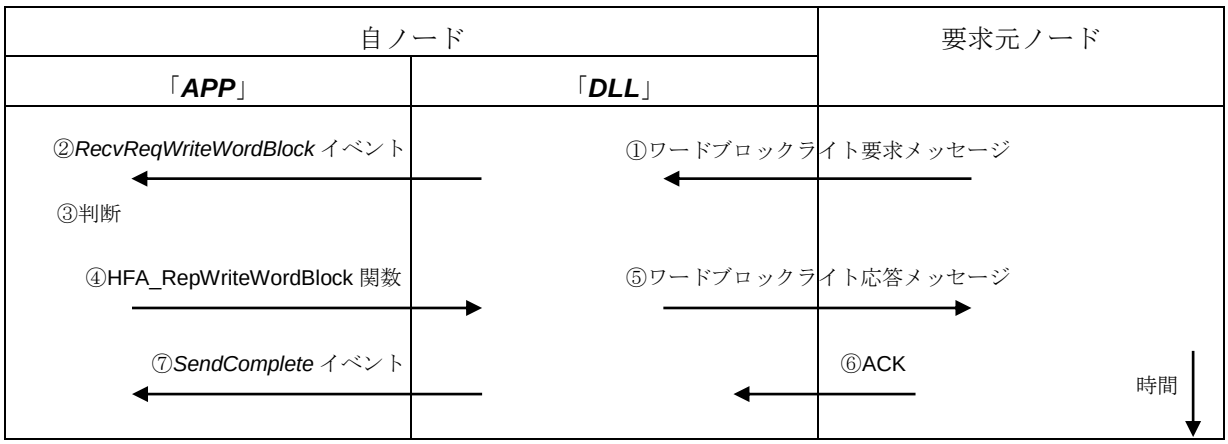
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
StartAddr	0≦（値）≦0xFFFFFFFF	
Words	1≦（値）≦512	
StartAddr+Words	（値）≦0x100000000	
ErrBytes	0≦（値）≦1024	応答結果=異常の場合のみ

＜詳細＞

他ノードからのワードブロックライト要求に対する応答を行います。ワードブロックライト要求メッセージを受信した時点で、*RecvReqWriteWordBlock* イベントが発生します。要求内容（開始アドレスやデータサイズなど）を「***APP***」で判断し、本関数をコールしてください。

引数で指定された条件に基づき、ワードブロックライト応答メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	応答先ノード未参加
6	送信ビジー（「**DLL**」内に送信中のメッセージが残っている）

＜使用例＞

１）ワードブロックライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteWordBlock(long NodeNo, long Addr, long Words, unsigned char *Data){
    long lRet;
    if (…) {          // 通知される引数情報を判断
        memcpy(&G_WordBlock[Addr], Data, Words * 2);          // ワードブロックデータの更新
        // ワードブロックライト応答（正常応答）
        lRet = HFA_RepWriteWordBlock(NodeNo, 0, Addr, Words, 0, 0);
    }
    else{
        unsigned char ucError=1;
        // ワードブロックライト応答（異常応答）
        lRet = HFA_RepWriteWordBlock(NodeNo, 1, Addr, Words, 1, &ucError);
    }
}
```

＜関連事項＞

・*RecvReqWriteWordBlock* イベント，*SendComplete* イベント

4.2.16. HFA_RepWriteNetParam

ネットワークパラメータライト応答メッセージ

＜関数 I/F＞

long HFA_RepWriteNetParam (long NodeNo, long Result, long ErrBytes, unsigned char * Error)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答先ノード番号。
long	Result	IN	応答結果。ネットワークパラメータライト要求内容を「***APP***」で判断した結果を指定してください。 ・ 0＝正常 ・ 1＝異常 ・ 2＝非実装
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答結果=異常の場合、値が有効です。

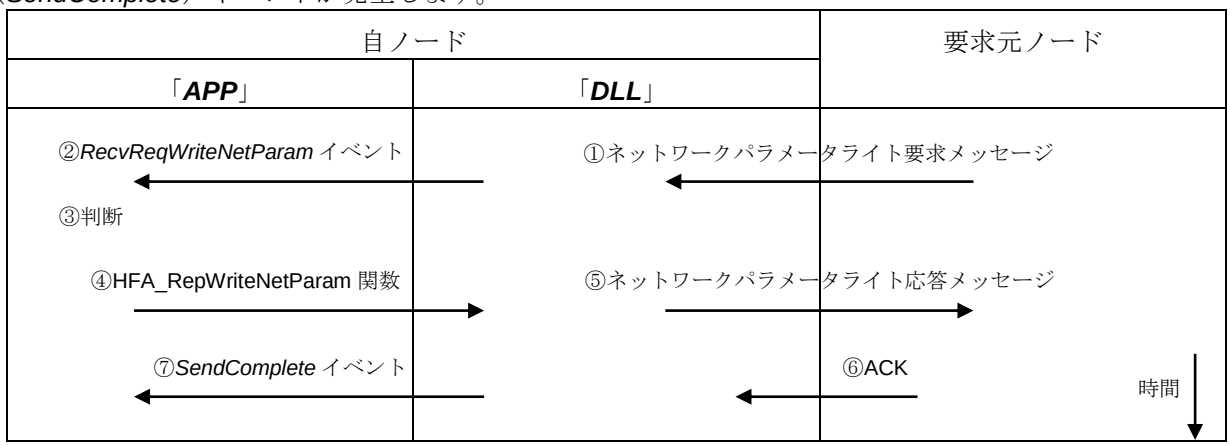
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
ErrBytes	0≦（値）≦1024	応答結果=異常の場合のみ

＜詳細＞

他ノードからのネットワークパラメータライト要求に対する応答を行います。ネットワークパラメータライト要求メッセージを受信した時点で、*RecvReqWriteNetParam* イベントが発生します。要求内容（コモンメモリアドレス及びサイズなど）を「***APP***」で判断し、本関数をコールしてください。

引数で指定された条件に基づき、ネットワークパラメータライト応答メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加(モニタモード含む)
3	応答先ノード未参加
6	送信ビジー(「***DLL***」内に送信中のメッセージが残っている)

＜使用例＞

1）ネットワークパラメータライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteNetParam(long NodeNo, long Common1Addr, long Common1Bytes,
        long Common2Addr, long Common2Words, char *NodeName, long SetMask){
    long lRet;
    if (…) {          // 通知される引数情報を判断
        // ネットワークパラメータライト応答(正常応答)
        lRet = HFA_RepWriteNetParam(NodeNo, 0, 0, 0);
    }
    else{
        unsigned char ucError=1;     // エラーコード=1
        // ネットワークパラメータライト応答(異常応答)
        lRet = HFA_RepWriteNetParam(NodeNo, 1, 1,   &ucError);
    }
}
```

＜関連事項＞

・*RecvReqWriteNetParam* イベント，*SendComplete* イベント

4.2.17. HFA_RepControlEquipment

運転/停止指令応答メッセージ

＜関数 I/F＞

long HFA_RepControlEquipment (long NodeNo, long Command, long Result, long ErrBytes,
unsigned char * Error)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答先ノード番号。
long	Command	IN	運転/停止指令 ・0=停止 ・0≠運転
long	Result	IN	応答結果。運転/停止指令要求内容を「***APP***」で判断した結果を指定してください。 ・ 0＝正常 ・ 1＝異常 ・ 2＝非実装
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答結果=異常の場合、値が有効です。

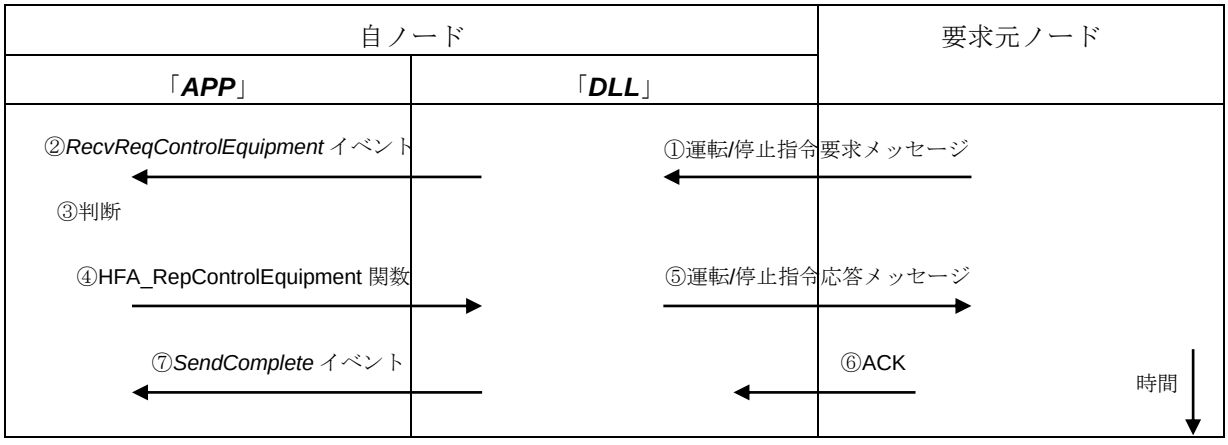
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
ErrBytes	0≦（値）≦1024	応答結果=異常の場合のみ

＜詳細＞

他ノードからの運転/停止指令要求に対する応答を行います。運転/停止指令要求メッセージを受信した時点で、*RecvReqControlEquipment* イベントが発生します。要求内容（指令内容）を「***APP***」で判断し、本関数をコールしてください。

引数で指定された条件に基づき、運転/停止指令応答メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加(モニタモード含む)
3	応答先ノード未参加
6	送信ビジー(「***DLL***」内に送信中のメッセージが残っている)

＜使用例＞

１）運転/停止指令要求受信時の処理を行う。

```
void CALLBACK RecvReqControlEquipment(long NodeNo, long Command){
    long lRet;
    if (Command == 0){      // 停止指令
        // 運転/停止指令応答(正常応答)
        lRet = HFA_RepControlEquipment(NodeNo, Command, 0, 0, 0);
    }
    else {                  // 運転指令
        unsigned char ucError=1;    // エラーコード=1
        // 運転/停止指令応答(異常応答)
        lRet = HFA_RepControlEquipment(NodeNo, Command, 1, 1, &ucError);
    }
}
```

＜関連事項＞

・*RecvReqControlEquipment* イベント，*SendComplete* イベント

4.2.18. HFA_SendTransparency

透過形メッセージ送信

＜関数 I/F＞

long HFA_SendTransparency (long NodeNo, long TransactionCode, long Bytes, unsigned char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	送信先ノード番号。
long	TransactionCode	IN	メッセージ番号。
long	Bytes	IN	送信データバイト数。
unsigned char *	Data	IN	送信データの先頭ポインタ。 注）呼び出し元「***APP***」では、バイト数以上の領域を確保してください。

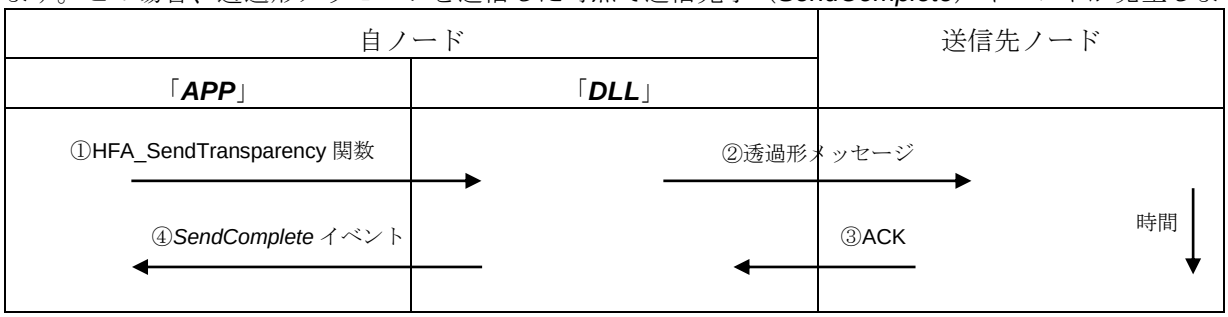
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦255	自ノード不可，255（=1 対 n メッセージ）指定可
TransactionCode	0≦（値）≦59999	0～9999 はリザーブ番号として予約されていますが、値の指定は可能です。この場合、引数異常とはならずに送信を行います。
Bytes	0≦（値）≦1024	

＜詳細＞

透過形メッセージの送信を行います。引数で指定された条件に基づき、透過形メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。

NodeNo 引数に 255（1 対 n メッセージ）を指定した場合は、全ノードに対し透過形メッセージの送信を行います。この場合、透過形メッセージを送信した時点で送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	送信先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 に透過形メッセージ（メッセージ番号=10000、テキストデータ=”12345”）を送信する。

```
void SampleSendTransparency1(){
      long lRet = HFA_SendTransparency(1, 10000, 5, “12345”);    // 1 対 1 透過形メッセージ送信
}
```

２）全ノードに透過形メッセージ（メッセージ番号=59999、バイナリデータ）を送信する。

```
void SampleSendTransparency2( ){
      unsigned char ucData[1024];
      ucData[0] = 0xFF;          // バイナリデータ設定
       :
      long lRet = HFA_SendTransparency(255, 59999, 1024, ucData);          // 1 対 n 透過形メッセージ送信
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvTransparency* イベント

4.2.19. HFA_ReqVendorMessage

ベンダ固有メッセージ要求

＜関数 I/F＞

long HFA_ReqVendorMessage (long NodeNo, char *VendorName, char *SubCode, long Bytes,
unsigned char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号。
char *	VendorName	IN	ベンダ名称。 **注）必ず 10 バイト以上のデータを指定してください。11 バイト目以降のデータは無視します。**
char *	SubCode	IN	サブコード。 **注）必ず 6 バイト以上のデータを指定してください。7 バイト目以降のデータは無視します。**
long	Bytes	IN	送信データバイト数。
unsigned char *	Data	IN	送信データの先頭ポインタ。 **注）呼び出し元「*APP*」では、バイト数以上の領域を確保してください。**

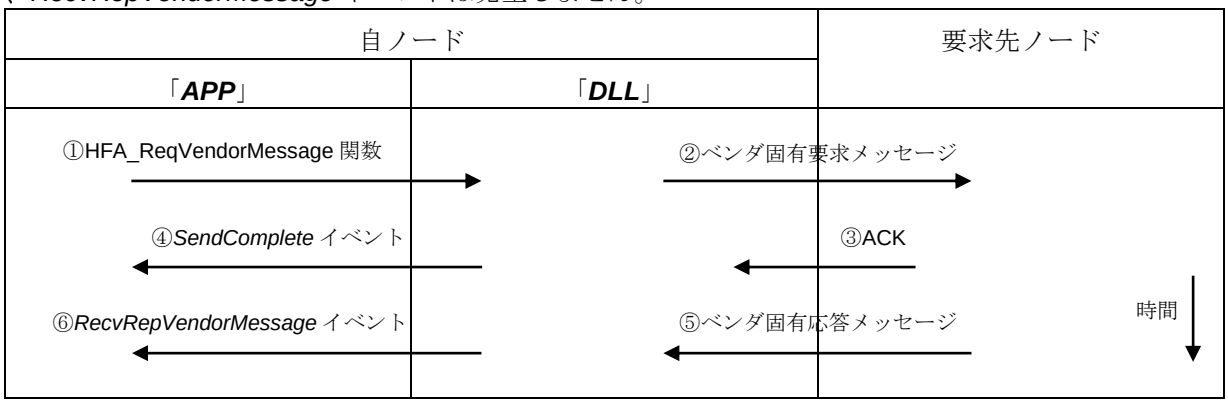
＜引数範囲＞

項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦255	自ノード不可，255（=1 対 n メッセージ）指定可
Bytes	0≦（値）≦1024	

＜詳細＞

他ノードに対し、ベンダ固有メッセージを要求します。引数で指定された条件に基づき、ベンダ固有要求メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。要求先ノードからベンダ固有応答メッセージを受信した時点で、*RecvRepVendorMessage* イベントが発生します。

NodeNo 引数に 255（1 対 n メッセージ）を指定した場合は、全ノードに対しベンダ固有要求メッセージの送信を行います。この場合、メッセージを送信した時点で送信完了（*SendComplete*）イベントが発生しますが、*RecvRepVendorMessage* イベントは発生しません。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	要求先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ノード番号=1 にベンダ固有要求メッセージ（ベンダ名=”Vendor”、サブコード=”000001”，テキストデータ=”12345”）を送信する。

```
void SampleReqVendorMessage( ){
    // 1 対 1 ベンダ固有要求メッセージ送信
    long lRet = HFA_ReqVendorMessage(1, “Vendor”, “000001”, 5, “12345”);
}
```

２）全ノードにベンダ固有メッセージ（バイナリデータ）を送信する。

```
void SampleReqVendorMessage2( ){
    char cScode[6];
    unsigned char ucData[1024];
    memset(cScode, 0, sizeof(cScode));
    cScode[5] = 1;
    ucData[0] = 0xFF;          // バイナリデータ設定
     :
    // 1 対 n ベンダ固有要求メッセージ送信
    long lRet = HFA_ReqVendorMessage(255, “Vendor”, cScode, 1024, cData);
}
```

＜関連事項＞

・*SendComplete* イベント，*RecvRepVendorMessage* イベント

4.2.20. HFA_RepVendorMessage

ベンダ固有メッセージ応答

＜関数 I/F＞

long HFA_RepVendorMessage (long NodeNo, long Result, char *VendorName, char *SubCode,
long Bytes, unsigned char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答先ノード番号。
long	Result	IN	応答結果。ベンダ固有要求メッセージ内容（ベンダ名，サブコードなど）を「***APP***」で判断した結果を指定してください。 ・ 0＝正常 ・ 1＝異常（ベンダ固有要求メッセージ内容のサブコードをサポートしていない場合など） ・ 2＝非実装（ベンダ固有要求メッセージ内容のベンダ名をサポートしていない場合など）
char *	VendorName	IN	ベンダ名称。 **注）要求内容のベンダ名称と同一の値を指定してください。必ず 10 バイト以上のデータを指定してください。**
char *	SubCode	IN	サブコード。 **注）要求内容のサブコードと同一の値を指定してください。必ず 6 バイト以上のデータを指定してください。**
long	Bytes	IN	送信データバイト数。
unsigned char *	Data	IN	送信データの先頭ポインタ。 応答結果に応じて、以下の内容を指定してください。 ・正常：正常データ ・異常：エラーコード ・非実装：指定データは無視されます。 **注）呼び出し元「*APP*」では、バイト数以上の領域を確保してください。**

＜引数範囲＞

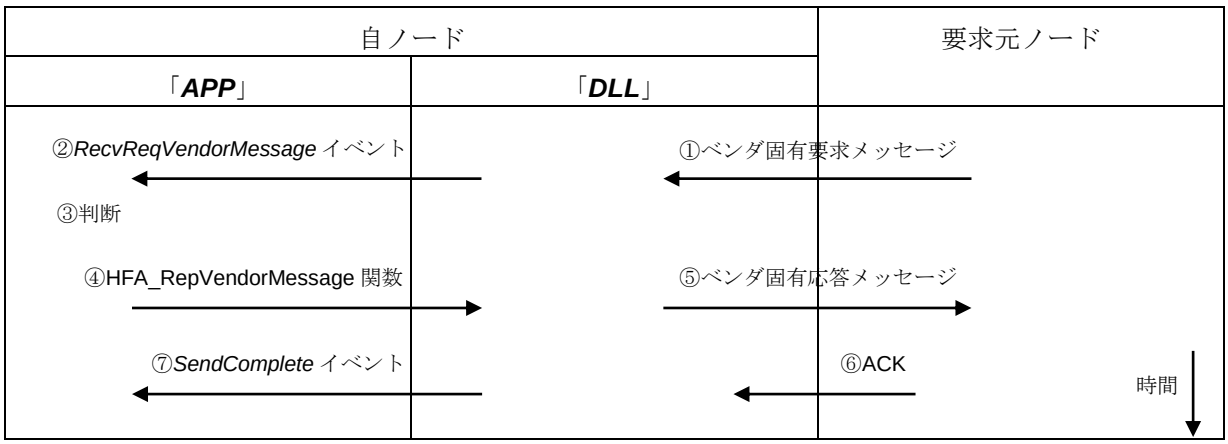
項 目（式）	値正常範囲	備 考
NodeNo	1≦（値）≦254	自ノード不可，255（=1 対 n メッセージ）不可
Bytes	0≦（値）≦1024	

＜詳細＞

他ノードから受信したベンダ固有要求メッセージに対する応答を行います。ベンダ固有要求メッセージを受信した時点で、*RecvReqVendorMessage* イベントが発生します。要求内容（ベンダ名およびサブコードなど）を「***APP***」で判断し、本関数をコールしてください。

注）ベンダ固有要求メッセージが 1 対 n メッセージの場合は、本関数をコールしないでください。

引数で指定された条件に基づき、ベンダ固有応答メッセージフレームを作成し送信します。本関数が正常終了するとメッセージ送信中の状態となり、メッセージ送信完了時に送信完了（*SendComplete*）イベントが発生します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモード含）
3	応答先ノード未参加
6	送信ビジー（「***DLL***」内に送信中のメッセージが残っている）

＜使用例＞

１）ベンダ固有要求メッセージ受信時の処理を行う。

```
void CALLBACK RecvReqVendorMessage(long NodeNo, long Multi, char *VendorName, char *SubCode,
        long Bytes, unsigned char *Data){
    long lRet;
    if (Multi == 0){ // 1 対 1 メッセージの場合のみ、応答する。
        if(…) {                 // 正常応答
            // ベンダ固有応答（正常応答）
            lRet = HFA_RepVendorMessage(NodeNo, 0, VendorName, SubCode, 4, “Data”);
        }
        else if(…){             // 異常応答（要求のサブコードをサポートしていない場合等）
            unsigned char ucError=1;
            // ベンダ固有応答（異常応答）
            lRet = HFA_RepVendorMessage(NodeNo, 1, VendorName, SubCode, 1, &ucError);
        }
        else{                   // 非実装応答（要求のベンダ名をサポートしていない場合等）
            // ベンダ固有応答（非実装応答）
            lRet = HFA_RepVendorMessage(NodeNo, 2, VendorName, SubCode, 0, 0);
        }
    }
}
```

＜関連事項＞

・*RecvReqVendorMessage* イベント，*SendComplete* イベント

4.3. DLL 制御関数

DLL 制御関数とは、「***DLL***」の動作を制御するための I/F 関数です。DLL 制御関数の一覧を以下に示します。

No.	関数名	概要
1	HFA_AttachLink	DLL へのアタッチ（DLL 使用開始）
2	HFA_DetachLink	DLL からのデタッチ（DLL 使用終了）
3	HFA_SetCallback	コールバック関数アドレス登録（FL-net Ver.2 専用）
4	HFA_SetCallbackV3	コールバック関数アドレス登録（FL-net Ver.3 対応版）
5	HFA_SetTimeout	タイムアウト値登録
6	HFA_DebugLog	デバッグ用ロギング出力モード設定

4.3.1. HFA_AttachLink

DLL へのアタッチ（DLL 使用開始）

＜関数 I/F＞

long HFA_AttachLink (LONG_PTR *LinkID)

＜引数＞

型	変 数	I/O	内 容
LONG_PTR	LinkID	OUT	メモリアクセス用識別子。

＜詳細＞

「***DLL***」使用を開始するための初期化処理を行います。

本関数が正常終了すると 1 以上の LinkID を返します。「***DLL***」内部メモリをアクセスする I/F 関数をコールする場合、本関数で取得した LinkID の識別子を指定してください。（「***DLL***」内で排他制御されます。）

「***DLL***」の使用を終了する場合は、本関数で取得した LinkID の識別子を指定して、HFA_DetachLink 関数をコールしてください。

本関数がコールされた時点では、FL-net ネットワークへの参加は行いません。ネットワークに参加する場合は、別途 HFA_LinkIn 関数をコールしてください。

注）「*DLL*」使用開始時には、本関数を必ずコールしてください。本関数をコールせずに「*DLL*」のその他 I/F 関数をコールした場合、動作が不安定になる場合があります。

＜戻り値＞

値	内 容
0	正常終了
-100	システムエラー。以下の原因が考えられます。 ・メモリ不足

＜使用例＞

1）メイン関数で DLL の使用を開始する。

```
long G_LinkID;      // メモリアクセス用識別子（グローバル変数）
void main( ){
    long lRet = HFA_AttachLink(&G_LinkID);    // DLL 使用開始
}
```

＜関連事項＞

・HFA_DetachLink 関数，HFA_LinkIn 関数

4.3.2. HFA_DetachLink

DLL からのデタッチ（DLL 使用終了）

＜関数 I/F＞

long HFA_DetachLink (LONG_PTR LinkID)

＜引数＞

型	変 数	I/O	内 容
LONG_PTR	LinkID	IN	「***DLL***」内部メモリをアクセスするための識別子。 HFA_AttachLink 関数で取得した値を指定してください。

＜詳細＞

HFA_AttachLink 関数で開始した「***DLL***」の使用を終了します。

FL-net ネットワーク参加中に本関数がコールされた場合は、FL-net ネットワークから離脱します。本関数コール後は、全てのイベントが通知されなくなります。また、本関数コール後に I/F 関数をコールした場合の動作保障はしません。

注）「*DLL*」使用終了時（「*APP*」終了時）には、本関数を必ずコールしてください。本関数をコールせずに「*APP*」を終了した場合、OS の動作が不安定になる場合があります。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常（未登録の LinkID）

＜使用例＞

１）DLL の使用を終了する。

```
void SampleProcessEnd( ){
    long lRet = HFA_DetachLink(G_LinkID);    // DLL 使用終了
}
```

＜関連事項＞

・HFA_AttachLink 関数

4.3.3. HFA_SetCallback

コールバック関数アドレス登録（FL-net Ver.2 専用）

＜関数 I/F＞

long HFA_SetCallback (LONG_PTR LinkID, long SetMask, CALLBACKFUNC *Callback)

＜引数＞

型	変 数	I/O	内 容
LONG_PTR	LinkID	IN	「***DLL***」内部メモリをアクセスするための識別子。 HFA_AttachLink 関数で取得した値を指定してください。
long	SetMask	IN	イベントを通知する項目を設定するマスクデータ。ビットの ON/OFF で設定する項目の登録更新有無を指定します。 ・ON=登録を更新する。 ・OFF=登録を更新しない。 ※詳細は、以下をご参照ください。
CALLBACKFUNC *	Callback	IN	コールバック登録テーブル[12]。イベント通知先のコールバック関数アドレスを、イベント毎に指定します。 ※詳細は、以下をご参照ください。

＜詳細＞

「***DLL***」で検知した非同期イベントを「***APP***」に通知する先のコールバック関数アドレス（FL-net Ver.2 専用）を登録します。FL-net Ver.3 対応のコールバック関数アドレスを登録する場合は、HFA_SetCallbakV3 関数を使用してください。

SetMask のビットが ON になっている項目のコールバック関数アドレスを登録し、ビットが OFF のアドレスについては変更しません。コールバック通知の登録を削除する場合は、マスク値に ON を設定し、コールバック登録テーブルのメンバ値に NULL を設定します。本関数がコールされない場合のデフォルト状態では、コールバック通知は行いません。

FL-net ネットワーク参加中（モニタモード含）に本関数をコールした場合、戻り値=参加中となり、登録の変更は行いません。

・マスクデータ（※登録を変更する項目のマスク値を論理和演算した結果を指定してください。）

マスク値	対応コールバック関数
0x00000001	LinkIn
0x00000002	LinkOut
0x00000004	CommonRefresh
0x00000008	LogClear
0x00000010	RecvReqReadByteBlock
0x00000020	RecvReqWriteByteBlock
0x00000040	RecvReqReadWordBlock
0x00000080	RecvReqWriteWordBlock
0x00000100	RecvRepReadByteBlock
0x00000200	RecvRepWriteByteBlock
0x00000400	RecvRepReadWordBlock
0x00000800	RecvRepWriteWordBlock

マスク値	対応コールバック関数
0x00004000	RecvRepControlEquipment
0x00008000	RecvRepReadProfile
0x00010000	RecvRepReadLog
0x00020000	RecvRepClearLog
0x00040000	RecvRepEchoMessage
0x00080000	RecvTransparency
0x00100000	LinkInTimeout
0x00200000	SendTimeout
0x00400000	ReplyTimeout
0x01000000	SendComplete
0x02000000	Error
0x10000000	RecvReqVendorMessage

[12] コールバック登録テーブルの構造体は、4.8.3 項をご参照ください。

0x00001000	RecvRepReadNetParam
0x00002000	RecvRepWriteNetParam

0x20000000	RecvRepVendorMessage

・コールバック登録テーブル

登録変数	コールバック発生条件
LinkIn	ノードが参加した
LinkOut	ノードが離脱した
CommonRefresh	コモンメモリの値が変化した
LogClear	自ノードのログデータがクリアされた
RecvReqReadByteBlock	バイトブロックリード要求メッセージを受信した
RecvReqWriteByteBlock	バイトブロックライト要求メッセージを受信した
RecvReqReadWordBlock	ワードブロックリード要求メッセージを受信した
RecvReqWriteWordBlock	ワードブロックライト要求メッセージを受信した
RecvRepReadByteBlock	バイトブロックリード応答メッセージを受信した
RecvRepWriteByteBlock	バイトブロックライト応答メッセージを受信した
RecvRepReadWordBlock	ワードブロックリード応答メッセージを受信した
RecvRepWriteWordBlock	ワードブロックライト応答メッセージを受信した
RecvRepReadNetParam	ネットワークパラメータリード応答メッセージを受信した
RecvRepWriteNetParam	ネットワークパラメータライト応答メッセージを受信した
RecvRepControlEquipment	運転／停止指令応答メッセージを受信した
RecvRepReadProfile	プロファイルリード応答メッセージを受信した
RecvRepReadLog	ログデータリード応答メッセージを受信した
RecvRepClearLog	ログデータクリア応答メッセージを受信した
RecvRepEchoMessage	メッセージ折り返し応答を受信した
RecvTransparency	透過形メッセージメッセージを受信した
LinkInTimeout	リンク参加タイムアウトが発生した
SendTimeout	メッセージ送信タイムアウトが発生した
ReplyTimeout	メッセージ応答受信タイムアウトが発生した
SendComplete	メッセージの送信が完了した（正常，異常）
Error	エラーが発生したとき
RecvReqVendorMessage	ベンダ固有要求メッセージを受信した
RecvRepVendorMessage	ベンダ固有応答メッセージを受信した

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常（LinkID 未登録）
4	自ノード参加中（モニタモード含）

＜使用例＞

１）ノード参加およびノード離脱イベント用のコールバック関数を登録する。

```
void SampleSetCallback1( ){
    long lRet;
    CALLBACKFUNC xFuncTbl;
    memset(&xFuncTbl, 0, sizeof(xFuncTbl));   // テーブル初期化
    xFuncTbl.LinkIn = CbLinkIn;               // ノード参加用コールバック関数アドレス設定
    xFuncTbl.LinkOut = CbLinkOut;             // ノード離脱用コールバック関数アドレス設定
    lRet = HFA_SetCallback(G_LinkID, 0x00000003, &xFuncTbl);// コールバック関数登録（SetMask=3）
}

// ノード参加用コールバック関数
void CALLBACK CbLinkIn(long NodeNo, long Reason){
    // ノード参加時の処理
}

// ノード離脱用コールバック関数
void CALLBACK CbLinkOut(long NodeNo, long Reason){
    // ノード離脱時の処理
}
```

２）コールバック関数の登録を全て解除する。

```
void SampleSetCallback2( ){
    long lRet;
    CALLBACKFUNC xFuncTbl;
    memset(&xFuncTbl, 0, sizeof(xFuncTbl));                    // テーブル初期化
    lRet = HFA_SetCallback(G_LinkID, 0xFFFFFFFF, &xFuncTbl);   // コールバック関数登録解除
}
```

＜関連事項＞

・HFA_AttachLink 関数，HFA_SetCallbackV3 関数

・コールバック関数

4.3.4. HFA_SetCallbackV3

コールバック関数アドレス登録（FL-net Ver.3 対応版）

＜関数 I/F＞

long HFA_SetCallbackV3(LONG_PTR　LinkID, HFA_CALLBACKFUNC_V3 *Callback)

＜引数＞

型	変 数	I/O	内 容
LONG_PTR	LinkID	IN	「***DLL***」内部メモリをアクセスするための識別子。 HFA_AttachLink 関数で取得した値を指定してください。
HFA_CALLBACKFUNC_V3 *	Callback	IN	コールバック登録テーブル[13]。イベント通知先のコールバック関数アドレスを、イベント毎に指定します。 メンバ値には以下の値を設定してください。 ・NULL=未登録または登録削除 ・コールバック関数アドレス=登録 ※詳細は、以下をご参照ください。

＜詳細＞

「***DLL***」で検知した同期・非同期イベントを「***APP***」に通知する先のコールバック関数アドレスを登録します。HFA_SetCallback 関数の拡張版で、FL-net Ver.2 および Ver.3 のコールバック関数を登録する場合に使用します。

コールバック通知の登録を削除する場合は、コールバック登録テーブルのメンバ値に NULL を設定します。本関数がコールされない場合のデフォルト状態では、コールバック通知は行いません。

FL-net ネットワーク参加中（モニタモード含）に本関数をコールした場合、戻り値=参加中となり、登録の変更は行いません。

HFA_SetCallback 関数と本関数を併用する場合は注意してください。HFA_SetCallback 関数で Ver.2 のコールバック関数を登録した後に本関数で Ver.3 の関数のみを登録した場合は、Ver.2 の登録は上書きされて未登録になります。

[13] コールバック登録テーブルの構造体は、4.8.14 項をご参照ください。

・コールバック登録テーブル

登録変数	コールバック発生条件
LinkIn	ノードが参加した
LinkOut	ノードが離脱した
CommonRefresh	コモンメモリの値が変化した
LogClear	自ノードのログデータがクリアされた
RecvReqReadByteBlock	バイトブロックリード要求メッセージを受信した
RecvReqWriteByteBlock	バイトブロックライト要求メッセージを受信した
RecvReqReadWordBlock	ワードブロックリード要求メッセージを受信した
RecvReqWriteWordBlock	ワードブロックライト要求メッセージを受信した
RecvRepReadByteBlock	バイトブロックリード応答メッセージを受信した
RecvRepWriteByteBlock	バイトブロックライト応答メッセージを受信した
RecvRepReadWordBlock	ワードブロックリード応答メッセージを受信した
RecvRepWriteWordBlock	ワードブロックライト応答メッセージを受信した
RecvRepReadNetParam	ネットワークパラメータリード応答メッセージを受信した
RecvRepWriteNetParam	ネットワークパラメータライト応答メッセージを受信した
RecvRepControlEquipment	運転／停止指令応答メッセージを受信した
RecvRepReadProfile	プロファイルリード応答メッセージを受信した
RecvRepReadLog	ログデータリード応答メッセージを受信した
RecvRepClearLog	ログデータクリア応答メッセージを受信した
RecvRepEchoMessage	メッセージ折り返し応答を受信した
RecvTransparency	透過形メッセージメッセージを受信した
LinkInTimeout	リンク参加タイムアウトが発生した
SendTimeout	メッセージ送信タイムアウトが発生した
ReplyTimeout	メッセージ応答受信タイムアウトが発生した
SendComplete	メッセージの送信が完了した（正常，異常）
Error	エラーが発生した
RecvReqVendorMessage	ベンダ固有要求メッセージを受信した
RecvRepVendorMessage	ベンダ固有応答メッセージを受信した
RecvReqWriteNetParam	ネットワークパラメータライト要求メッセージを受信した
RecvReqControlEquipment	運転/停止指令要求メッセージを受信した
LinkInSlave	スレーブノードが参加した
LinkOutSlave	スレーブノードが離脱した
InputDataRefresh	入力データが変化した
InputStatusRefresh	入力ステータスが変化した
ChangeTokenTimeMeasureStatus	トークン保持時間測定状態が変化した
ChangeDataLogMeasureStatus	汎用通信データ送信元ログ測定状態が変化した
RecvReqReadByteBlockFromSettingTool	設定ツールからのバイトブロックリード要求を受信した
RecvReqReadWordBlockFromSettingTool	設定ツールからのワードブロックリード要求を受信した
RecvReqWriteByteBlockFromSettingTool	設定ツールからのバイトブロックライト要求を受信した
RecvReqWriteWordBlockFromSettingTool	設定ツールからのワードブロックライト要求を受信した
RecvReqWriteNetParamFromSettingTool	設定ツールからのネットワークパラメータライト要求を受信した
RecvReqControlEquipmentFromSettingTool	設定ツールからの運転/停止指令要求を受信した
RecvReqSetIOFromSettingTool	設定ツールからの IO 割付設定要求を受信した
RecvReqSetConfigParamFromSettingTool	設定ツールからのコンフィギュレーション用パラメータ設定要求を受信した
RecvReqResetNodeFromSettingTool	設定ツールからのノードリセット要求を受信した

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常（LinkID 未登録）
4	自ノード参加中（モニタモード含）

＜使用例＞

１）ノード参加およびノード離脱イベント用のコールバック関数を登録する。

```
void SampleSetCallbackV3( ){
    HFA_CALLBACKFUNC_V3 xFuncTbl;
    memset(&xFuncTbl, 0, sizeof(xFuncTbl));   // テーブル初期化
    xFuncTbl.LinkIn = CbLinkIn;               // ノード参加用コールバック関数アドレス設定
    xFuncTbl.LinkOut = CbLinkOut;             // ノード離脱用コールバック関数アドレス設定
    long lRet = HFA_SetCallbackV3 (G_LinkID, &xFuncTbl);      // コールバック関数登録
}

// ノード参加用コールバック関数
void CALLBACK CbLinkIn(long NodeNo, long Reason){
    // ノード参加時の処理
}

// ノード離脱用コールバック関数
void CALLBACK CbLinkOut(long NodeNo, long Reason){
    // ノード離脱時の処理
}
```

２）コールバック関数の登録を全て解除する。

```
void SampleSetCallbackV3_2( ){
    HFA_CALLBACKFUNC_V3 xFuncTbl;
    memset(&xFuncTbl, 0, sizeof(xFuncTbl));                   // テーブル初期化
    long lRet = HFA_SetCallbackV3 (G_LinkID, &xFuncTbl);      // コールバック関数登録解除
}
```

＜関連事項＞

・HFA_AttachLink 関数，HFA_SetCallback 関数

・コールバック関数

4.3.5. HFA_SetTimeout

タイムアウト値登録

＜関数 I/F＞

long HFA_SetTimeout (long LinkIn, long Send, long Reply)

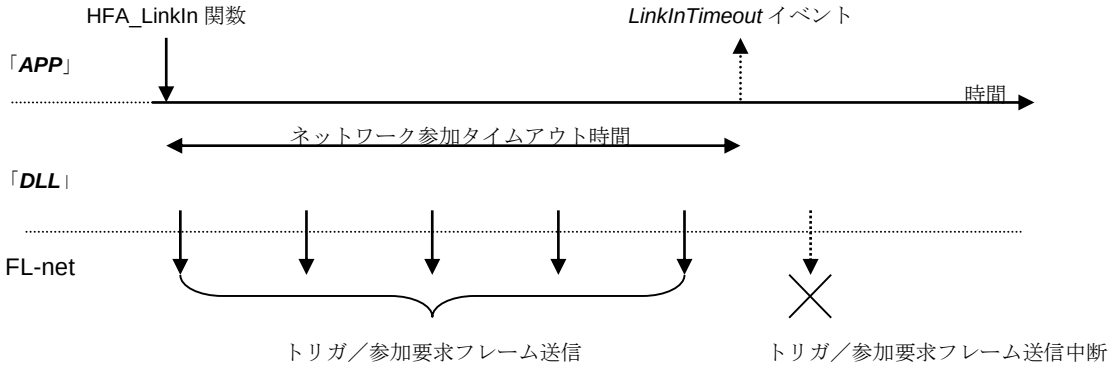
＜引数＞

型	変 数	I/O	内 容
long	LinkIn	IN	ネットワーク参加タイムアウト時間（ms 単位）。
long	Send	IN	メッセージ送信完了タイムアウト時間（ms 単位）。
long	Reply	IN	メッセージ応答受信タイムアウト時間（ms 単位）。

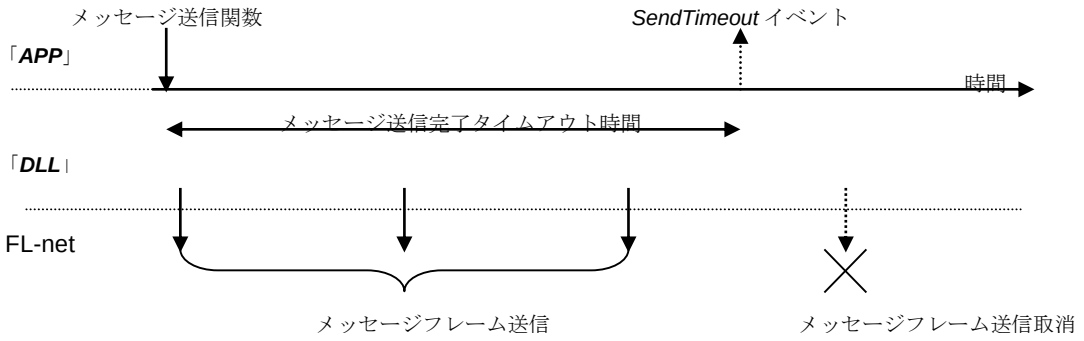
＜詳細＞

「***DLL***」内で使用するタイマのタイムアウト時間を登録します。各処理（ネットワーク参加処理，メッセージ送信処理，応答メッセージ受信処理）が登録された時間内に完了しない場合、それぞれに対応したタイムアウトイベントが発生します。

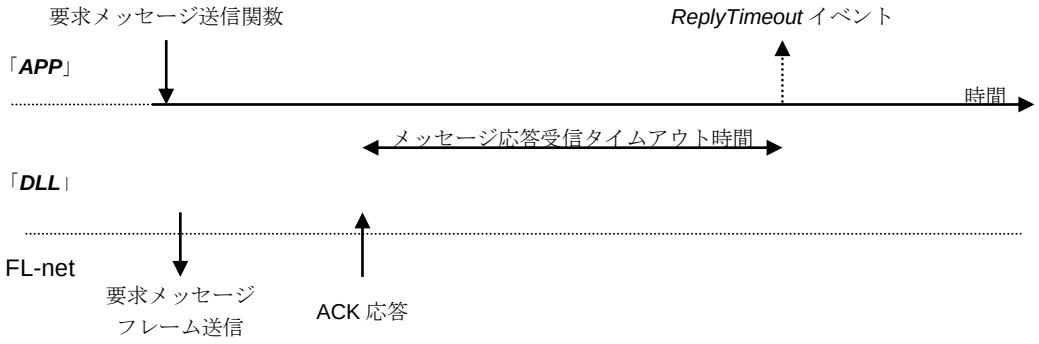
ネットワーク参加タイムアウト値は、ネットワークへの参加関数（HFA_LinkIn 関数）をコールしてからネットワークへ参加完了するまでの時間です。指定時間内に参加できない場合は、*LinkInTimeout* イベントが発生し、ネットワークへの参加を中断します。



送信完了タイムアウト値は、メッセージ送信関数をコールしてから送信先ノードより送達確認（ACK 応答）を受信するまでの時間です。指定時間内に ACK 応答を受信できない場合は、*SendTimeout* イベントが発生し、メッセージの送信を取り消します。



応答受信タイムアウト値は、要求メッセージ送信関数をコールし、要求先ノードより送達確認（ACK 正常応答）を受信してから、要求先ノードから対応する応答メッセージを受信するまでの時間です。指定時間内に応答メッセージを受信できない場合は、*ReplyTimeout* イベントが発生します。



タイムアウト時間が 0 の場合は、タイムアウト処理（タイムアウトによる処理の中断およびタイムアウトイベントの通知）がなくなります。本関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

引数に負の値を指定した場合は、タイムアウト時間の登録は行いません。なお、引数に短時間の値を設定した場合は、イベントの通知順序が発生順序と異なる場合があります。

FL-net ネットワーク参加中（モニタモード含）に本関数をコールした場合、戻り値=参加中となり設定の変更は行いません。

＜戻り値＞

値	内 容
0	正常終了
4	自ノード参加中（モニタモード含）

＜使用例＞

1）ネットワーク参加タイムアウト=無、メッセージ送信タイムアウト=3 秒に設定し、メッセージ応答タイムアウトは変更しない。

```
void SampleSetTimeout1(){
    long lRet = HFA_SetTimeout(0, 3000, -1);          // タイムアウト値登録
}
```

＜関連事項＞

・ HFA_LinkIn 関数，メッセージ送信関数

・ *LinkInTimeout* イベント，*SendTimeout* イベント，*ReplyTimeout* イベント

4.3.6. HFA_DebugLog

デバッグ用ロギング出力モード設定

＜関数 I/F＞

long HFA_DebugLog(long Logging)

＜引数＞

型	変 数	I/O	内 容
long	Logging	IN	ロギングを出力する内容を示す設定モード。設定する項目に応じて以下の値を論理和演算した結果を指定します。 ・ 0x00000001：関数コール正常結果（関数戻り値＝0）のロギング出力。 ・ 0x00000002：イベント通知時のロギング出力。 ・ 0x00000010：サイクリック通信のアナライズ出力。 ・ 0x00000020：メッセージ通信のアナライズ出力。 ・ 0x00000040：参加要求通信のアナライズ出力。 ※以下の値は、通常時は設定しないでください。 ・ 0x00000100：デバッグ情報の出力。 ・ 0x00001000：「***DLL***」管理情報種別ロギングの停止。 ・ 0x00002000：警告種別ロギングの停止。 ・ 0x00004000：エラー種別ロギングの停止。 ・ 0x00010000：関数コール警告結果（関数戻り値＞0）ロギングの停止 ・ 0x00020000：関数コールエラー結果（関数戻り値＜0）ロギングの停止

＜詳細＞

「***DLL***」の動作をロギングデータとして、イベントビューアに出力する項目を設定します。ロギングデータの出力項目は、種別（情報／警告／エラー）および出力タイミング（関数コール時／イベント発生時／通信時）に応じて分類されます。

ロギング中にロギングを開始したり、停止中に停止を行っても関数戻り値=正常終了となり、ロギングモードが更新されます。本関数がコールされない場合のデフォルト値は、6.2 節をご参照ください。

＜戻り値＞

値	内 容
0	正常終了

＜使用例＞

1）関数コール警告結果（関数戻り値＞0）のロギングを停止する。

```
void SampleDebugLog1( ){
    long lRet = HFA_DebugLog(0x10000);      // デバッグロギング出力モード設定
}
```

＜関連事項＞

なし

4.4. コマンドサーバ関数

コマンドサーバ関数とは、設定ツール対応コマンドサーバを起動・停止するための I/F 関数です。コマンドサーバ関数の一覧を以下に示します。

No.	関数名	概要
1	HFA_StartCmdServerUdp	コマンドサーバ(UDP)起動
2	HFA_StopCmdServerUdp	コマンドサーバ(UDP)停止
3	HFA_StartCmdServerTcp	コマンドサーバ(TCP)起動
4	HFA_StopCmdServerTcp	コマンドサーバ(TCP)停止
5	HFA_GetCmdServerUdpStatus	コマンドサーバ(UDP)の起動状態取得
6	HFA_GetCmdServerTcpStatus	コマンドサーバ(TCP)の起動状態取得

4.4.1. HFA_StartCmdServerUdp

コマンドサーバ(UDP)起動

＜関数 I/F＞

long HFA_StartCmdServerUdp(char * LocalIP)

＜引数＞

型	変 数	I/O	内 容
char *	LocalIP	IN	ローカル IP アドレス。 複数の LAN カードを実装している場合に、特定の LAN カードを指定することが可能です。 ・LAN カードを特定しない場合は、””（空文字）を指定します。この場合、OS が管理するデフォルトの LAN カードが選択されます。 ・LAN カードを特定する場合は、10 進数のゼロサプレスで”XXX.XXX.XXX.XXX”形式で指定します。

＜詳細＞

引数で指定した IP アドレスで、コマンドサーバ(UDP)を起動します。コマンドサーバ起動中に本関数を呼び出した場合に起動条件(指定するローカル IP アドレス)が異なる場合は、一旦停止して再起動します。コマンドサーバは UDP/IP および TCP/IP を同時に起動することも可能です。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常（IP アドレス不正）
-100	システムエラー

＜使用例＞

１）LAN カードを指定しないでコマンドサーバ(UDP)を起動する。

```
void SampleStartCmdServerUdp(){
    long lRet = HFA_StartCmdServerUdp("");
}
```

２） “192.168.250.25”の LAN カードを指定してコマンドサーバ(UDP)を起動する。

```
void SampleStartCmdServerUdp(){
    long lRet = HFA_StartCmdServerUdp("192.168.250.25");
}
```

＜関連事項＞

・HFA_StopCmdServerUdp 関数，HFA_GetCmdServerUdpStatus 関数，HFA_StartCmdServerTcp 関数

4.4.2. HFA_StopCmdServerUdp

コマンドサーバ(UDP)停止

＜関数 I/F＞

long HFA_StopCmdServerUdp(void)

＜引数＞

なし

＜詳細＞

コマンドサーバ(UDP)を停止します。

＜戻り値＞

値	内 容
0	正常終了
-100	システムエラー

＜使用例＞

１） コマンドサーバ(UDP)を停止する。

```
void SampleStopCmdServerUdp(){
    long lRet = HFA_StopCmdServerUdp();
}
```

＜関連事項＞

・HFA_StartCmdServerUdp 関数，HFA_GetCmdServerUdpStatus 関数

4.4.3. HFA_StartCmdServerTcp

コマンドサーバ(TCP)起動

＜関数 I/F＞

long HFA_StartCmdServerTcp(char * LocalIP)

＜引数＞

型	変 数	I/O	内 容
char *	LocalIP	IN	ローカル IP アドレス。 複数の LAN カードを実装している場合に、特定の LAN カードを指定することが可能です。 ・LAN カードを特定しない場合は、””（空文字）を指定します。この場合、OS が管理するデフォルトの LAN カードが選択されます。 ・LAN カードを特定する場合は、10 進数のゼロサプレスで”XXX.XXX.XXX.XXX”形式で指定します。

＜詳細＞

引数で指定した IP アドレスで、コマンドサーバ(TCP)を起動します。コマンドサーバ起動中に本関数を呼び出した場合に起動条件(指定するローカル IP アドレス)が異なる場合は、一旦停止して再起動します。コマンドサーバは UDP/IP および TCP/IP を同時に起動することも可能です。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常（IP アドレス不正）
-100	システムエラー

＜使用例＞

１）LAN カードを指定しないでコマンドサーバ(TCP)を起動する。

```
void SampleStartCmdServerTcp(){
    long lRet = HFA_StartCmdServerTcp("");
}
```

２） “192.168.250.25”の LAN カードを指定してコマンドサーバ(TCP)を起動する。

```
void SampleStartCmdServerTcp(){
    long lRet = HFA_StartCmdServerTcp("192.168.250.25");
}
```

＜関連事項＞

・HFA_StopCmdServerTcp 関数，HFA_GetCmdServerTcpStatus 関数，HFA_StartCmdServerUdp 関数

4.4.4. HFA_StopCmdServerTcp

コマンドサーバ(TCP)停止

＜関数 I/F＞

long HFA_StopCmdServerTcp(void)

＜引数＞

なし

＜詳細＞

コマンドサーバ(TCP)を停止します。

＜戻り値＞

値	内 容
0	正常終了
-100	システムエラー

＜使用例＞

１）コマンドサーバ(TCP)を停止する。

```
void SampleStopCmdServerTcp(){
    long lRet = HFA_StopCmdServerTcp();
}
```

＜関連事項＞

・HFA_StartCmdServerTcp 関数，HFA_GetCmdServerTcpStatus 関数

4.4.5. HFA_GetCmdServerUdpStatus

コマンドサーバ(UDP)の起動状態取得

＜関数 I/F＞

long HFA_GetCmdServerUdpStatus(long *Status, char *LocalIP)

＜引数＞

型	変 数	I/O	内 容
long *	Status	OUT	起動状態 ・0=停止中 ・1=起動中
char *	LocalIP	OUT	コマンドサーバの起動中の IP アドレス。10 進数のゼロサプレスで”XXX.XXX.XXX.XXX”形式で取得します。 コマンドサーバ停止中は不定です。 **注）呼び出し元「*APP*」では、必ず 32 バイト以上の領域を確保してください。**

＜詳細＞

コマンドサーバ(UDP)の起動状態を取得します。起動中の場合は現在起動している IP アドレスを取得します。停止中の場合は、不定になります。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）コマンドサーバ(UDP)の起動状態を取得する。

```
void SampleGetCmdServerUdpStatus(){
      long lRet, Status;
      char clp[32];
      lRet = HFA_GetCmdServerUdpStatus(&Status, clp);
}
```

＜関連事項＞

・HFA_StartCmdServerUdp 関数，HFA_StopCmdServerUdp 関数

4.4.6. HFA_GetCmdServerTcpStatus

コマンドサーバ(TCP)の起動状態取得

＜関数 I/F＞

long HFA_GetCmdServerTcpStatus(long *Status, char *LocalIP)

＜引数＞

型	変 数	I/O	内 容
long *	Status	OUT	起動状態 ・0=停止中 ・1=起動中
char *	LocalIP	OUT	コマンドサーバの起動中の IP アドレス。10 進数のゼロサプレスで”XXX.XXX.XXX.XXX”形式で取得します。 コマンドサーバ停止中は不定です。 **注）呼び出し元「*APP*」では、必ず 32 バイト以上の領域を確保してください。**

＜詳細＞

コマンドサーバ(TCP)の起動状態を取得します。起動中の場合は現在起動している IP アドレスを取得します。停止中の場合は、不定になります。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）コマンドサーバ(TCP)の起動状態を取得する。

```
void SampleGetCmdServerTcpStatus(){
      long lRet, Status;
      char cIp[32];
      lRet = HFA_GetCmdServerTcpStatus(&Status, cIp);
}
```

＜関連事項＞

・HFA_StartCmdServerTcp 関数，HFA_StopCmdServerTcp 関数

4.5. 負荷測定関数

負荷測定関数とは、トークン保持時間や汎用通信データ送信元ログを測定するための I/F 関数です。負荷測定関数の一覧を以下に示します。

No.	関数名	概要
1	HFA_StartTokenTimeMeasure	トークン保持時間測定開始
2	HFA_StopTokenTimeMeasure	トークン保持時間測定停止
3	HFA_GetTokenTimeMeasureStatus	トークン保持時間測定状態取得
4	HFA_StartDataLogMeasure	汎用通信データ送信元ログ測定開始
5	HFA_StopDataLogMeasure	汎用通信データ送信元ログ測定停止
6	HFA_GetDataLogMeasureStatus	汎用通信データ送信元ログ測定状態取得

4.5.1. HFA_StartTokenTimeMeasure

トークン保持時間測定開始

＜関数 I/F＞

long HFA_StartTokenTimeMeasure(unsigned long *ID)

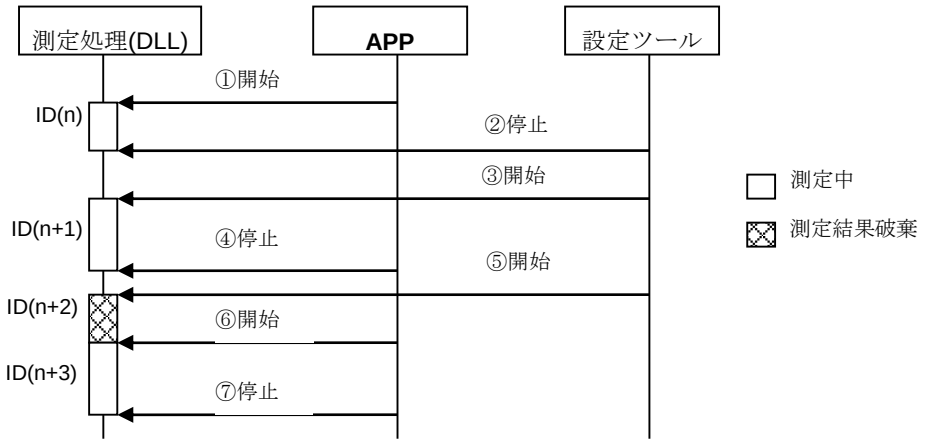
＜引数＞

型	変 数	I/O	内 容
unsigned long *	ID	OUT	測定開始時にシーケンシャルな ID を付与します。 測定開始毎に 1 つずつインクリメントされます。

＜詳細＞

トークン保持時間の測定を開始します。既に測定中の場合は、測定結果を破棄し再度測定し直します。

トークン保持時間の測定は「***APP***」以外にも、設定ツールから可能です。そのため、測定開始を指示した処理を識別するために、測定開始毎にシーケンシャルな ID を付与します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
-100	システムエラー

＜使用例＞

1） トークン保持時間の測定を開始する。

```
void SampleStartTokenTimeMeasure(){
    unsigned long uID;
    long lRet = HFA_StartTokenTimeMeasure(&uID);
}
```

＜関連事項＞

・HFA_StopTokenTimeMeasure 関数，HFA_GetTokenTimeMeasureStatus 関数

4.5.2. HFA_StopTokenTimeMeasure

トークン保持時間測定停止

＜関数 I/F＞

long HFA_StopTokenTimeMeasure(HFA_LOG_TOKENTIME *TokenSts, unsigned long *ID, long *Result)

＜引数＞

型	変 数	I/O	内 容
HFA_LOG_TOKENTIME *	TokenSts	OUT	トークン保持時間測定情報[14]
unsigned long *	ID	OUT	測定開始 ID
long *	Result	OUT	停止結果 ・0=正常停止 ・1=既に停止中

＜詳細＞

トークン保持時間の測定を停止して、引数にトークン保持時間測定情報の測定結果を格納します。既に停止中に本関数をコールした場合は、前回正常停止した時点の測定結果を格納します。

測定開始 ID は HFA_StartTokenTimeMeasure 関数の説明を参照してください。

トークン保持時間測定情報は停止結果が正常停止の場合、ログ領域に保存されます。保存された測定結果は HFA_GetMyNodeLogV3 関数でも取得可能です。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
-100	システムエラー

＜使用例＞

1）トークン保持時間の測定を停止する。

```
void SampleStopTokenTimeMeasure(){
    HFA_LOG_TOKENTIME xToken;
    long lResult;
    unsigned long uID;
    if(HFA_StopTokenTimeMeasure(&xToken, &uID, &lResult) == 0){      // トークン保持時間測定停止
        printf("トークン破棄回数=%ld¥n", xToken.Destroy);            // トークン破棄回数を参照
    }
}
```

＜関連事項＞

・HFA_StartTokenTimeMeasure 関数，HFA_GetTokenTimeMeasureStatus 関数

・HFA_GetMyNodeLogV3 関数

[14] トークン保持時間測定情報の構造体は、4.8.21 項をご参照ください。

4.5.3. HFA_GetTokenTimeMeasureStatus

トークン保持時間測定状態取得

＜関数 I/F＞

long HFA_GetTokenTimeMeasureStatus(unsigned long *ID, long *Status)

＜引数＞

型	変 数	I/O	内 容
unsigned long *	ID	OUT	測定開始 ID
long *	Status	OUT	トークン保持時間測定状態 ・0=停止中 ・1=測定中

＜詳細＞

トークン保持時間の測定状態を取得します。トークン保持時間の測定は複数の設定ツールからも開始・停止が可能です。測定開始 ID については、HFA_StartTokenTimeMeasure 関数の説明を参照してください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）トークン保持時間の測定状態を取得する。

```
void SampleGetTokenTimeMeasureStatus(){
    long lStatus;
    unsigned long uID;
    if(HFA_GetTokenTimeMeasureStatus(&uID, &lStatus) == 0){ //トークン保持時間測定状態取得
        printf("測定状態=%ld¥n", lStatus);                    //トークン保持時間測定状態参照
    }
}
```

＜関連事項＞

・HFA_StartTokenTimeMeasure 関数，HFA_StopTokenTimeMeasure 関数

4.5.4. HFA_StartDataLogMeasure

汎用通信データ送信元ログ測定開始

＜関数 I/F＞

long HFA_StartDataLogMeasure (unsigned long *ID)

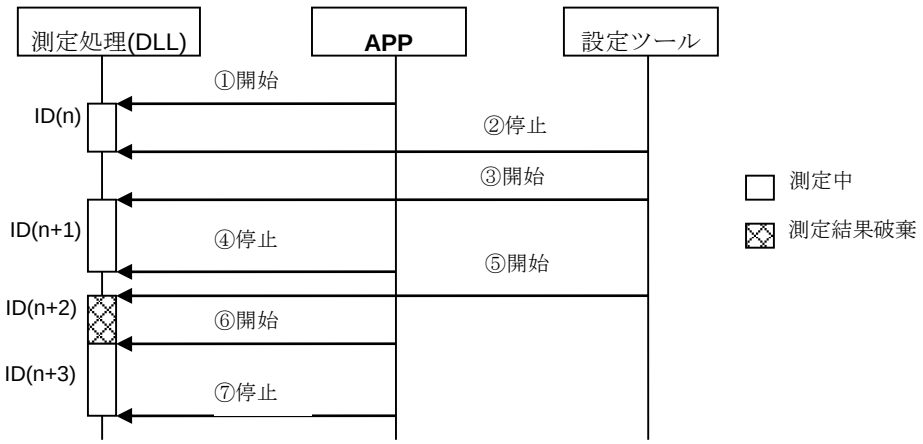
＜引数＞

型	変 数	I/O	内 容
unsigned long *	ID	OUT	測定開始時にシーケンシャルな ID を付与します。 測定開始毎に 1 つずつインクリメントされます。

＜詳細＞

汎用通信データ送信元ログの測定を開始します。既に測定開始中の場合は、測定結果を破棄して再度測定し直します。汎用通信データ送信元ログの測定は「***APP***」以外にも、設定ツールから可能です。そのため、測定開始を指示した処理を識別するために、測定開始毎にシーケンシャルな ID を付与します。

複数の NIC アダプタがある場合、自ノード参加中の場合は、参加に使用している NIC アダプタを対象とします。自ノード未参加の場合は、バインドされる優先度が一番高い NIC アダプタを対象にします。優先度の変更方法は OS のヘルプを参照してください。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
-3	NIC アダプタのオープンに失敗
-100	システムエラー(スレッドの起動に失敗)

＜使用例＞

１）汎用通信データ送信元ログの測定を開始する。

```
void SampleStartDataLogMeasure(){
      long lRet;
      unsigned long uID;
      lRet = HFA_StartDataLogMeasure(&uID);
}
```

＜関連事項＞

・HFA_StopDataLogMeasure 関数，HFA_GetDataLogMeasure 関数

4.5.5. HFA_StopDataLogMeasure

汎用通信データ送信元ログ測定停止

＜関数 I/F＞

long HFA_StopDataLogMeasure (HFA_LOG_IP_V3 *IPSts, unsigned long *ID, long *Result)

＜引数＞

型	変 数	I/O	内 容
HFA_LOG_IP_V3 *	IPSts	OUT	汎用通信データ送信元ログ情報[15]
unsigned long *	ID	OUT	測定開始 ID。
long *	Result	OUT	停止結果 ・0=正常停止 ・1=既に停止中

＜詳細＞

汎用通信データ送信元ログの測定を停止して、引数に汎用通信データ送信元ログ情報の測定結果を格納します。既に停止中に本関数をコールした場合は、前回正常停止した時点の測定結果を格納します。

測定開始 ID は HFA_StartDataLogMeasure 関数の説明を参照してください。

汎用通信データ送信元ログ情報は停止結果が正常停止の場合、ログ領域に保存されます。保存された測定結果は HFA_GetMyNodeLogV3 関数でも取得可能です。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
-100	システムエラー(スレッドの終了に失敗)

＜使用例＞

1）汎用通信データ送信元ログの測定を停止する。

```
void SampleStopDataLogMeasure(){
    HFA_LOG_IP_V3 xIP;
    long lResult;
    unsigned long uID;
    if(HFA_StopDataLogMeasure(&xIP, &uID, &lResult) == 0){    // 汎用通信データ送信元ログ測定停止
        printf("IP1 アドレス=%x¥n", xIP.Log[0].Address);          // IP1 アドレスを参照
    }
}
```

＜関連事項＞

・HFA_StartDataLogMeasure 関数，HFA_GetDataLogMeasureStatus 関数

・HFA_GetMyNodeLogV3 関数

[15]汎用通信データ送信元ログ情報の構造体は、4.8.30 項をご参照ください。

4.5.6. HFA_GetDataLogMeasureStatus

汎用通信データ送信元ログ測定状態取得

＜関数 I/F＞

long HFA_GetDataLogMeasureStatus(unsigned long *ID, long *Status)

＜引数＞

型	変 数	I/O	内 容
unsigned long *	ID	OUT	測定開始 ID。
long *	Status	OUT	汎用通信データ送信元ログ測定状態 ・0=停止中 ・1=測定中

＜詳細＞

汎用通信データ送信元ログの測定状態を取得します。汎用通信データ送信元ログの測定は複数の設定ツールからも開始・停止が可能です。測定開始 ID については、HFA_StartDataLogMeasure 関数の説明を参照してください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常

＜使用例＞

１）汎用通信データ送信元ログの測定状態を取得する。

```
void SampleGetDataLogMeasureStatus(){
    long lStatus;
    unsigned long uID;
    if(HFA_GetDataLogMeasureStatus(&uID, &lStatus) == 0){ //汎用通信データ送信元ログ測定状態取得
        printf("測定状態=%ld¥n", lStatus);                    //汎用通信データ送信元ログ測定状態参照
    }
}
```

＜関連事項＞

・HFA_StartDataLogMeasure 関数，HFA_StopDataLogMeasure 関数

4.6. デバイスレベルネットワーク関数

デバイスレベルネットワーク関数とは、任意マスタ機能を実行するための I/F 関数です。デバイスレベルネットワーク関数の一覧を以下に示します。

No.	関数名	概要
1	HFA_GetSlaveNodeLinkStatus	スレーブノード参加状態取得
2	HFA_SetIO	IO 割付情報の設定
3	HFA_GetIO	IO 割付情報の取得
4	HFA_SetOutputStatus	出力ステータス設定
5	HFA_GetOutputStatus	出力ステータス取得
6	HFA_GetInputStatus	入力ステータス取得
7	HFA_ReadInputData	入力データの全読出し
8	HFA_ReadInputBitData	入力データの連続ビット読出し
9	HFA_ReadInputWordData	入力データの連続ワード読出し
10	HFA_ReadInputRandomBitData	入力データのランダムビット読出し
11	HFA_ReadInputRandomWordData	入力データのランダムワード読出し
12	HFA_ReadOutputData	出力データの全読出し
13	HFA_ReadOutputBitData	出力データの連続ビット読出し
14	HFA_ReadOutputWordData	出力データの連続ワード読出し
15	HFA_ReadOutputRandomBitData	出力データのランダムビット読出し
16	HFA_ReadOutputRandomWordData	出力データのランダムワード読出し
17	HFA_WriteOutputBitData	出力データの連続ビット書込み
18	HFA_WriteOutputWordData	出力データの連続ワード書込み
19	HFA_WriteOutputRandomBitData	出力データのランダムビット書込み
20	HFA_WriteOutputRandomWordData	出力データのランダムワード書込み

4.6.1. HFA_GetSlaveNodeLinkStatus

スレーブノード参加状態読出し

＜関数 I/F＞

long HFA_GetSlaveNodeLinkStatus(unsigned char *Node)

＜引数＞

型	変 数	I/O	内 容
unsigned char *	Node	OUT	ノード参加状態。スレーブノードの参加状態を、ノード番号の昇順（1～249）で 1 バイト毎に格納します。unsigned char 型配列の添え字がノード番号と対応し、参加状態の値は以下となります。 ・0=未参加 ・1=参加中 ・2=スレーブノード以外 **注）呼び出し元「*APP*」では、必ず 250 バイト以上の領域を確保してください。**

＜詳細＞

自ノードが管理しているスレーブノード参加状態を読出します。FL-net ネットワーク未参加の状態で本関数をコールした場合、戻り値=自ノード未参加となり、ステータスの読出しを行いません。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加（モニタモードを含む）

＜使用例＞

1）スレーブノード参加状態を読出す。

```
void SampleGetSlaveNodeLinkStatus(){
    unsigned char ucNode[250];
    if(HFA_GetSlaveNodeLinkStatus(ucNode) == 0){          // スレーブノード参加状態読出し
        printf("ノード 100 の参加状態=%d¥n",ucNode[100]);   // ノード 100 の参加状態を参照
    }
}
```

＜関連事項＞

・ HFA_GetNetworkStatus 関数

4.6.2. HFA_SetIO

IO 割付情報の設定

＜関数 I/F＞

long HFA_SetIO (long SlaveNum, HFA_SLAVE_INFO *IOInfo, unsigned char *Error)

＜引数＞

型	変 数	I/O	内 容
long	SlaveNum	IN	制御スレーブ個数（0 の場合は任意マスタ機能無し）
HFA_SLAVE_INFO *	IOInfo	IN	IO 割付情報[16]の配列の先頭ポインタ。
unsigned char *	Error	OUT	割付エラー情報を IO 割付情報の配列と同じ並びで 1 バイト毎に格納します。 ・0=正常 ・1=ノード番号が範囲外 ・2=ノード番号が自ノード番号と同じ ・3=ノード番号が重複 ・4=領域が範囲外(1,2 以外) ・5=アドレスまたはサイズが範囲外 ・6=出力データと出力ステータスが重複 ・7=スレーブ出力エリアが重複 ・8=スレーブ出力エリアが自ノード送信領域範囲外 ・9=入力データと入力ステータスが重複 ・10=スレーブ入力エリアが重複 ・11=スレーブ入力エリアが自ノード送信領域範囲内 ・12=入出力点数が範囲外 **注）呼び出し元「*APP*」では、必ず制御スレーブ個数バイト以上の領域を確保してください。**

＜引数範囲＞

項目(式)	値正常範囲	備 考
SlaveNum	0≦(値)≦248	

＜詳細＞

IO 割付情報を設定します。本関数はリンク未参加時に設定可能です。IO 割付を設定する前に、あらかじめ HFA_SetCommon 関数をコールして、自ノードのコモンメモリ送信領域を設定しておく必要があります。本関数のコールが正常終了すると、前回の設定はクリアされて新しい設定で上書きします。従って IO 割付情報を追加する目的で使用出来ません。

[16] IO 割付情報の構造体は、4.8.16 項をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
4	自ノード参加中（モニタモード含む）
7	既に参加待機中
10	設定ツールから設定中

＜使用例＞

１）IO 割付情報の設定をする。

```
#define SLAVE_NUM 2       //スレーブ個数
void SampleSetIO(){
      HFA_SLAVE_INFO xIO[SLAVE_NUM];                  // IO 割付情報
      unsigned char ucErr[SLAVE_NUM];                 // エラー情報を格納
      int i;
      // IO 割付情報の設定
      //ノード番号 1 の設定
      xIO[0].NodeNo = 1;                              // ノード番号
      xIO[0].InDataAddress.Area = 1;                  // IO 入力データアドレス領域=領域 1
      xIO[0].InDataAddress.Address = 0x0010;          // IO 入力データアドレス
      xIO[0].InDataSize = 4;                          // IO 入力データサイズ(4 ワード)
      xIO[0].InputPoint = 32;                         // IO 入力点数(32 点)
      xIO[0].OutDataAddress.Area = 1;                 // IO 出力データアドレス領域=領域 1
      xIO[0].OutDataAddress.Address = 0x0020;         // IO 出力データアドレス
      xIO[0].OutDataSize = 4;                         // IO 出力データサイズ(4 ワード)
      xIO[0].OutputPoint = 32;                        // IO 出力点数(32 点)
      xIO[0].InStatusAddress.Area = 2;                // IO 入力ステータスアドレス領域=領域 2
      xIO[0].InStatusAddress.Address = 0x0100;        // IO 入力ステータスアドレス
      xIO[0].OutStatusAddress.Area = 2;               // IO 出力ステータスアドレス領域=領域 2
      xIO[0].OutStatusAddress.Address = 0x0110;       // IO 出力ステータスアドレス
      //ノード番号 3 の設定
      xIO[1].NodeNo = 3;                              // ノード番号
      xIO[1].InDataAddress.Area = 1;                  // IO 入力データアドレス領域=領域 1
      xIO[1].InDataAddress.Address = 0x0030;          // IO 入力データアドレス
      xIO[1].InDataSize = 4;                          // IO 入力データサイズ(4 ワード)
      xIO[1].InputPoint = 32;                         // IO 入力点数(32 点)
      xIO[1].OutDataAddress.Area = 1;                 // IO 出力データアドレス領域=領域 1
      xIO[1].OutDataAddress.Address = 0x0040;         // IO 出力データアドレス
      xIO[1].OutDataSize = 4;                         // IO 出力データサイズ(4 ワード)
      xIO[1].OutputPoint = 32;                        // IO 出力点数(32 点)
      xIO[1].InStatusAddress.Area = 2;                // IO 入力ステータスアドレス領域=領域 2
      xIO[1].InStatusAddress.Address = 0x0120;        // IO 入力ステータスアドレス
      xIO[1].OutStatusAddress.Area = 2;               // IO 出力ステータスアドレス領域=領域 2
      xIO[1].OutStatusAddress.Address = 0x0130;       // IO 出力ステータスアドレス

      // IO 割付情報の登録
      if(HFA_SetIO(SLAVE_NUM, xIO, ucErr) == -1){     // 引数異常の場合エラーを表示する
            for(i=0;i< SLAVE_NUM;i++){
                  printf("エラー要因[%d] = %d¥n", i, ucErr[i]);
            }
      }
}
```

＜関連事項＞

・HFA_SetCommon 関数，HFA_GetIO 関数

4.6.3. HFA_GetIO

IO 割付情報の取得

＜関数 I/F＞

long HFA_GetIO (HFA_SLAVES_INFO *IOInfo)

＜引数＞

型	変 数	I/O	内 容
HFA_SLAVES_INFO *	IOInfo	OUT	IO 割付情報[17]のポインタ

＜詳細＞

IO 割付情報を取得します。取得される点数情報は HFA_GetOutputStatus 関数で取得されるものと同じです。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
10	設定ツールから設定中（※取得した I/O 割付情報は有効です）

＜使用例＞

１）IO 割付情報を取得する。

```
void SampleGetIO(){
    HFA_SLAVES_INFO xIO;
    int i;
    if(HFA_GetIO(&xIO) == 0){      // IO 割付情報の取得
        printf("スレーブ個数=%d¥n",xIO.Number);
        for(i=0;i<xIO.Number;i++){
            printf("スレーブ番号=%d¥n", xIO.Infos[i].NodeNo);
            printf("IO 入力データアドレス=%d¥n", xIO.Infos[i].InDataAdress.Address);
        }
    }
}
```

＜関連事項＞

・HFA_SetIO 関数

[17] IO 割付情報の構造体は、4.8.17 項をご参照ください。

4.6.4. HFA_SetOutputStatus

出力ステータス設定

＜関数 I/F＞

long HFA_SetOutputStatus(long NodeNo, HFA_SLAVE_OUTPUT_STATUS *Status, long SetMask)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	スレーブノード番号
HFA_SLAVE_OUTPUT_STATUS *	Status	IN	出力ステータス[18]のポインタ
long	SetMask	IN	入出力点数を更新するかどうかを指定します。 ・0x1･･･入出力点数を更新する

＜引数範囲＞

項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力ステータスを設定します。設定項目を以下に示します。自ノードおよび対象スレーブノードが未参加でも設定可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

設定項目	設定値
リモート制御エリア	Status.RemoteControl の値を設定します。
指定スレーブ種別	SetMask=1 の条件で、Status.OutputStatus.InputPoint メンバの値および Status.OutputStatus.OutputPoint の値に基づき設定します。
簡易設定エリア マスタノード番号指示	「***DLL***」が自ノード番号を設定します。 Status.OutputStatus.MasterNodeNo の値は参照しません。
簡易設定エリア マスタ離脱時の IO 出力	Status.OutputStatus.MasterOffOutput の値を設定します。
簡易設定エリア リモート制御フラグ OFF 時の IO 出力	Status.OutputStatus.RemoteOffOutput の値を設定します。
簡易設定エリア リモート制御フラグ OFF 時の IO 入力	Status.OutputStatus.RemoteOffInput の値を設定します。
簡易設定エリア マスタからスレーブ毎に指定する領域	Status.OutputStatus.FreeData の値を設定します。
予約エリア	Status.OutputStatus.GeneralPurpose の値を設定します。

[18] 出力ステータス情報の構造体は、4.8.20 項をご参照ください。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※設定は成功しています）
5	対象ノード未参加（※設定は成功しています）

＜使用例＞

１）対象スレーブノードの出力ステータスを設定する。

```
void SampleSetOutputStatus(){
    HFA_SLAVE_OUTPUT_STATUS xStatus;
    memset(xStatus,0, sizeof(xStatus));
    xStatus.RemoteControl = 1;                  // リモート動作
    xStatus.OutputStatus.InputPoint = 16;       // 入力点数
    xStatus.OutputStatus.OutputPoint = 16;      // 出力点数
    xStatus.OutputStatus.MasterOffOutput = 1; // マスタ離脱時の IO 出力=ホールド
    xStatus.OutputStatus.RemoteOffInput = 1;  // リモート制御フラグ OFF 時の IO 入力=ホールド
    long lRet = HFA_SetOutputStatus(1, &xStatus,1);
}
```

＜関連事項＞

・HFA_GetOutputStatus 関数

4.6.5. HFA_GetOutputStatus

出力ステータス取得

＜関数 I/F＞

long HFA_GetOutputStatus(long NodeNo, HFA_SLAVE_OUTPUT_STATUS *Status)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	スレーブノード番号
HFA_SLAVE_OUTPUT_STATUS *	Status	OUT	出力ステータス[19]のポインタ

＜引数範囲＞

項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力ステータスを取得します。取得可能な項目を以下に示します。自ノードおよび対象スレーブノードが未参加でも取得可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

取得可能項目	変数名
リモート制御エリア	Status.RemoteControl
指定スレーブ種別	入力点数：Status.OutputStatus.InputPoint 出力点数：Status.OutputStatus.OutputPoint
マスタノード番号指示	Status.OutputStatus.MasterNodeNo
マスタ離脱時の IO 出力	Status.OutputStatus.MasterOffOutput
リモート制御フラグ OFF 時の IO 出力	Status.OutputStatus.RemoteOffOutput
リモート制御フラグ OFF 時の IO 入力	Status.OutputStatus.RemoteOffInput
マスタからスレーブ毎に指定する領域	Status.OutputStatus.FreeData
予約エリア	Status.OutputStatus.GeneralPurpose

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※取得した出力ステータスは有効です）
5	対象ノード未参加（※取得した出力ステータスは有効です）

[19] 出力ステータス情報の構造体は、4.8.20 項をご参照ください。

＜使用例＞

１）対象スレーブノードの出力ステータスを取得する。

```
void SampleGetOutputStatus(){
    HFA_SLAVE_OUTPUT_STATUS xStatus;
    if(HFA_GetOutputStatus(1, &xStatus) != -1){    // 出力ステータスを読出す
        printf("リモート制御=%d¥n",xStatus.RemoteControl);
        printf("入力点数=%d, 出力点数=%d",
            xStatus.OutputStatus.InputPoint, xStatus.OutputStatus.OutputPoint);
    }
}
```

＜関連事項＞

・HFA_SetOutputStatus 関数，HFA_GetInputStatus 関数

4.6.6. HFA_GetInputStatus

入力ステータス取得

＜関数 I/F＞

long HFA_GetInputStatus(long NodeNo, HFA_SLAVE_INPUT_STATUS *Status)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	スレーブノード番号
HFA_SLAVE_INPUT_STATUS *	Status	OUT	入力ステータス[20]のポインタ

＜引数範囲＞

項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの入力ステータスを取得します。取得可能な項目を以下に示します。自ノードおよび対象スレーブノードが未参加でも取得可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

取得可能項目	変数名
スレーブ状態ステータス	Status.StatusNo
実構成スレーブ種別	入力点数：Status.InputStatus.InputPoint 出力点数：Status.InputStatus.OutputPoint
マスタノード番号	Status.InputStatus.MasterNodeNo
マスタ離脱時の IO 出力	Status.InputStatus.MasterOffOutput
リモート制御フラグ OFF 時の IO 出力	Status.InputStatus.RemoteOffOutput
リモート制御フラグ OFF 時の IO 入力	Status.InputStatus.RemoteOffInput
マスタから指定された値	Status.InputStatus.FreeData
汎用ステータスエリア	Status.InputStatus.GeneralPurpose

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※取得した入力ステータスは有効です）
5	対象ノード未参加（※取得した入力ステータスは有効です）
8	コモンメモリ無効（対象ノードの FA リンク状態がコモンメモリ無効の場合）

[20] 入力ステータス情報の構造体は、4.8.19 項をご参照ください。

＜使用例＞

１）対象スレーブノードの入力ステータスを取得する。

```
void SampleGetInputStatus(){
      HFA_SLAVE_INPUT_STATUS xStatus;
      if(HFA_GetInputStatus(1, &xStatus) == 0){     // 入力ステータスを読出す
            printf(“入力スレーブ点数=%d¥n”,xStatus.InputStatus.InputPoint);
      }
}
```

＜関連事項＞

・HFA_GetOutputStatus 関数

・*InputStatusRefresh* イベント

4.6.7. HFA_ReadInputData

入力データの全読出し

＜関数 I/F＞

long HFA_ReadInputData(long NodeNo, HFA_ADDRESS *Addr,
unsigned short *Size, unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
HFA_ADDRESS *	Addr	OUT	読出し先アドレス情報[21]のポインタ
unsigned short *	Size	IN/OUT	読出し先サイズ（ワード単位）のポインタ
unsigned short *	Data	OUT	読出し先ワードデータのポインタ **注）呼び出し元「*APP*」では、必ず読出しサイズ以上の領域を確保してください。**

＜引数範囲＞

項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの入力データを Size 分読出します。Size には実際に読み出したサイズが格納されます。自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）
8	コモンメモリ無効（対象ノードの FA リンク状態がコモンメモリ無効の場合）

[21] アドレス情報の構造体は、4.8.15 項をご参照ください。

＜使用例＞

１）対象スレーブノードの入力データを読出す。

```
void SampleReadInputData(){
   HFA_ADDRESS xAddr;
   unsigned short usSize = 100;
   unsigned short usData[100];
   if(HFA_ReadInputData(1, &xAddr, &usSize, usData) == 0){      // 入力データを読出す
      printf(“入力データのサイズ=%d¥n”,usSize);                  // 入力データのサイズを参照
   }
}
```

＜関連事項＞

・HFA_ReadInputBitData 関数

・HFA_ReadInputWordData 関数

・HFA_ReadInputRandomBitData 関数

・HFA_ReadInputRandomWordData 関数

・*InputDataRefresh* イベント

4.6.8. HFA_ReadInputBitData

入力データの連続ビット読出し

＜関数 I/F＞

long HFA_ReadInputBitData(long NodeNo, unsigned long StartAddr, unsigned long Size, char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long	StartAddr	IN	読出し開始アドレス（ビット単位） ※IO 入力データエリア内の相対アドレス
unsigned long	Size	IN	読出し点数（ビット単位）
char *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ビットに対応します。配列要素の値は以下の通りとなります。 ・0=ビット値 0 ・1=ビット値 1 **注)　呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

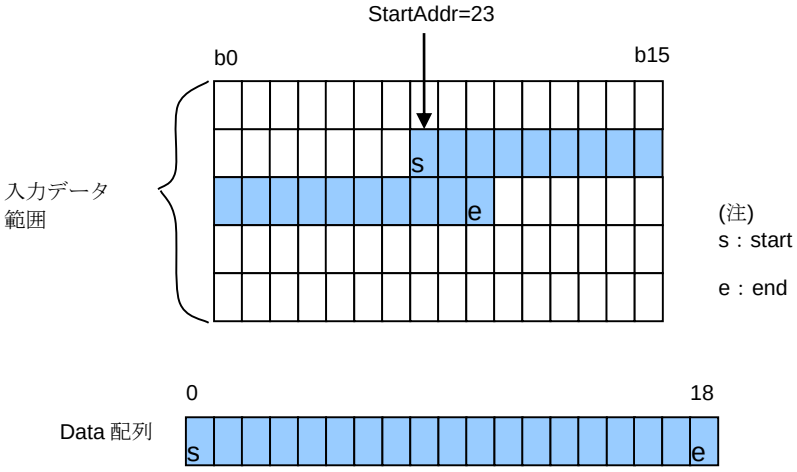
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
StartAddr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの入力データを連続ビット読出します。読出し開始アドレスに、I/O 入力データエリア内の相対アドレスを指定して、指定した読出し点数のビットデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

入力データアドレスの 23 ビット目から 19 点ビットデータを読出す例を以下に示します。



配列サイズ 19 の Data に結果(0 or 1)が格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています)
8	コモンメモリ無効（対象ノードの FA リンク状態＝コモンメモリ無効の場合）

＜使用例＞

１）対象スレーブノードの入力データを読出す。

```
void SampleReadInputBitData(){
      char Data[19];
      int i;
      // ノード番号１、23 ビット目から 19 点ビットデータを読出す
      if(HFA_ReadInputBitData(1, 23, 19, Data) == 0){
            for(i=0;i<19;i++){       //ビット値を確認する
               printf(“%d[bit]=%d¥n”,i+23, Data[i]);
            }
      }
}
```

＜関連事項＞

・HFA_ReadInputData 関数

・HFA_ReadInputWordData 関数

・HFA_ReadInputRandomBitData 関数

・HFA_ReadInputRandomWordData 関数

・*InputDataRefresh* イベント

4.6.9. HFA_ReadInputWordData

入力データの連続ワード読出し

＜関数 I/F＞

long HFA_ReadInputWordData(long NodeNo, unsigned long StartAddr, unsigned long Size,
unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long	StartAddr	IN	読出し開始アドレス（ワード単位） ※IO 入力データエリア内の相対アドレス
unsigned long	Size	IN	読出し点数（ワード単位）
unsigned short *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ワードに対応します。 **注) 呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

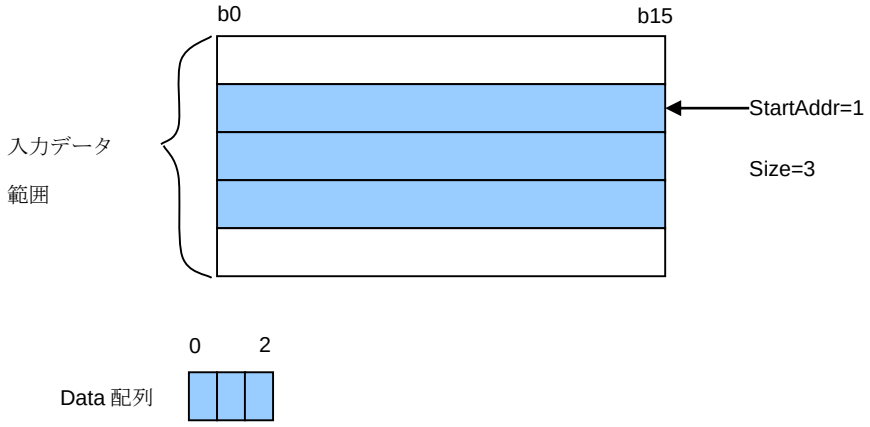
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
StartAddr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの入力データを連続ワード読出します。読出し開始アドレスに、I/O 入力データエリア内の相対アドレスを指定して、指定した読出し点数のワードデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

入力データアドレスの 1 ワード目から 3 ワードデータを読出す例を以下に示します。



配列サイズ 3 の Data に結果が格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）
8	コモンメモリ無効（対象ノードの FA リンク状態＝コモンメモリ無効の場合）

＜使用例＞

１）対象スレーブノードの入力データを読出す。

```
void SampleReadInputWordData(){
    unsigned short Data[3];
    int i;

    // ノード番号１、1 ワード目から 3 ワードデータを読出す
    if(HFA_ReadInputWordData(1, 1, 3, Data) == 0){
        for(i=0;i<3;i++){      // ワード値を確認する
            printf("%d[word]=%d¥n",i+1, Data[i]);
        }
    }
}
```

＜関連事項＞

- HFA_ReadInputData 関数
- HFA_ReadInputBitData 関数
- HFA_ReadInputRandomBitData 関数
- HFA_ReadInputRandomWordData 関数
- *InputDataRefresh* イベント

4.6.10. HFA_ReadInputRandomBitData

入力データのランダムビット読出し

＜関数 I/F＞

long HFA_ReadInputRandomBitData(long NodeNo, unsigned long *Addr, unsigned long Size,
char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long *	Addr	IN	読出し先アドレス（ビット単位）の配列 ※I/O 入力データエリア内の相対アドレス **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**
unsigned long	Size	IN	読出し点数（ビット単位）
char *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ビットに対応します。配列要素の値は以下の通りとなります。 ・0=ビット値 0 ・1=ビット値 1 **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

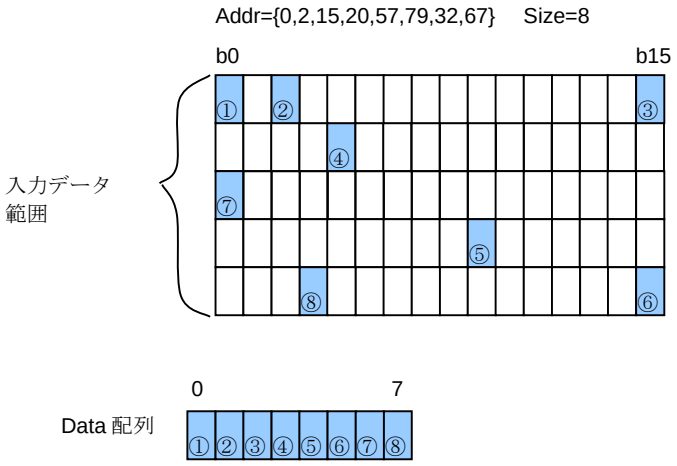
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
Addr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの入力データをランダムビット読出します。読出しアドレスに、I/O 入力データエリア内の相対アドレスを複数指定して、指定した読出し点数のビットデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

入力データアドレスからランダムに 8 点ビットデータを読出す例を以下に示します。



配列サイズ 8 の Data に結果(0 or 1)が Addr の並び順に格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）
8	コモンメモリ無効（対象ノードの FA リンク状態＝コモンメモリ無効の場合）

＜使用例＞

１）対象スレーブノードの入力データを読出す。

```
void SampleReadInputRandomBitData(){
    unsigned long Addr[8] = {0,2,15,20,57,79,32,67};
    char Data[8];
    int i;

    // ノード番号１、ランダムに 8 点ビットデータを読出す
    if(HFA_ReadInputRandomBitData(1, Addr, 8, Data) == 0){
        for(i=0;i<8;i++){      // ビット値を確認する
            printf("%d[bit]=%d¥n",Addr[i], Data[i]);
        }
    }
}
```

＜関連事項＞

・HFA_ReadInputData 関数

・HFA_ReadInputBitData 関数

・HFA_ReadInputWordData 関数

・HFA_ReadInputRandomWordData 関数

・*InputDataRefresh* イベント

4.6.11. HFA_ReadInputRandomWordData

入力データのランダムワード読出し

＜関数 I/F＞

long HFA_ReadInputRandomWordData(long NodeNo, unsigned long *Addr, unsigned long Size, unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long *	Addr	IN	読出し先アドレス（ワード単位）の配列 ※I/O 入力データエリア内の相対アドレス **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**
unsigned long	Size	IN	読出し点数（ワード単位）
unsigned short *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ワードに対応します。 **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

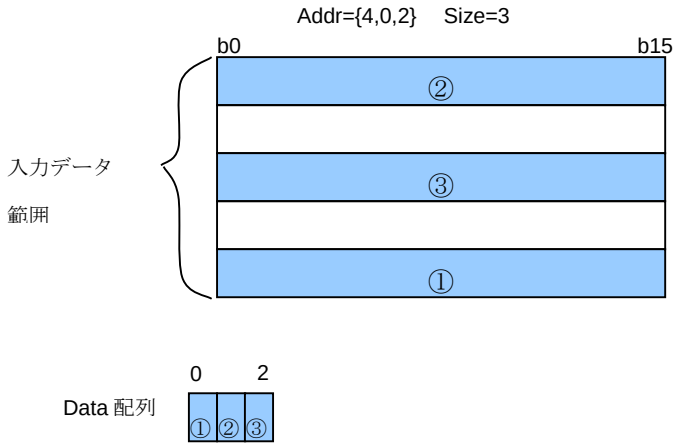
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
Addr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの入力データをランダムワード読出します。読出しアドレスに、I/O 入力データエリア内の相対アドレスを複数指定して、指定した読出し点数のワードデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

入力データアドレスからランダムに 3 ワードデータを読出す例を以下に示します。



配列サイズ 3 の Data に結果が Addr の並び順に格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）
8	コモンメモリ無効（対象ノードの FA リンク状態＝コモンメモリ無効の場合）

＜使用例＞

１）対象スレーブノードの入力データをランダムに読出す。

```
void SampleReadInputRandomWordData(){
    unsigned long Addr[3] = {4,0,2};
    unsigned short Data[3];
    int i;

    // ノード番号１、ランダムに 3 ワードデータを読出す
    if(HFA_ReadInputRandomWordData(1, Addr, 3, Data) == 0){
        for(i=0;i<3;i++){      //ワード値を確認する
            printf(“%d[word]=%d¥n”,Addr[i], Data[i]);
        }
    }
}
```

＜関連事項＞

・HFA_ReadInputData 関数

・HFA_ReadInputBitData 関数

・HFA_ReadInputWordData 関数

・HFA_ReadInputRandomBitData 関数

・InputDataRefresh イベント

4.6.12. HFA_ReadOutputData

出力データの全読出し

＜関数 I/F＞

long HFA_ReadOutputData(long NodeNo, HFA_ADDRESS *Addr,
unsigned short *Size, unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
HFA_ADDRESS *	Addr	OUT	読出し先アドレス情報[22]のポインタ
unsigned short *	Size	IN/OUT	読出し先サイズ（ワード単位）のポインタ
unsigned short *	Data	OUT	読出し先ワードデータのポインタ **注）呼び出し元「*APP*」では、必ず読出しサイズ以上の領域を確保してください。**

＜引数範囲＞

項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力データを Size 分読出します。Size には実際に読み出したサイズが格納されます。自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）

[22] アドレス情報の構造体は、4.8.15 項をご参照ください。

＜使用例＞

１）対象スレーブノードの出力データを読出す。

```
void SampleReadOutputData(){
      HFA_ADDRESS xAddr;
      unsigned short usSize = 100;
      unsigned short usData[100];

      if(HFA_ReadOutputData(1, &xAddr, &usSize, usData) == 0){  // 出力データを読出す
            printf(“出力データのサイズ=%d¥n”,usSize);                    // 出力データのサイズを参照
    }
}
```

＜関連事項＞

・HFA_ReadOutputBitData 関数

・HFA_ReadOutputWordData 関数

・HFA_ReadOutputRandomBitData 関数

・HFA_ReadOutputRandomWordData 関数

4.6.13. HFA_ReadOutputBitData

出力データの連続ビット読出し

＜関数 I/F＞

long HFA_ReadOutputBitData(long NodeNo, unsigned long StartAddr, unsigned long Size, char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long	StartAddr	IN	読出し開始アドレス（ビット単位） ※IO 出力データエリア内の相対アドレス
unsigned long	Size	IN	読出し点数（ビット単位）
char *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ビットに対応します。配列要素の値は以下の通りとなります。 ・0=ビット値 0 ・1=ビット値 1 **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

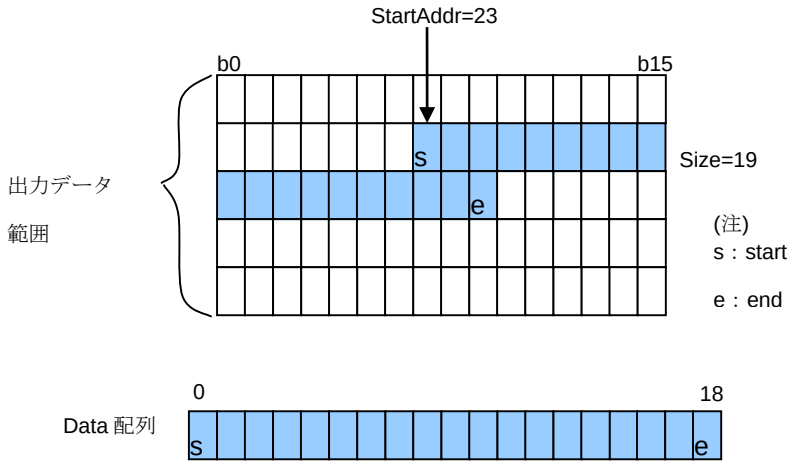
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
StartAddr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力データを連続ビット読出します。読出し開始アドレスに、I/O 出力データエリア内の相対アドレスを指定して、指定した読出し点数のビットデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

出力データアドレスの 23 ビット目から 19 点ビットデータを読出す例を以下に示します。



配列サイズ 19 の Data に結果(0 or 1)が格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）

＜使用例＞

１）対象スレーブノードの出力データを読出す。

```
void SampleReadOutputBitData(){
    char Data[19];
    int i;

    // ノード番号１、23 ビット目から 19 点ビットデータを読出す
    if(HFA_ReadOutputBitData(1, 23, 19, Data) == 0){
        for(i=0;i<19;i++){    // ビット値を確認する
            printf("%d[bit]=%d¥n",i+23, Data[i]);
        }
    }
}
```

＜関連事項＞

・HFA_ReadOutputData 関数

・HFA_ReadOutputWordData 関数

・HFA_ReadOutputRandomBitData 関数

・HFA_ReadOutputRandomWordData 関数

4.6.14. HFA_ReadOutputWordData

出力データの連続ワード読出し

＜関数 I/F＞

long HFA_ReadOutputWordData(long NodeNo, unsigned long StartAddr, unsigned long Size,
unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long	StartAddr	IN	読出し開始アドレス（ワード単位） ※IO 出力データエリア内の相対アドレス
unsigned long	Size	IN	読出し点数（ワード単位）
unsigned short *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ワードに対応します。 **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

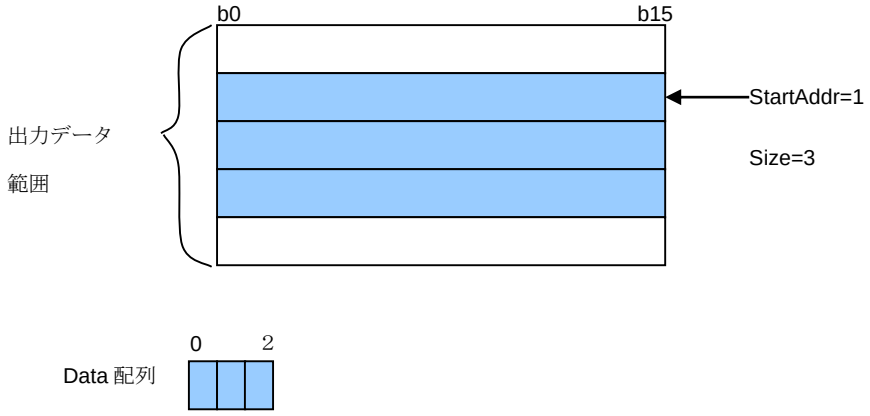
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
StartAddr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力データを連続ワード読出します。読出し開始アドレスに、I/O 出力データエリア内の相対アドレスを指定して、指定した読出し点数のワードデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

出力データアドレスの 1 ワード目から 3 ワードデータを読出す例を以下に示します。



配列サイズ 3 の Data に結果が格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）

＜使用例＞

１）対象スレーブノードの出力データを読出す。

```
void SampleReadOutputWordData(){
    unsigned short Data[3];
    int i;

    // ノード番号１、1 ワード目から 3 ワードデータを読出す
    if(HFA_ReadOutputWordData(1, 1, 3, Data) == 0){
        for(i=0;i<3;i++){      // ワード値を確認する
            printf("%d[word]=%d¥n",i+1, Data[i]);
        }
    }
}
```

＜関連事項＞

・HFA_ReadOutputData 関数

・HFA_ReadOutputBitData 関数

・HFA_ReadOutputRandomBitData 関数

・HFA_ReadOutputRandomWordData 関数

4.6.15. HFA_ReadOutputRandomBitData

出力データのランダムビット読出し

＜関数 I/F＞

long HFA_ReadOutputRandomBitData(long NodeNo, unsigned long *Addr, unsigned long Size,
char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long *	Addr	IN	読出し先アドレス（ビット単位）の配列 ※I/O 出力データエリア内の相対アドレス **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**
unsigned long	Size	IN	読出し点数（ビット単位）
char *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ビットに対応します。配列要素の値は以下の通りとなります。 ・0=ビット値 0 ・1=ビット値 1 **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

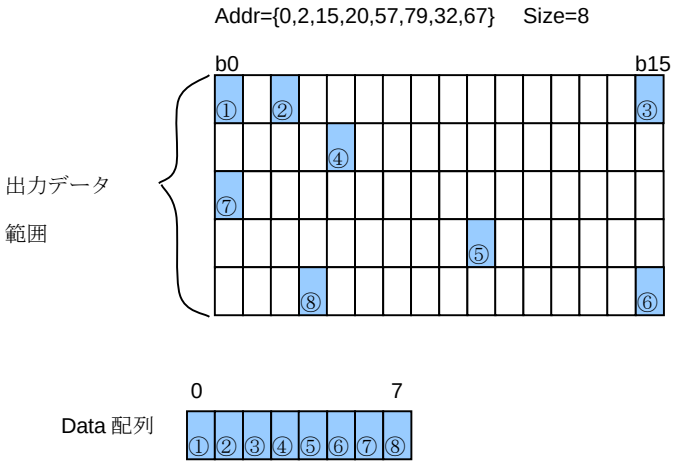
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
StartAddr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力データをランダムビット読出します。読出しアドレスに、I/O 出力データエリア内の相対アドレスを複数指定して、指定した読出し点数のビットデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

出力データアドレスからランダムに 8 点ビットデータを読出す例を以下に示します。



配列サイズ 8 の Data に結果(0 or 1)が Addr の並び順に格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）

＜使用例＞

１）対象スレーブノードの出力データを読出す。

```
void SampleReadOutputRandomBitData(){
    unsigned long Addr[8] = {0,2,15,20,57,79,32,67};
    char Data[8];
    int i;

    // ノード番号１、ランダムに 8 点ビットデータを読出す
    if(HFA_ReadOutputRandomBitData(1, Addr, 8, Data) == 0){
        for(i=0;i<8;i++){      //ビット値を確認する
            printf(“%d[bit]=%d¥n”,Addr[i], Data[i]);
        }
    }
}
```

＜関連事項＞

・HFA_ReadOutputData 関数

・HFA_ReadOutputBitData 関数

・HFA_ReadOutputWordData 関数

・HFA_ReadOutputRandomWordData 関数

4.6.16. HFA_ReadOutputRandomWordData

出力データのランダムワード読出し

＜関数 I/F＞

long HFA_ReadOutputRandomWordData(long NodeNo, unsigned long *Addr, unsigned long Size, unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	読出し先スレーブノード番号
unsigned long *	Addr	IN	読出し先アドレス（ワード単位）の配列 ※I/O 出力データエリア内の相対アドレス **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**
unsigned long	Size	IN	読出し点数（ワード単位）
unsigned short *	Data	OUT	読出し先データのポインタ 配列の 1 要素が 1 ワードに対応します。 **注）呼び出し元「*APP*」では、必ず読出し点数以上の領域を確保してください。**

＜引数範囲＞

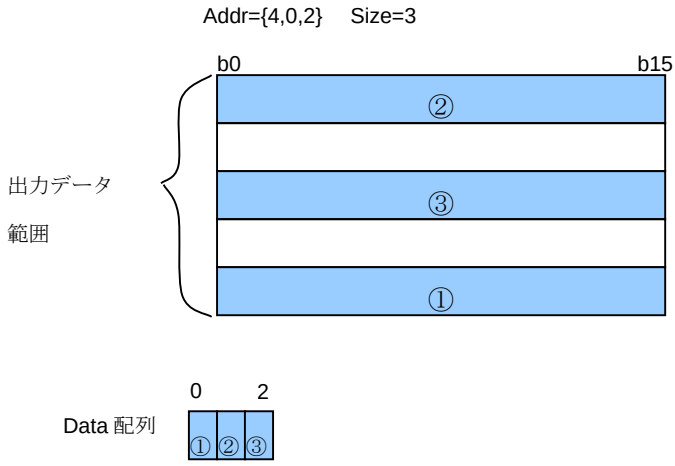
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
Addr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力データをランダムワード読出します。読出しアドレスに、I/O 入力データエリア内の相対アドレスを複数指定して、指定した読出し点数のワードデータを読出します。

自ノードおよび対象スレーブノードが未参加でも読出し可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。

出力データアドレスからランダムに 3 ワードデータを読出す例を以下に示します。



配列サイズ 3 の Data に結果が Addr の並び順に格納されます。

＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※読出しは成功しています）
5	対象ノード未参加（※読出しは成功しています）

＜使用例＞

１）対象スレーブノードの出力データをランダムに読出す。

```
void SampleReadOutputRandomWordData(){
    unsigned long Addr[3] = {4,0,2};
    unsigned short Data[3];
    int i;

    // ノード番号１、ランダムに 3 ワードデータを読出す
    if(HFA_ReadOutputRandomWordData(1, Addr, 3, Data) == 0){
        for(i=0;i<3;i++){      // ワード値を確認する
            printf("%d[word]=%d¥n",Addr[i], Data[i]);
        }
    }
}
```

＜関連事項＞

・HFA_ReadOutputData 関数

・HFA_ReadOutputBitData 関数

・HFA_ReadOutputWordData 関数

・HFA_ReadOutputRandomBitData 関数

4.6.17. HFA_WriteOutputBitData

出力データの連続ビット書込み

＜関数 I/F＞

long HFA_WriteOutputBitData(long NodeNo, unsigned long StartAddr, unsigned long Size, char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	書込み先スレーブノード番号
unsigned long	StartAddr	IN	書込み開始アドレス（ビット単位） ※IO 出力データエリア内の相対アドレス
unsigned long	Size	IN	書込み点数（ビット単位）
char *	Data	IN	書込み先データのポインタ 配列の 1 要素が 1 ビットに対応します。配列要素に以下の値を指定します。 ・0=ビット値 0 ・1=ビット値 1 **注）呼び出し元「*APP*」では、必ず書込み点数以上の領域を確保してください。**

＜引数範囲＞

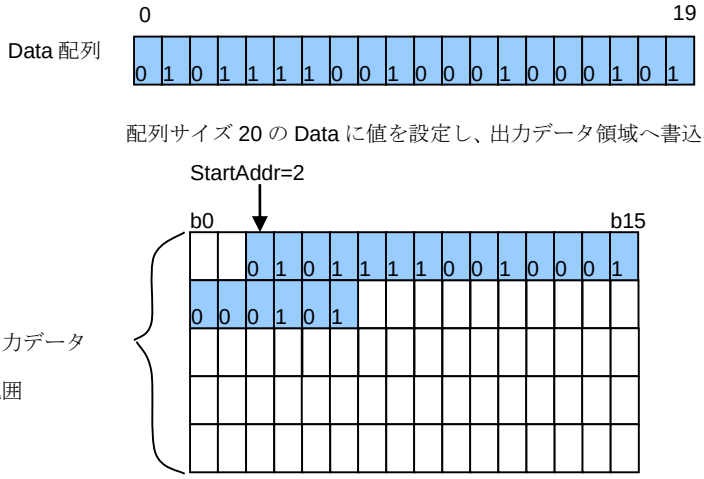
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
StartAddr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

I/O 割付情報に基づき、対象スレーブノードの出力データを連続ビット書込みます。書込み開始アドレスに、I/O 出力データエリア内の相対アドレスを指定して、指定した書込み点数のビットデータを書込みます。

自ノードおよび対象スレーブノードが未参加でも書込み可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。書込み可能な範囲は、自ノード送信領域（HFA_SetCommon 関数で設定）かつ、IO 割付情報（HFA_SetIO 関数で設定）の対象スレーブノードの出力データ範囲です。自ノード送信領域以外に書込みを行う場合、戻り値=引数異常となり、コモンメモリを更新しません。

出力データアドレスの 2 ビット目から 20 点ビットデータを書込む例を以下に示します。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※書込みは成功しています）
5	対象ノード未参加（※書込みは成功しています）

＜使用例＞

１）対象スレーブノードの出力データ領域へデータを書込む。

```
void SampleWriteOutputBitData(){
    char Data[20] = {0,1,0,1,1,1,1,0,0,1,0,0,0,1,0,0,0,1,0,1};
    long lRet;

    // ノード番号１、2 ビット目から 20 点ビットデータを書込む
    lRet = HFA_WriteOutputBitData(1, 2, 20, Data);
}
```

＜関連事項＞

・HFA_WriteOutputWordData 関数

・HFA_WriteOutputRandomBitData 関数

・HFA_WriteOutputRandomWordData 関数

4.6.18. HFA_WriteOutputWordData

出力データの連続ワード書込み

＜関数 I/F＞

long HFA_WriteOutputWordData(long NodeNo, unsigned long StartAddr, unsigned long Size,
unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	書込み先スレーブノード番号
unsigned long	StartAddr	IN	書込み開始アドレス（ワード単位） ※IO 出力データエリア内の相対アドレス
unsigned long	Size	IN	書込み点数（ワード単位）
unsigned short *	Data	IN	書込み先データのポインタ 配列の 1 要素が 1 ワードに対応します。 **注）呼び出し元「*APP*」では、必ず書込み点数以上の領域を確保してください。**

＜引数範囲＞

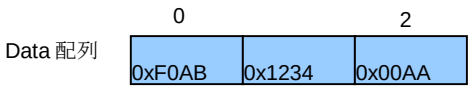
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
StartAddr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

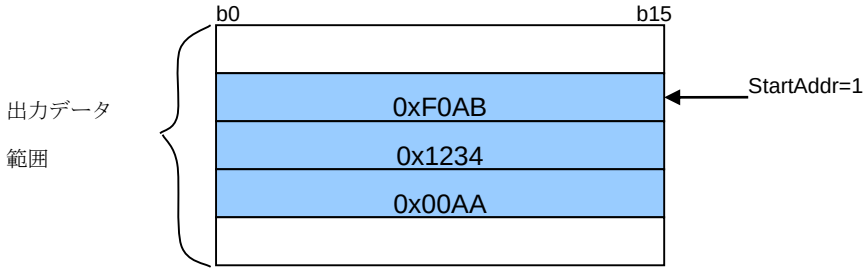
I/O 割付情報に基づき、対象スレーブノードの出力データを連続ワード書込みます。書込み開始アドレスに、I/O 出力データエリア内の相対アドレスを指定して、指定した書込み点数のワードデータを書込みます。

自ノードおよび対象スレーブノードが未参加でも書込み可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。書込み可能な範囲は、自ノード送信領域（HFA_SetCommon 関数で設定）かつ、IO 割付情報（HFA_SetIO 関数で設定）の対象スレーブノードの出力データ範囲です。自ノード送信領域以外に書込みを行う場合、戻り値=引数異常となり、コモンメモリを更新しません。

出力データアドレスの 1 ワード目から 3 ワードデータを書込む例を以下に示します。



配列サイズ 3 の Data に値を設定し、出力データ領域へ書込みします。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※書込みは成功しています）
5	対象ノード未参加（※書込みは成功しています）

＜使用例＞

１）対象スレーブノードの出力データ領域へデータを書込む。

```
void SampleWriteOutputWordData(){
    unsigned short Data[3] = {0xF0AB, 0x1234, 0x00AA};
    long lRet;

    // ノード番号１、1 ワード目から 3 ワードデータを書込む
    lRet = HFA_WriteOutputWordData(1, 1, 3, Data);
}
```

＜関連事項＞

・HFA_WriteOutputBitData 関数

・HFA_WriteOutputRandomBitData 関数

・HFA_WriteOutputRandomWordData 関数

4.6.19. HFA_WriteOutputRandomBitData

出力データのランダムビット書込み

＜関数 I/F＞

long HFA_WriteOutputRandomBitData(long NodeNo, unsigned long *Addr, unsigned long Size,
char *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	書込み先スレーブノード番号
unsigned long *	Addr	IN	書込み先アドレス（ビット単位）の配列 ※I/O 出力データエリア内の相対アドレス **注）呼び出し元「*APP*」では、必ず書込み点数以上の領域を確保してください。**
unsigned long	Size	IN	書込み点数（ビット単位）
char *	Data	IN	書込みデータのポインタ 配列の 1 要素が 1 ビットに対応します。配列要素に以下の値を指定します。 ・0=ビット値 0 ・1=ビット値 1 **注）呼び出し元「*APP*」では、必ず書込み点数以上の領域を確保してください。**

＜引数範囲＞

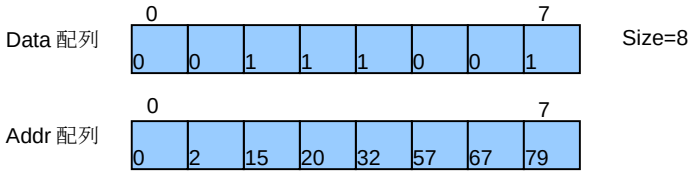
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
Addr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

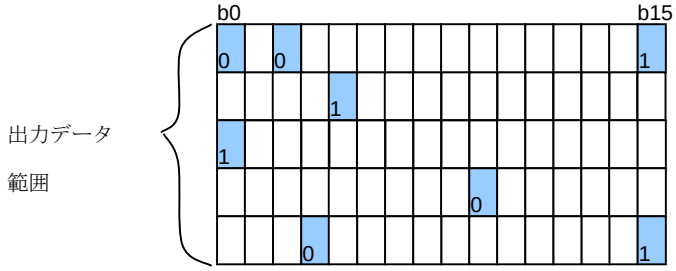
I/O 割付情報に基づき、対象スレーブノードの出力データをランダムビット書込みます。書込みアドレスに、I/O 出力データエリア内の相対アドレスを複数指定して、指定した書込み点数のビットデータを書込みます。

自ノードおよび対象スレーブノードが未参加でも書込み可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。書込み可能な範囲は、自ノード送信領域（HFA_SetCommon 関数で設定）かつ、IO 割付情報（HFA_SetIO 関数で設定）の対象スレーブノードの出力データ範囲です。自ノード送信領域以外に書込みを行う場合、戻り値=引数異常となり、コモンメモリを更新しません。

出力データアドレスからランダムに 8 点ビットデータを書込む例を以下に示します。



Data 配列に値を設定し、Addr 配列に書込み先アドレスを設定し、出力データ領域へ書込みします。Data 配列と Addr 配列の添え字は 1 対 1 対応しています。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※書込みは成功しています）
5	対象ノード未参加（※書込みは成功しています）

＜使用例＞

1）対象スレーブノードの出力データ領域へデータを書込む。

```
void SampleWriteOutputRandomBitData(){
    unsigned long Addr[8] = {0,2,15,20,32,57,67,79};
    char Data[8] = {0,0,1,1,1,0,0,1};
    long lRet;

    // ノード番号１、ランダムに 8 点ビットデータを書込む
    lRet = HFA_WriteOutputRandomBitData(1, Addr, 8, Data);
}
```

＜関連事項＞

・HFA_WriteOutputBitData 関数

・HFA_WriteOutputWordData 関数

・HFA_WriteOutputRandomWordData 関数

4.6.20. HFA_WriteOutputRandomWordData

出力データのランダムワード書込み

＜関数 I/F＞

long HFA_WriteOutputRandomWordData(long NodeNo, unsigned long *Addr, unsigned long Size, unsigned short *Data)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	書込み先スレーブノード番号
unsigned long *	Addr	IN	書込み先アドレス（ワード単位）の配列 ※I/O 出力データエリア内の相対アドレス **注）呼び出し元「*APP*」では、必ず書込み点数以上の領域を確保してください。**
unsigned long	Size	IN	書込み点数（ワード単位）
unsigned short *	Data	IN	書込みデータのポインタ 配列の 1 要素が 1 ワードに対応します。 **注）呼び出し元「*APP*」では、必ず書込み点数以上の領域を確保してください。**

＜引数範囲＞

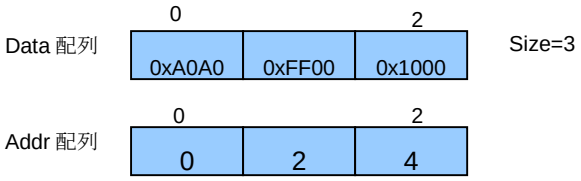
項目(式)	値正常範囲	備 考
NodeNo	1≦(値)≦249	自ノード以外，対象スレーブノードのみ
Addr	0≦(値)，IO 割付設定範囲内	
Size	1≦(値)，IO 割付設定範囲内	

＜詳細＞

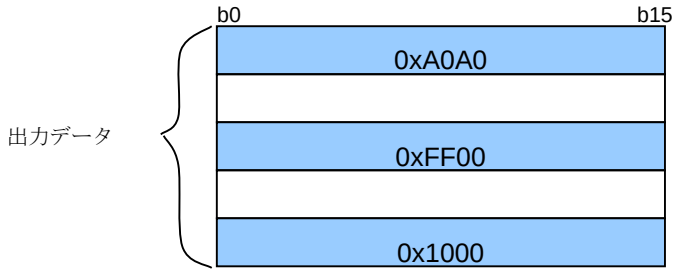
I/O 割付情報に基づき、対象スレーブノードの出力データをランダムビット書込みます。書込みアドレスに、I/O 出力データエリア内の相対アドレスを複数指定して、指定した書込み点数のビットデータを書込みます。

自ノードおよび対象スレーブノードが未参加でも書込み可能ですが、あらかじめ I/O 割付情報を設定しておく必要があります。書込み可能な範囲は、自ノード送信領域（HFA_SetCommon 関数で設定）かつ、IO 割付情報（HFA_SetIO 関数で設定）の対象スレーブノードの出力データ範囲です。自ノード送信領域以外に書込みを行う場合、戻り値=引数異常となり、コモンメモリを更新しません。

出力データアドレスからランダムに 3 ワードデータを書込む例を以下に示します。



Data 配列に値を設定し、Addr 配列に書込み先アドレスを設定し、出力データ領域へ書込みします。Data 配列と Addr 配列の添え字は 1 対 1 対応しています。



＜戻り値＞

値	内 容
0	正常終了
-1	引数異常
2	自ノード未参加　（※書込みは成功しています）
5	対象ノード未参加（※書込みは成功しています）

＜使用例＞

1) 対象スレーブノードの出力データ領域へデータを書込む。

```
void SampleWriteOutputRandomWordData(){
      unsigned long Addr[3] = {0,2,4};
      unsigned short Data[3] = {0xA0A0,0xFF00,0x1000};
      long lRet;

      // ノード番号１、ランダムに 3 ワードデータを書込む
      lRet = HFA_WriteOutputRandomWordData(1, Addr, 3, Data);
}
```

＜関連事項＞

・HFA_WriteOutputBitData 関数

・HFA_WriteOutputWordData 関数

・HFA_WriteOutputRandomBitData 関数

4.7. コールバック関数

「***DLL***」が「***APP***」に同期・非同期でイベントを通知する場合は、「***DLL***」が「***APP***」内に用意されたコールバック関数をコールします。同期・非同期イベントの通知を有効にするためには、「***APP***」側でコールバック関数を実装して、HFA_SetCallback 関数または HFA_SetCallbackV3 関数をコールしてコールバック関数のアドレスを「***DLL***」に登録する必要があります。

コールバック関数が登録されていない状態で、「***DLL***」内で同期・非同期イベントを検知した場合は、コールバック関数はコールされません（イベントは発生しません）。

また、「***APP***」側でコールバック処理実行中は、新たなイベントが発生しても即座にコールバック関数は呼び出されません。

コールバック関数の一覧を以下に示します。

No.	関数名	概要
1	*LinkIn*	ノードが参加した
2	*LinkOut*	ノードが離脱した
3	*CommonRefresh*	コモンメモリの値が変化した
4	*LogClear*	自ノードのログデータがクリアされた
5	*RecvReqReadByteBlock*	バイトブロックリード要求メッセージを受信した
6	*RecvReqWriteByteBlock*	バイトブロックライト要求メッセージを受信した
7	*RecvReqReadWordBlock*	ワードブロックリード要求メッセージを受信した
8	*RecvReqWriteWordBlock*	ワードブロックライトメッセージを受信した
9	*RecvRepReadByteBlock*	バイトブロックリード応答メッセージを受信した
10	*RecvRepWriteByteBlock*	バイトブロックライト応答メッセージを受信した
11	*RecvRepReadWordBlock*	ワードブロックリード応答メッセージを受信した
12	*RecvRepWriteWordBlock*	ワードブロックライト応答メッセージを受信した
13	*RecvRepReadNetParam*	ネットワークパラメータリード応答メッセージを受信した
14	*RecvRepWriteNetParam*	ネットワークパラメータライト応答メッセージを受信した
15	*RecvRepControlEquipment*	運転／停止指令応答メッセージを受信した
16	*RecvRepReadProfile*	プロファイルリード応答メッセージを受信した
17	*RecvRepReadLog*	ログデータリード応答メッセージを受信した
18	*RecvRepClearLog*	ログデータクリア応答メッセージを受信した
19	*RecvRepEchoMessage*	メッセージ折り返し応答を受信した
20	*RecvTransparency*	透過形メッセージメッセージを受信した
21	*LinkInTimeout*	リンク参加タイムアウトが発生した
22	*SendTimeout*	メッセージ送信タイムアウトが発生した
23	*ReplyTimeout*	メッセージ応答受信タイムアウトが発生した
24	*SendComplete*	メッセージの送信が完了した（正常，異常）
25	*Error*	エラーが発生したとき

(次頁へ続く)

(続き)

No.	関数名	概要
26	*RecvReqVendorMessage*	ベンダ固有要求メッセージを受信した
27	*RecvRepVendorMessage*	ベンダ固有応答メッセージを受信した
28	*RecvReqWriteNetParam*	ネットワークパラメータライト要求メッセージ受信
29	*RecvReqControlEquipment*	運転/停止指令要求メッセージ受信
30	*LinkInSlave*	スレーブノード参加
31	*LinkOutSlave*	スレーブノード離脱
32	*InputDataRefresh*	入力データ変更
33	*InputStatusRefresh*	入力ステータス変更
34	*ChangeTokenTimeMeasureStatus*	トークン保持時間測定状態変化
35	*ChangeDataLogMeasureStatus*	汎用通信データ送信元ログ測定状態変化
36	*RecvReqReadByteBlockFromSettingTool*	設定ツールからのバイトブロックリード要求受信
37	*RecvReqReadWordBlockFromSettingTool*	設定ツールからのワードブロックリード要求受信
38	*RecvReqWriteByteBlockFromSettingTool*	設定ツールからのバイトブロックライト要求受信
39	*RecvReqWriteWordBlockFromSettingTool*	設定ツールからのワードブロックライト要求受信
40	*RecvReqWriteNetParamFromSettingTool*	設定ツールからのネットワークパラメータライト要求受信
41	*RecvReqControlEquipmentFromSettingTool*	設定ツールからの運転/停止指令要求受信
42	*RecvReqSetIOFromSettingTool*	設定ツールからの IO 割付設定要求受信
43	*RecvReqSetConfigParamFromSettingTool*	設定ツールからのコンフィギュレーション用パラメータ設定要求受信
44	*RecvReqResetNodeFromSettingTool*	設定ツールからのノードリセット要求受信

4.7.1. ノードの参加

＜関数 I/F＞

void LinkIn (long NodeNo, long Reason)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	FL-net ネットワークに参加したノードの番号（1～254）。 ※参加失敗の場合は、自ノード番号を示します。
long	Reason	IN	参加結果。 ・0＝参加成功 ・1＝ノード No 重複（参加失敗） ・2＝コモンメモリ重複（参加成功） ※この場合、自ノードのコモンメモリ領域は、0 リセットされます。 ・3＝トークンモード不一致（参加失敗）

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

・ 自ノードが FL-net ネットワークに参加したとき
・ 自ノードが FL-net ネットワーク参加中に、異常（ノード番号重複，トークンモード不一致）を検知して参加できなかったとき
・ 他ノードが FL-net ネットワークに途中参加したとき

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、自ノード未参加の場合は、他ノードに対する参加イベントは通知しません。

＜詳細＞

自ノードが FL-net ネットワークに参加する前から、既に参加中の他ノードについては通知しません。自ノード参加イベント発生後、HFA_GetNetworkStatus 関数で参加中ノードの一覧を確認してください。

モニタモード開始の場合は、自ノードの参加イベントは発生しません。他ノードがネットワークに新たに参加したイベントのみを通知します。

＜使用例＞

１）ノード参加時の処理を行う。

```
void CALLBACK LinkIn (long NodeNo, long Reason ){
    if (NodeNo != G_MyNode) {
        // 他ノード参加時の処理
    }
    else if (Reason == 0 || Reason == 2){
        // 自ノード参加時の処理
        long lRet;
        char cNode[256];
        NETWORK xNetwork;
        lRet = HFA_GetNetworkStatus(&xNetwork, cNode);        // 参加中ノードの一覧を取得
    }
    else{
        // 自ノード参加失敗時の処理
    }
}
```

＜関連事項＞

・ HFA_LinkIn 関数，HFA_GetNetworkStatus 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.2. ノードの離脱

＜関数 I/F＞

void LinkOut (long NodeNo, long Reason)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	リンクから離脱したノードの番号（1～254）。
long	Reason	IN	離脱理由。 ・1＝「***APP***」からの HFA_LinkOut コール ・2＝トークン欠落 ・3＝試用版タイムアウト ・4＝セキュリティ異常

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

- HFA_LinkOut 関数コールによって、自ノードを離脱させたとき
- トークン欠落（トークン監視時間タイムアウト／トークン周回抜け）によって、他ノードの離脱を検知したとき
- 自ノードあてのトークンを 3 回連続して受信できなかったとき
- 他ノードの離脱を検知したことにより、FL-net ネットワーク上に存在するノードが自ノードのみになったとき
- プロテクトキーの接続が正しくないとき
- 試用時間を超過したとき

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、自ノード未参加の場合は、他ノードの離脱イベントは通知しません。

＜詳細＞

FL-net ネットワークの状況により、自ノードの離脱を検出する場合もあります。この場合、Reason=2 として自ノードの離脱イベントが発生します。FL-net ネットワークに再参加する場合は、HFA_LinkIn 関数または HFA_LinkInDefault 関数をコールしてください。

モニタモード終了の場合は、自ノードの離脱イベントは発生しません。他ノードがネットワークから新たに離脱したイベントのみを通知します。

＜使用例＞

１）自ノード離脱時、FL-net ネットワークに再参加する。

```
void CALLBACK LinkOut (long NodeNo, long Reason ){
    if (NodeNo != G_MyNode) {
        // 他ノード離脱時の処理
    }
    else if (Reason == 2){
        // トークン欠落による自ノード離脱時の処理
        long lRet = HFA_LinkInDefault();        // FL-net ネットワークに再参加
    }
    else{
        // HFA_LinkOut コールによる自ノード離脱時の処理
    }
}
```

＜関連事項＞

・ HFA_LinkOut 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.3. コモンメモリ更新

＜関数 I/F＞

void CommonRefresh (long NodeNo, long Area, long Reason)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	コモンメモリが変化したノード番号（1～254）。
long	Area	IN	コモンメモリが変化した領域。変化を検出した領域に応じて以下の値を論理和演算した結果を通知します。 ・1＝コモンメモリ領域 1 ・2＝コモンメモリ領域 2 ※領域 1 および領域 2 の両方が変化した場合、値は 3 となります。
long	Reason	IN	コモンメモリ変化理由。 ・ 0=トークン受信 ・ 1=自ノードコモンメモリエリア書込み

＜イベント発生＞

HFA_SetCommonRefreshDegree 関数にてコモンメモリ更新イベントの検知範囲が設定されているコモンメモリ領域において、以下の条件を満たす場合にイベントが発生します。

・ HFA_WriteCommon 関数コールによって、コモンメモリの値が更新されたとき
・ 他ノードから受信したサイクリックデータをコモンメモリにコミット（反映）させる時点で、受信した値が受信前の値と異なっているとき

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、自ノード未参加の場合は、イベント通知しません。

＜詳細＞

本関数コールバック時に HFA_ReadCommon 関数をコールすることで、「***APP***」はコモンメモリの最新の値を読出すことができます。HFA_SetCommonRefreshDegree 関数で更新イベントの検知範囲を限定することができます。

＜使用例＞

１）コモンメモリ更新時の処理を行う。

```
void CALLBACK CommonRefresh (long NodeNo, long Area, long Reason ){
    if (Area & 0x01) {          // 領域 1 の変化判定
        unsigned char ucData;
        if (HFA_ReadCommon1(0, 1, &ucData, 0, NodeNo) == 0){// コモンメモリの値を読出す。
            printf("%d¥n", ucData);              // 値を画面に表示する。
        }
    }
    if (Area & 0x02) {          // 領域 2 の変化判定
        unsigned short usData;
        if (HFA_ReadCommon2(0, 1, &usData, 0, NodeNo) == 0){// コモンメモリの値を読出す。
           printf("%d¥n", usData);               // 値を画面に表示する。
        }
    }
}
```

＜関連事項＞

・ HFA_SetCommonRefreshDegree 関数

・ HFA_ReadCommon?関数，HFA_WriteCommon?関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.4. ログデータクリア

＜関数 I/F＞

void LogClear (long NodeNo, long Reason)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	ログクリアを要求したノード番号（1～254）。
long	Reason	IN	ログクリア理由。 ・ 0=他ノードからのログクリア要求受信によるクリア ・ 1=「***APP***」からの自ノードログクリア

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

・ HFA_ClearMyNodeLog 関数をコールして、自ノードのログデータをクリアしたとき

・ 自ノード番号を指定して HFA_ReqClearLog 関数をコールしたとき

・ 他ノードからログクリア要求メッセージを受信したとき

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、自ノード未参加の場合は、イベント通知しません。

＜詳細＞

他ノードからログデータクリア要求を受信した場合、「***DLL***」は自ノードのログ情報を自動的にクリアし、イベントを通知します。

＜関連事項＞

・ HFA_ClearMyNodeLog 関数，HFA_ReqClearLog 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.5. バイトブロックリード要求受信

＜関数 I/F＞

void RecvReqReadByteBlock (long NodeNo, unsigned long Addr, long Bytes)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求元ノード番号（1～254）。
unsigned long	Addr	IN	バイトブロックの読出し開始アドレス（バイト単位）。
long	Bytes	IN	バイトブロックの読出しデータサイズ（バイト単位）。

＜イベント発生＞

他ノードからバイトブロックリード要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、バイトブロックリード応答メッセージを送信する（HFA_RepReadByteBlock 関数をコールする）必要があります。

＜使用例＞

１）バイトブロックリード要求受信時の処理を行う。

```
void CALLBACK RecvReqReadByteBlock(long NodeNo, long Addr, long Bytes){
    long lRet;
    if (…) {                    // 通知される引数情報を判断
        // バイトブロックリード応答（正常応答）
        lRet = HFA_RepReadByteBlock(NodeNo, 0, Addr, Bytes, &G_ByteBlock[Addr], 0, 0);
    }
    else{
        unsigned char ucError=1;    // エラーコード=1
        // バイトブロックリード応答（異常応答）
        lRet = HFA_RepReadByteBlock(NodeNo, 1, Addr, Bytes, 0, 1, &ucError);
    }
}
```

＜関連事項＞

・ HFA_RepReadByteBlock 関数
・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.6. バイトブロックライト要求受信

＜関数 I/F＞

*void RecvReqWriteByteBlock (long NodeNo, unsigned long Addr, long Bytes, unsigned char *Data)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求元ノード番号（1～254）。
unsigned long	Addr	IN	バイトブロックの書込み開始アドレス（バイト単位）。
long	Bytes	IN	バイトブロックの書込みデータサイズ（バイト単位）。
unsigned char *	Data	IN	書込みデータの先頭ポインタ。

＜イベント発生＞

他ノードからバイトブロックライト要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、バイトブロックライト応答メッセージを送信する（HFA_RepWriteByteBlock 関数をコールする）必要があります。

＜使用例＞

1）バイトブロックライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteByteBlock(long NodeNo, long Addr, long Bytes, unsigned char *Data){
    long lRet;
    if (…) {                    // 通知される引数情報を判断
        memcpy(&G_ByteBlock[Addr], Data, Bytes);    // バイトブロックデータの更新
        // バイトブロックライト応答（正常応答）
        lRet = HFA_RepWriteByteBlock(NodeNo, 0, Addr, Bytes, 0, 0);
    }
    else{
        unsigned char ucError=1;    // エラーコード=1
        // バイトブロックライト応答（異常応答）
        lRet = HFA_RepWriteByteBlock(NodeNo, 1, Addr, Bytes, 1, &ucError);
    }
}
```

＜関連事項＞

・ HFA_RepWriteByteBlock 関数
・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.7. ワードブロックリード要求受信

＜関数 I/F＞

void RecvReqReadWordBlock (long NodeNo, unsigned long Addr, long Words)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求元ノード番号（1～254）。
unsigned long	Addr	IN	ワードブロックの読出し開始アドレス（ワード単位）。
long	Words	IN	ワードブロックの読出しデータサイズ（ワード単位）。

＜イベント発生＞

他ノードからワードブロックリード要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、ワードブロックリード応答メッセージを送信する（HFA_RepReadWordBlock 関数をコールする）必要があります。

＜使用例＞

１）ワードブロックの読出し要求受信時の処理を行う。

```
void CALLBACK RecvReqReadWordBlock(long NodeNo, long Addr, long Words){
    long lRet;
    if (…) {                    // 通知される引数情報を判断
        // ワードブロックリード応答（正常応答）
        lRet = HFA_RepReadWordBlock(NodeNo, 0, Addr, Words,
            (unsigned char *)&G_WordBlock[Addr], 0, 0);
    }
    else{
        unsigned char ucError=1;    // エラーコード=1
        // ワードブロックリード応答（異常応答）
        lRet = HFA_RepReadWordBlock(NodeNo, 1, Addr, Words, 0, 1, &ucError);
    }
}
```

＜関連事項＞

・ HFA_RepReadWordBlock 関数
・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.8. ワードブロックライト要求受信

＜関数 I/F＞

*void RecvReqWriteWordBlock (long NodeNo, unsigned long Addr, long Words, unsigned char *Data)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求元ノード番号（1～254）。
unsigned long	Addr	IN	ワードブロックの書込み開始アドレス（ワード単位）。
long	Words	IN	ワードブロックの書込みデータサイズ（ワード単位）。
unsigned char *	Data	IN	書込みデータの先頭ポインタ。 注）「***APP***」では、ワード単位（=2 バイト単位）に型変換してから、アクセスしてください。

＜イベント発生＞

他ノードからワードブロックライト要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、ワードブロックライト応答メッセージを送信する（HFA_RepWriteWordBlock 関数をコールする）必要があります。

＜使用例＞

1）ワードブロックライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteWordBlock(long NodeNo, long Addr, long Words, unsigned char *Data){
    long lRet;
    if (…) {                    // 通知される引数情報を判断
        memcpy(&G_WordBlock[Addr], Data, Words * 2);        // ワードブロックデータの更新
        // ワードブロックライト応答（正常応答）
        lRet = HFA_RepWriteWordBlock(NodeNo, 0, Addr, Words, 0, 0);
    }
    else{
        unsigned char ucError=1;
        // ワードブロックライト応答（異常応答）
        lRet = HFA_RepWriteWordBlock(NodeNo, 1, Addr, Words, 1, &ucError);
    }
}
```

＜関連事項＞

- HFA_RepWriteWordBlock 関数
- HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.9. ベンダ固有要求メッセージ受信

＜関数 I/F＞

*void RecvReqVendorMessage (long NodeNo, long Multi, char *VendorName, char *SubCode,*
*long Bytes, unsigned char *Data)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求元ノード番号（1～254）。
long	Multi	IN	1 対 n メッセージ通信の判定。 ・ 0=1 対 1 メッセージ ・ 1=1 対 n メッセージ
char *	VendorName	IN	ベンダ名（データサイズ=10 バイト）。
char *	SubCode	IN	サブコード（データサイズ=6 バイト）。
long	Bytes	IN	要求データサイズ（バイト単位）。
unsigned char *	Data	IN	要求データの先頭ポインタ。

＜イベント発生＞

他ノードからベンダ固有要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、ベンダ固有応答メッセージを送信する（HFA_RepVendorMessage 関数をコールする）必要があります。

・要求されたベンダ名をサポートしていない場合は、結果=非実装で応答してください。

・要求されたサブコードをサポートしていない場合は、結果=異常で応答してください。

注）要求メッセージが 1 対 n メッセージ（Multi 引数=1）の場合は、ベンダ固有応答メッセージを送信しないでください。

＜使用例＞

１）ベンダ固有要求メッセージ受信時の処理を行う。

```
void CALLBACK RecvReqVendorMessage(long NodeNo, long Multi, char *VendorName, char *SubCode,
        long Bytes, unsigned char *Data){
    long lRet;
    if (Multi == 0){            // 1 対 1 メッセージの場合のみ、応答する。
        if(…) {                 // 正常応答
            // ベンダ固有応答（正常応答）
            lRet = HFA_RepVendorMessage(NodeNo, 0, VendorName, SubCode, 4, “Data”);
        }
        else if(…){             // 異常応答（要求のサブコードをサポートしていない場合等）
            unsigned char ucError=1;
            // ベンダ固有応答（異常応答）
            lRet = HFA_RepVendorMessage(NodeNo, 1, VendorName, SubCode, 1, &ucError);
        }
        else{                   // 非実装応答（要求のベンダ名をサポートしていない場合等）
            // ベンダ固有応答（非実装応答）
            lRet = HFA_RepVendorMessage(NodeNo, 2, VendorName, SubCode, 0, 0);
        }
    }
}
```

＜関連事項＞

・ HFA_RepVendorMessage 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.10. バイトブロックリード応答受信

＜関数 I/F＞

void RecvRepReadByteBlock (long NodeNo, long Result, unsigned long Addr,
*long Bytes, unsigned char *Data, long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。 ・ 0=正常 ・ 1=異常 ・ 2=非実装
unsigned long	Addr	IN	バイトブロックの読出し開始アドレス（バイト単位）。
long	Bytes	IN	バイトブロックの読出しデータサイズ（バイト単位）。
unsigned char *	Data	IN	受信したバイトデータの先頭ポインタ。応答種別=正常の場合、値が有効です。
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答種別=異常の場合、値が有効です。

＜イベント発生＞

他ノードからバイトブロックリード応答メッセージを受信したときにイベントが発生します。

注）事前にバイトブロックリード要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、バイトブロックリード要求（HFA_ReqReadByteBlock 関数）との突き合わせ（受信バイト数の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

- Result=0 の場合：応答元ノードがバイトブロックリード要求メッセージに正常応答し、Data 引数にバイトブロックリードデータの内容を通知します。
- Result=1 の場合：応答元ノードがバイトブロックリード要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。
- Result=2 の場合：応答元ノードがバイトブロックリード要求メッセージを実装していないことを示します。

＜使用例＞

１）バイトブロックリード応答受信時の処理を行う。

```
void CALLBACK RecvRepReadByteBlock(long NodeNo, long Result, unsigned long Addr, long Bytes,
        unsigned char *Data, long ErrBytes, unsigned char *Error){
    long lRet;
    if (…) {                                // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                    // 正常応答の場合
            memcpy(&G_ByteBlock[NodeNo][Addr], Data, Bytes);          // リード結果を待避
        }
        else if(Result == 1){               // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);                          // エラーコード待避
            printf("バイトブロックリード異常発生。[エラー内容=%s]¥n", cError);    // エラー表示
        }
    }
}
```

＜関連事項＞

・ HFA_ReqReadByteBlock 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.11. バイトブロックライト応答受信

＜関数 I/F＞

void RecvRepWriteByteBlock (long NodeNo, long Result, unsigned long Addr, long Bytes,
*long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。 ・ 0=正常 ・ 1=異常 ・ 2=非実装
unsigned long	Addr	IN	バイトブロックの書込み先頭アドレス（バイト単位）。
long	Bytes	IN	バイトブロックの書込みデータサイズ（バイト単位）。
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答種別=異常の場合、値が有効です。

＜イベント発生＞

他ノードからバイトブロックライト応答メッセージを受信したときにイベントが発生します。

注）事前にバイトブロックライト要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、バイトブロックライト要求（HFA_ReqWriteByteBlock 関数）との突き合わせ（受信バイト数の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

・ Result=0 の場合：応答元ノードがバイトブロックライト要求メッセージに正常応答したことを示します。
・ Result=1 の場合：応答元ノードがバイトブロックライト要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。
・ Result=2 の場合：応答元ノードがバイトブロックライト要求メッセージを実装していないことを示します。

＜使用例＞

１）バイトブロックライト応答受信時の処理を行う。

```
void CALLBACK RecvRepWriteByteBlock(long NodeNo, long Result, unsigned long Addr, long Bytes,
        long ErrBytes, unsigned char *Error){
    if (…) {                    // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                // 正常応答の場合
            printf(“バイトブロックライト完了¥n”);
        }
        else if(Result == 1){           // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);            // エラーコード待避
            printf(“バイトブロックライト異常発生。[エラー内容=%s]¥n”, cError);    // エラー表示
        }
    }
}
```

＜関連事項＞

・ HFA_ReqWriteByteBlock 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.12. ワードブロックリード応答受信

＜関数 I/F＞

void RecvRepReadWordBlock (long NodeNo, long Result, unsigned long Addr, long Words,
*unsigned char *Data, long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。 ・ 0=正常 ・ 1=異常 ・ 2=非実装
unsigned long	Addr	IN	ワードブロックの読出し先頭アドレス（ワード単位）。
long	Words	IN	ワードブロックの読出しデータサイズ（ワード単位）。
unsigned char *	Data	IN	受信したワードデータの先頭ポインタ。応答種別=正常の場合、値が有効です。 **注）「*APP*」では、ワード単位（=2 バイト単位）に型変換してから、アクセスしてください。**
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答種別=異常の場合、値が有効です。

＜イベント発生＞

他ノードからワードブロックリード応答メッセージを受信したときにイベントが発生します。

注）事前にワードブロックリード要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、ワードブロックリード要求（HFA_ReqReadWordBlock 関数）との突き合わせ（受信バイト数の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

- Result=0 の場合：応答元ノードがワードブロックリード要求メッセージに正常応答し、Data 引数にワードブロックリードデータの内容を通知します。
- Result=1 の場合：応答元ノードがワードブロックリード要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。

・ Result=2 の場合：応答元ノードがワードブロックリード要求メッセージを実装していないことを示します。

＜使用例＞

1）ワードブロックリード応答受信時の処理を行う。

```
void CALLBACK RecvRepReadWordBlock(long NodeNo, long Result, unsigned long Addr, long Words,
        unsigned char *Data,
    long ErrBytes, unsigned char *Error){
    long lRet;
    if (…) {                            // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                // 正常応答の場合
            memcpy(&G_WordBlock[NodeNo][Addr], Data, Words * 2);      // リード結果を待避
        }
        else if(Result == 1){           // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);                          // エラーコード待避
            printf("ワードブロックリード異常発生。[エラー内容=%s]¥n", cError);    // エラー表示
        }
    }
}
```

＜関連事項＞

・ HFA_ReqReadWordBlock 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.13. ワードブロックライト応答受信

＜関数 I/F＞

void RecvRepWriteWordBlock (long NodeNo, long Result, unsigned long Addr, long Words,
*long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。 ・ 0=正常 ・ 1=異常 ・ 2=非実装
unsigned long	Addr	IN	ワードブロックの書込み先頭アドレス（ワード単位）。
long	Words	IN	ワードブロックの書込みデータサイズ（ワード単位）。
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答種別=異常の場合、値が有効です。

＜イベント発生＞

他ノードからワードブロックライト応答メッセージを受信したときにイベントが発生します。

注）事前にワードブロックライト要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、ワードブロックライト要求（HFA_ReqWriteWordBlock 関数）との突き合わせ（受信バイト数の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

- Result=0 の場合：応答元ノードがワードブロックライト要求メッセージに正常応答したことを示します。
- Result=1 の場合：応答元ノードがワードブロックライト要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。
- Result=2 の場合：応答元ノードがワードブロックライト要求メッセージを実装していないことを示します。

＜使用例＞

１）ワードブロックライト応答受信時の処理を行う。

```
void CALLBACK RecvRepWriteWordBlock(long NodeNo, long Result, unsigned long Addr, long Words,
        long ErrBytes, unsigned char *Error){
    if (…) {                            // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                // 正常応答の場合
            printf(“ワードブロックライト完了¥n”);
        }
        else if(Result == 1){           // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);  // エラーコード待避
            printf(“ワードブロックライト異常発生。[エラー内容=%s]¥n”, cError);    // エラー表示
        }
    }
}
```

＜関連事項＞

・ HFA_ReqWriteWordBlock 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.14. ネットワークパラメータリード応答受信

＜関数 I/F＞

*void RecvRepReadNetParam (long NodeNo, long Result, long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。※非実装応答はありません。 ・ 0=正常 ・ 1=異常
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答種別=異常の場合、値が有効です。

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

・ 自ノード番号を指定して HFA_ReqClearLog 関数をコールしたとき

・ 他ノードからネットワークパラメータリード応答メッセージを受信したとき

注）事前にネットワークパラメータリード要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、ネットワークパラメータリード要求（HFA_ReqReadNetParam 関数）との突き合わせ（ノード番号の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

・ Result=0 の場合：応答元ノードがネットワークパラメータリード要求メッセージに正常応答したことを示します。この場合、「***DLL***」は応答元ノード情報に対する内部メモリを更新します。「***APP***」側では、HFA_GetNodeStatus 関数をコールすることで、最新のノード情報データを取得することが可能です。

・ Result=1 の場合：応答元ノードがネットワークパラメータリード要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。

＜使用例＞

１）ネットワークパラメータのリード応答受信時に、応答ノードのノード情報を取得する。

```
void CALLBACK RecvRepReadNetParam(long NodeNo, long Result, long ErrBytes, unsigned char *Error){
    if(Result == 0){
        long lNodeNo = NodeNo;
        NODE xNode;
        // 応答ノードの管理情報パラメータ読出し
        if(HFA_GetNodeStatus(&lNodeNo, &xNode) == 0){
            printf("ノード番号=%d のノード名称=%s¥n", xNode.NodeName);          // ノード名称を参照
        }
    }
}
```

＜関連事項＞

・ HFA_ReqReadNetParam 関数，HFA_GetNodeStatus 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.15. ネットワークパラメータライト応答受信

＜関数 I/F＞

*void RecvRepWriteNetParam (long NodeNo, long Result, long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。 ・ 0=正常 ・ 1=異常 ・ 2=非実装
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答種別=異常の場合、値が有効です。

＜イベント発生＞

他ノードからネットワークパラメータライト応答メッセージを受信したときにイベントが発生します。

注）事前にネットワークパラメータライト要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合には、イベント通知しません。

＜詳細＞

「***DLL***」は、ネットワークパラメータライト要求（HFA_ReqWriteNetParam 関数）との突き合わせ（ノード番号の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

- Result=0 の場合：応答元ノードがネットワークパラメータライト要求メッセージに正常応答したことを示します。
- Result=1 の場合：応答元ノードがネットワークパラメータライト要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。
- Result=2 の場合：応答元ノードがネットワークパラメータライト要求メッセージを実装していないことを示します。

＜使用例＞

１）ネットワークパラメータライト応答受信時の処理を行う。

```
void CALLBACK RecvRepWriteNetParam (long NodeNo, long Result, long ErrBytes, unsigned char *Error){
    if (…) {                                    // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                        // 正常応答の場合
            printf(“正常応答¥n”);
        }
        else if(Result == 1){                   // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);                    // エラーコード待避
            printf(“異常応答[エラー内容=%s]¥n”, cError);        // エラー表示
        }
        else if(Result == 2){                   // 非実装応答の場合
            printf(“非実装応答¥n”);
        }
    }
}
```

＜関連事項＞

・ HFA_ReqWriteNetParam 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.16. 運転／停止指令応答受信

＜関数 I/F＞

void RecvRepControlEquipment (long NodeNo, long Command, long Result,
*long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Command	IN	制御内容。以下の値が格納されます。 ・ 1＝運転指令 ・ 0＝停止指令
long	Result	IN	応答種別。以下の値が格納されます。 ・ 0=正常 ・ 1=異常 ・ 2=非実装
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答種別=異常の場合、値が有効です。

＜イベント発生＞

他ノードから運転／停止指令の応答メッセージを受信したときにイベントが発生します。

注）事前に運転／停止要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、運転／停止指令要求（HFA_ReqControlEquipment 関数）との突き合わせ（ノード番号の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

・ Result=0 の場合：応答元ノードが運転／停止指令要求メッセージに正常応答したことを示します。
・ Result=1 の場合：応答元ノードが運転／停止指令要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。
・ Result=2 の場合：応答元ノードが運転／停止指令要求メッセージを実装していないことを示します。

＜使用例＞

１）運転／停止指令応答受信時の処理を行う。

```
void CALLBACK RecvRepControlEquipment (long NodeNo, long Command, long Result, long ErrBytes,
        unsigned char *Error){
    if (…) {                            // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                // 正常応答の場合
            printf(“正常応答¥n”);
        }
        else if(Result == 1){           // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);                    // エラーコード待避
            printf(“異常応答[エラー内容=%s]¥n”, cError);        // エラー表示
        }
        else if(Result == 2){           // 非実装応答の場合
            printf(“非実装応答¥n”);
        }
    }
}
```

＜関連事項＞

・ HFA_ReqControlEquipment 関数
・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.17. プロファイルリード応答受信

＜関数 I/F＞

*void RecvRepReadProfile (long NodeNo, long Result, long Size, unsigned char *Data)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。※非実装応答はありません。 ・ 0=正常 ・ 1=異常
long	Size	IN	プロファイル情報のバイト数（最大 1024 バイト）。
unsigned char *	Data	IN	プロファイル情報の先頭ポインタ。応答元ノードより受信したプロファイル情報を透過で格納します。「***APP***」では、プロファイル情報を分析する必要があります。 応答種別=異常の場合は、エラー情報を示します。

＜イベント発生＞

他ノードからプロファイルリード応答メッセージを受信したときにイベントが発生します。

注）事前にプロファイルリード要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、プロファイルリード要求（HFA_ReqReadProfile 関数）との突き合わせ（ノード番号の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

- Result=0 の場合：応答元ノードがプロファイルリード要求メッセージに正常応答し、Size 引数および Data 引数に応答元ノードからのプロファイル情報を通知します。
- Result=1 の場合：応答元ノードがプロファイルリード要求メッセージに異常応答し、Size 引数および Data 引数に応答元ノードからのエラー情報を通知します。

＜使用例＞

１）プロファイルリード応答受信時の処理を行う。

```
void CALLBACK RecvRepReadProfile (long NodeNo, long Result, long Size, unsigned char *Data){
    if (…) {                                    // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                        // 正常応答の場合
            // Size および Data 引数を参照して、プロファイル情報を解析する
            printf(“正常応答¥n”);
        }
        else if(Result == 1){                   // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Data, Size);                         // エラーコード待避
            printf(“異常応答[エラー内容=%s]¥n”, cError);        // エラー表示
        }
    }
}
```

＜関連事項＞

・ HFA_ReqReadProfile 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.18. ログデータリード応答受信

＜関数 I/F＞

*void RecvRepReadLog (long NodeNo, long Result, long Size, unsigned char *Log)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。※非実装応答はありません。 ・ 0=正常 ・ 1=異常
long	Size	IN	ログ情報のバイト数（最大 1024 バイト）。
unsigned char *	Log	IN	ログ情報の先頭ポインタ。応答元ノードより受信したログ情報を透過で格納します。「***APP***」では、応答元ノードのログ仕様に基づき、ログ情報を分析する必要があります。（自ノードのログ仕様は、付録 6 をご参照ください。） ログ情報は、4 バイト単位のデータです。データのアクセス方法につきましては、使用例をご参照ください。 応答種別=異常の場合は、エラー情報を示します。

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

・ 自ノード番号を指定して HFA_ReqReadLog 関数をコールしたとき

・ 他ノードからログデータリード応答メッセージを受信したとき

注）事前にログデータリード要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、ログリード要求（HFA_ReqReadLog 関数）との突き合わせ（ノード番号の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

・ Result=0 の場合：応答元ノードがログデータリード要求メッセージに正常応答し、Size 引数および Log 引数に応答元ノードからのログ情報を通知します。

・ Result=1 の場合：応答元ノードがログデータリード要求メッセージに異常応答し、Size 引数および Log 引数に応答元ノードからのエラー情報を通知します。

＜使用例＞

１）ログデータリード応答受信時の処理を行う。

```
void CALLBACK RecvRepReadLog(long NodeNo, long Result, long Size, unsigned char *Log){
    if (…) {                    // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                // 正常応答の場合
            unsigned long ulLog[128];
            memcpy(ulLog, Log, Size);          // unsigned long 型に変換
            printf(“通算ソケット部送信回数=%lu¥n”, ulLog[0]);    // ログ情報参照
            printf(“加入回数=%lu¥n”, ulLog[74]);                 // ログ情報参照
        }
        else if(Result == 1){           // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Log, Size);                           // エラーコード待避
            printf(“異常応答[エラー内容=%s]¥n”, cError);          // エラー表示
        }
    }
}
```

＜関連事項＞

・ HFA_ReqReadLog 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.19. ログデータクリア応答受信

＜関数 I/F＞

*void RecvRepClearLog (long NodeNo, long Result, long ErrBytes, unsigned char *Error)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。※非実装応答はありません。 ・ 0=正常 ・ 1=異常
long	ErrBytes	IN	エラー情報のバイト数。
unsigned char *	Error	IN	エラー情報の先頭ポインタ。応答結果=異常の場合、値が有効です。

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

・ 自ノード番号を指定して HFA_ReqClearLog 関数をコールしたとき

・ 他ノードからログデータクリア応答メッセージを受信したとき

注）事前にログデータクリア要求メッセージを送信したノード以外から応答メッセージを受信した場合もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、ログデータクリア要求（HFA_ReqClearLog 関数）との突き合わせ（ノード番号の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

・ Result=0 の場合：応答元ノードがログデータクリア要求メッセージに正常応答したことを示します。

・ Result=1 の場合:応答元ノードがログデータクリア要求メッセージに異常応答し、ErrBytes 引数および Error 引数に応答元ノードからのエラー情報を通知します。

＜使用例＞

１）ログデータクリア応答受信時の処理を行う。

```
void CALLBACK RecvRepClearLog (long NodeNo, long Result, long ErrBytes, unsigned char *Error){
    if (…) {                                  // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){                      // 正常応答の場合
            printf(“正常応答¥n”);
        }
        else if(Result == 1){                 // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Error, ErrBytes);              // エラーコード待避
          printf(“異常応答[エラー内容=%s]¥n”, cError);  // エラー表示
        }
    }
}
```

＜関連事項＞

・ HFA_ReqClearLog 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.20. メッセージ折り返し応答受信

＜関数 I/F＞

*void RecvRepEchoMessage (long NodeNo, long Bytes, unsigned char *Message)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Bytes	IN	受信メッセージのバイト数。
unsigned char *	Message	IN	受信メッセージの先頭ポインタ。

＜イベント発生＞

他ノードから折り返しメッセージ応答を受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、メッセージ折り返し要求（HFA_ReqEchoMessage 関数）との突き合わせ（メッセージ内容の一致チェックなど）は行わずに、受信した応答メッセージを透過で通知します。「***APP***」では、要求メッセージとの比較を行う必要があります。

＜使用例＞

１）メッセージ折返し応答受信時の処理を行う。

```
void CALLBACK RecvRepEchoMessage(long NodeNo, long Bytes, unsigned char *Message){
    if(NodeNo == 1 && Bytes == 5){                  // 応答ノード番号，バイト数チェック
        if(strcmp(Message, “12345”) == 0){          // 応答データ内容チェック
            printf(“メッセージ折返し内容=OK”);      // 内容一致
        }
        else{
            printf(“折返しメッセージ不一致”);       // 内容不一致
        }
    }
}
```

＜関連事項＞

・ HFA_ReqEchoMessage 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.21. ベンダ固有応答メッセージ受信

＜関数 I/F＞

*void RecvRepVendorMessage (long NodeNo, long Result, char *VendorName, char *SubCode, long Size, unsigned char *Data)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	Result	IN	応答種別。以下の値が格納されます。 ・ 0=正常 ・ 1=異常 ・ 2=非実装
char *	VendorName	IN	ベンダ名（データサイズ=10 バイト）。
char *	SubCode	IN	サブコード（データサイズ=6 バイト）。
long	Bytes	IN	応答データのバイト数（最大 1024 バイト）。
unsigned char *	Data	IN	応答データの先頭ポインタ。応答元ノードより受信したデータを透過で格納します。応答種別=異常の場合は、エラーコードを示します。

＜イベント発生＞

他ノードからベンダ固有応答メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***DLL***」は、ベンダ固有要求メッセージ（HFA_ReqVendorMessage 関数）との突き合わせ（ノード番号の一致チェックなど）は行わずに、受信した応答内容を透過で通知します。「***APP***」では、要求と応答の関連性を判断する必要があります。

Result は応答元ノードから返される応答種別です。

- Result=0 の場合：応答元ノードがベンダ固有要求メッセージに正常応答し、Bytes 引数および Data 引数に応答データの内容を通知します。
- Result=1 の場合：応答元ノードがベンダ固有要求メッセージに異常応答し、Bytes 引数および Data 引数に応答元ノードからのエラー情報を通知します。
- Result=2 の場合：応答元ノードがベンダ固有要求メッセージを実装していないことを示します。

＜使用例＞

１）ベンダ固有応答メッセージ受信時の処理を行う。

```
void CALLBACK RecvRepVendorMessage (long NodeNo, long Result, char *VendorName, char *SubCode,
        long Bytes, unsigned char *Data){
    if (…) {                          // 通知される引数情報より、要求との関連性を判断
        if(Result == 0){              // 正常応答の場合
            char cData[1025];
            memset(cData, 0, sizeof(cData));
            memcpy(cData, Data, Bytes);                    // 応答データ待避
            printf("正常応答[データ=%s]¥n", cData);
        }
        else if(Result == 1){         // 異常応答の場合
            char cError[1025];
            memset(cError, 0, sizeof(cError));
            memcpy(cError, Data, Bytes);                   // エラーコード待避
            printf("異常応答[エラー内容=%s]¥n", cError);    // エラー表示
        }
        else if(Result == 2){         // 非実装応答の場合
            printf("非実装応答¥n");
        }
    }
}
```

＜関連事項＞

・ HFA_RepVendorMessage 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.22. 透過形メッセージ受信

＜関数 I/F＞

*void RecvTransparency (long NodeNo, long TransactionCode, long Bytes, unsigned char *Message)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	応答元ノード番号（1～254）。
long	TransactionCode	IN	受信メッセージ番号（0～59999）。
long	Bytes	IN	受信バイト数（最大 1024 バイト）。
unsigned char *	Message	IN	受信メッセージ内容の先頭ポインタ。

＜イベント発生＞

他ノードから透過形メッセージを受信したときにイベントが発生します。

注）0～9999 のリザーブメッセージを受信した際もイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

他ノードから透過形メッセージを受信した内容を「***APP***」に通知します。1 対 n メッセージの透過形メッセージを受信した場合も、「***APP***」に通知します。

＜使用例＞

1）透過形メッセージ受信時の処理を行う。

```
void CALLBACK RecvTransparency (long NodeNo, long TransactionCode, long Bytes, unsigned char
        *Message){
    printf("ノード%ld から TCD=%ld の透過形メッセージを受信¥n", NodeNo, TransactionCode);
}
```

＜関連事項＞

・ HFA_SendTransparency 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.23. リンク参加タイムアウト

＜関数 I/F＞

void LinkInTimeout (long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	自ノード番号（1～254）。

＜イベント発生＞

HFA_LinkIn 関数をコールしてから、ネットワーク参加タイムアウト時間以内に、FL-net ネットワークに参加できなかったときにイベントが発生します。タイムアウト時間は、HFA_SetTimeout 関数の LinkIn 引数で設定します。なお、タイムアウト時間が 0 の場合、イベントは発生しません。

＜イベント通知先＞

コールバック関数が登録されている参加待機中の「***APP***」。ただし、モニタモードの場合には通知しません。

＜詳細＞

リンク参加タイムアウトが発生した場合は、FL-net ネットワークへの参加処理を中断します。FL-net ネットワークに再参加する場合は、HFA_LinkIn 関数または HFA_LinkInDefault 関数をコールしてください。

＜使用例＞

１）リンク参加タイムアウト発生時に再参加処理を行う。

```
void CALLBACK LinkInTimeout (long NodeNo){
    printf("リンク参加タイムアウト発生¥n");
    // FL-net ネットワーク再参加
    long lRet = HFA_LinkInDefault();
}
```

＜関連事項＞

- HFA_SetTimeout 関数
- HFA_LinkIn 関数，HFA_LinkInDefault 関数
- HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.24. メッセージ送信タイムアウト

＜関数 I/F＞

void SendTimeout (long NodeNo, long TransactionCode)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	送信できなかった送信先ノード番号。
long	TransactionCode	IN	送信できなかったメッセージ番号（付録 1 参照）。

＜イベント発生＞

メッセージ送信関数をコールしてから送信完了タイムアウト時間が経過するまで、以下の条件を満たす場合に、イベントが発生します。

・ 1 対 1 メッセージを送信した場合に、送信先ノードから送達確認（ACK 応答）を受信できなかった

・ 1 対 n メッセージを送信した場合に、メッセージの送信ができなかった

タイムアウト時間は、HFA_SetTimeout 関数の Send 引数で設定します。なお、タイムアウト時間が 0 の場合、イベントは発生しません。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

メッセージ送信タイムアウトが発生した場合は、メッセージの送信を取り消します。

＜使用例＞

1）メッセージ送信タイムアウト発生時の処理を行う。

```
void CALLBACK SendTimeout (long NodeNo, long TransactionCode){
    printf("ノード%ld への TCD=%ld のメッセージ送信タイムアウト発生¥n", NodeNo, TransactionCode);
}
```

＜関連事項＞

・ HFA_SetTimeout 関数

・ 各メッセージ送信関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.25. メッセージ応答タイムアウト

＜関数 I/F＞

void ReplyTimeout (long NodeNo, long TransactionCode)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求先ノード番号（1～254）。
long	TransactionCode	IN	応答を受信できなかった要求メッセージ番号（付録 1 参照）。

＜イベント発生＞

1 対 1 の要求メッセージ送信関数をコールし、要求先ノードから送達確認（ACK 応答）を受信してから、応答受信以内に、応答メッセージを受信できなかったときにイベントが発生します。タイムアウト時間は、HFA_SetTimeout 関数の Reply 引数で設定します。なお、タイムアウト時間が 0 の場合、イベントは発生しません。また、1 対 n の要求メッセージを送信した場合も、イベントは発生しません。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜使用例＞

1）メッセージ応答タイムアウト発生時の処理を行う。

```
void CALLBACK ReplyTimeout (long NodeNo, long TransactionCode){
      printf("ノード%ld への TCD=%ld のメッセージ応答タイムアウト発生¥n", NodeNo, TransactionCode);
}
```

＜関連事項＞

- HFA_SetTimeout 関数
- 各メッセージ送信関数
- HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.26. メッセージ送信完了

＜関数 I/F＞

void SendComplete(long NodeNo, long TransactionCode, long Result)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	送信先ノード番号（1～255）。
long	TransactionCode	IN	送信メッセージ番号（付録 1 参照）。
long	Result	IN	送信結果。以下の値が格納されます。 ・ 0=送信完了（正常応答） ・ 2=送信先ノードバッファフル ・ 3=送信先ノード初期化未完 ・ 4=送信先ノード無応答 ・ 5=送信先ノードで通番バージョン番号エラーを検知 ・ 6=送信先ノードでフォーマット異常を検知

＜イベント発生＞

メッセージ送信関数をコールしてから以下の条件を満たす場合に、イベントが発生します。

・ 1 対 1 メッセージを送信した場合に、送信先ノードから送達確認（ACK 応答）を受信した。

・ 1 対 n メッセージを送信した場合に、メッセージの送信が成功した。（この場合、Result=0 でイベントが発生します。）

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

送信結果の内容は、送信先ノードから受信した ACK 内容に基づきます。なお、ACK 内容が異常の場合はメッセージの再送を繰り返し、メッセージの再送回数が 3 回を越えた場合にイベントが発生します。

＜使用例＞

1）メッセージ送信完了時の処理を行う。

```
void CALLBACK SendComplete (long NodeNo, long TransactionCode, long Result){
      printf("ノード%ld への TCD=%ld のメッセージ送信完了[送信結果=%ld]¥n",
            NodeNo, TransactionCode, Result);
}
```

＜関連事項＞

・ 各メッセージ送信関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.27. エラー

＜関数 I/F＞

void Error (long ErrorCode, long SpecCode)

＜引数＞

型	変 数	I/O	内 容
long	ErrorCode	IN	エラーコード（エラー種別）。 ・1=システムエラー ・2=ネットワークエラー ・3=致命的エラー ・4=プロトコルエラー ・5=セキュリティエラー
long	SpecCode	IN	エラー詳細コード。詳細は、以下をご参照ください。

＜イベント発生＞

「***DLL***」内部でエラーを検知した場合にイベントが発生します。内容を解析して、エラーの種別および詳細コードを通知します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

エラー詳細コードは、エラー種別に応じて値の意味が異なります。

・ 致命的エラー／プロトコルエラー／セキュリティエラーの詳細コードは、以下の表をご参照ください。

・ システムエラー／ネットワークエラーの詳細コードは、付録 2, 3 をご参照ください。

1）致命的エラーの詳細コード

詳細コード	意 味
10001	メモリ確保エラー（送信データバッファ）
10002	メモリ確保エラー（サイクリック受信バッファ）
10003	メモリ確保エラー（メッセージ受信バッファ）
10004	メモリ確保エラー（参加要求受信バッファ）
10005	メモリ確保エラー（サイクリック送信バッファ･0 ページ）
10006	メモリ確保エラー（サイクリック送信バッファ･1 ページ）
10007	メモリ確保エラー（メッセージ送信管理メモリ）
10008	メモリ確保エラー（ACK バッファ）
20001	スレッド生成エラー（サイクリック受信スレッド）
20002	スレッド生成エラー（メッセージ受信スレッド）
20003	スレッド生成エラー（参加要求フレーム受信スレッド）
20004	スレッド生成エラー（メッセージ受信メインスレッド）
20005	スレッド生成エラー（メッセージ送信メインスレッド）
20006	スレッド生成エラー（サイクリック送信スレッド）
20007	スレッド生成エラー（プロトコル管理スレッド）
20008	スレッド生成エラー（コマンドサーバメインスレッド）
20009	スレッド生成エラー（コマンドサーバ(UDP)スレッド）
20010	スレッド生成エラー（コマンドサーバ(TCP)スレッド）
20011	スレッド生成エラー（汎用通信データ送信元ログ測定スレッド）
20012	スレッド生成エラー（任意マスタスレッド）
30001	イベント生成エラー（サイクリック受信スレッド）
30002	イベント生成エラー（メッセージ受信スレッド）
30003	イベント生成エラー（参加要求フレーム受信スレッド）
30004	イベント生成エラー（タイマ通知）
30005	イベント生成エラー（メッセージ受信メインスレッド）
30006	イベント生成エラー（メッセージ送信メインスレッド）
30007	イベント生成エラー（サイクリック送信スレッド）
30008	イベント生成エラー（プロトコル管理スレッド）
30009	イベント生成エラー（最小許容フレーム通知）
30010	イベント生成エラー（コマンドサーバメインスレッド）
30011	イベント生成エラー（コマンドサーバ(UDP)スレッド）
30012	イベント生成エラー（コマンドサーバ(TCP)スレッド）
30013	イベント生成エラー（デバイスレベル管理）
40001	初期化処理エラー(ミューテックス取得)
40002	初期化処理エラー(DLL 読込み)

2）プロトコルエラーの詳細コード

詳細コード	意 味
10001	トークン重複検出（トークン剥奪。他ノードがトークンを保持している）
10002	トークン重複検出（トークン保持。自ノードがトークンを保持している）
10003	トークン重複検出（トークン保留。トークン保持中に、自ノードへのトークン受信）
10004	トークン取りこぼし（受信バッファフル）
20001	カレントフラグメントブロック異常
20002	カレントブロック長異常
20003	トータルフラグメントブロック異常
30001	受信通番バージョン異常
30002	受信通番異常
30003	受信フォーマット異常
30004	メッセージ取りこぼし（受信バッファフル）
30005	ACK 送信バッファフル
40001	ヘッダタイプ異常（サイクリックデータ）
40002	ヘッダタイプ異常（メッセージデータ）
40003	ヘッダタイプ異常（参加要求データ）
40004	トークンモード不一致検知
40005	ヘッダタイプ異常（コマンドサーバ）
50001	コモンメモリ（アドレス・サイズ）異常
60001	ACK 内容異常
60002	通番バージョン不一致
60003	通番不一致
70001	監視タイムアウト発生
70002	トークン保持タイムアウト発生

3）セキュリティエラーの詳細コード

詳細コード	意 味
10001	プロテクトキー通信エラー
10002	プロテクトキー無またはライセンス認証データ不一致
10003	プロテクトキー不整合
20001	試用版制限時間経過

＜関連事項＞

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.28. ネットワークパラメータライト要求メッセージ受信

＜関数 I/F＞

void RecvReqWriteNetParam (long NodeNo, long Common1Addr, long Common1Bytes,
*long Common2Addr, long Common2Words, char *NodeName, long SetMask)*

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求元ノード番号(1～254)。
long	Common1Addr	IN	コモンメモリ１アドレス（バイト単位）。
long	Common1Bytes	IN	コモンメモリ１バイト数（バイト単位）。
long	Common2Addr	IN	コモンメモリ２アドレス（ワード単位）。
long	Common2Words	IN	コモンメモリ２ワード数（ワード単位）。
char *	NodeName	IN	ノード名称。10 バイト以内で NULL 終端とします。
long	SetMask	IN	設定マスク。設定する項目に応じて以下の値を論理和演算で指定します。 ・0x00000001：アドレスの設定 ・0x00000002：ノード名称の設定

＜イベント発生＞

他ノードからネットワークパラメータライト要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、ネットワークパラメータライト応答メッセージを送信する(HFA_RepWriteNetParam 関数をコールする)必要があります。

ネットワークパラメータライト応答を送信後、コモンメモリの設定を変更するため離脱した場合、ネットワークパラメータライト応答を送信完了する前に離脱処理が実行されてしまいます。そのため送信完了を待って離脱する必要があります。

＜使用例＞

１）ネットワークパラメータライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteNetParam(long NodeNo, long Common1Addr, long Common1Bytes, long
        Common2Addr, long Common2Words, char *NodeName, long SetMask){
    long lRet;
    if (…) {                    // 通知される引数情報を判断
        // ネットワークパラメータライト応答(正常応答)
        lRet = HFA_RepWriteNetParam(NodeNo, 0, 0, 0);
        // 送信完了を待つ
        Sleep(100);
        // 離脱
        HFA_LinkOut(G_LinkID);
        // コモンメモリ設定
        If(SetMask & 0x1){
            HFA_SetCommon(Common1Addr, Common1Bytes, Common2Addr, Common2Words);
        }
        // ノード名称設定
        If(SetMask & 0x2){
            HFA_SetNodeName (NodeName);
        }
        // 再参加
        HFA_LinkInDefault(G_LinkID);
    }
    else{
        unsigned char ucError=1;    // エラーコード=1
        // ネットワークパラメータライト応答(異常応答)
        lRet = HFA_RepWriteNetParam (NodeNo, 1, 1, &ucError);
    }
}
```

＜関連事項＞

・ HFA_RepWriteNetParam 関数

・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.29. 運転/停止指令要求メッセージ受信

＜関数 I/F＞

void RecvReqControlEquipment (long NodeNo, long Command)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	要求元ノード番号(1～254)。
long	Command	IN	運転/停止指令 ・0＝停止指令 ・0≠運転指令

＜イベント発生＞

他ノードから運転/停止指令要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、運転/停止指令応答メッセージを送信する(HFA_RepControlEquipment 関数をコールする)必要があります。

＜使用例＞

１）運転/停止指令要求受信時の処理を行う。

```
void CALLBACK RecvReqControlEquipment(long NodeNo, long Command){
    long lRet;
    if (…) {                    // 通知される引数情報を判断
        // 運転/停止指令応答(正常応答)
        lRet = HFA_RepControlEquipment(NodeNo, 0, Command, 0, 0);
    }
    else{
        unsigned char ucError=1;    // エラーコード=1
        // 運転/停止指令応答(異常応答)
        lRet = HFA_RepControlEquipment(NodeNo, 1, Command, 1, &ucError);
    }
}
```

＜関連事項＞

・ HFA_RepControlEquipment 関数
・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.30. スレーブノード参加

＜関数 I/F＞

void LinkInSlave (long NodeNo, long Reason)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	FL-net ネットワークに参加したスレーブノードの番号（1～249）。
long	Reason	IN	参加結果。 ・0＝参加成功

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

・ スレーブノードが FL-net ネットワークに途中参加したとき

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、自ノード未参加の場合は、スレーブノードに対する参加イベントは通知しません。

＜詳細＞

自ノードが FL-net ネットワークに参加する前から､既に参加中のスレーブノードについては通知しません。HFA_GetSlaveNodeLinkStatus 関数でスレーブノードの一覧を確認してください。

本イベントと同時に LinkIn イベントも発生します。発生順序は、*LinkIn* イベント→*LinkInSlave* イベントとなります。

＜使用例＞

１）スレーブノード参加時の処理を行う。

```
void CALLBACK LinkInSlave (long NodeNo, long Reason ){
    printf("スレーブノード%ld が参加しました。", NodeNo);
}
```

＜関連事項＞

・ HFA_GetSlaveNodeLinkStatus 関数
・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数
・ *LinkIn* イベント

4.7.31. スレーブノード離脱

＜関数 I/F＞

void LinkOutSlave (long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	リンクから離脱したスレーブノードの番号（1～249）。

＜イベント発生＞

以下の条件を満たす場合に、イベントが発生します。

・ トークン欠落（トークン監視時間タイムアウト／トークン周回抜け）によって、スレーブノードの離脱を検知したとき

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、自ノード未参加の場合は、スレーブノードの離脱イベントは通知しません。

＜詳細＞

自ノードが FL-net ネットワークから離脱した後に離脱したスレーブノードについては通知しません。HFA_GetSlaveNodeLinkStatus 関数でスレーブノードの一覧を確認してください。

本イベントと同時に *LinkOut* イベントも発生します。発生順序は、*LinkOut* イベント→*LinkOutSlave* イベントとなります。

＜使用例＞

１）スレーブノード離脱時の処理を行う。

```
void CALLBACK LinkOutSlave (long NodeNo){
    printf("スレーブノード%ld が離脱しました。", NodeNo);
}
```

＜関連事項＞

・ HFA_GetSlaveNodeLinkStatus 関数
・ HFA_SetCallback 関数，HFA_SetCallbackV3 関数
・ *LinkOut* イベント

4.7.32. 入力データ変更

＜関数 I/F＞

void InputDataRefresh (long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	変更元スレーブノード番号（1～249）。

＜イベント発生＞

スレーブノードから受信したサイクリックデータをコモンメモリにコミット（反映）させる時点で、受信した入力データ値が受信前の値と異なっているときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

本関数コールバック時に HFA_ReadInputData 関数をコールすることで、「***APP***」は最新の入力データ値を読出すことができます。

本イベントと同時に *CommonRefresh* イベントも発生します。発生順序は、*CommonRefresh* イベント→ *InputDataRefresh* イベントとなります。

＜使用例＞

１）入力データの値を監視し、画面に表示する。

```
void CALLBACK InputDataRefresh (long NodeNo){
    HFA_ADDRESS xAddr;
    unsigned short usSize;
    unsigned char ucData[100];
    // 入力データ読出し
    if(HFA_ReadInputData(NodeNo, &xAddr, &usSize, ucData) == 0){
        printf("入力データのサイズ=%d¥n",  usSize);          // 値を画面に表示する。
    }
}
```

＜関連事項＞

- HFA_ReadInputData 関数
- HFA_SetCallback 関数，HFA_SetCallbackV3 関数
- *CommonRefresh* イベント

4.7.33. 入力ステータス変更

＜関数 I/F＞

void InputStatusRefresh (long NodeNo)

＜引数＞

型	変 数	I/O	内 容
long	NodeNo	IN	変更元スレーブノード番号（1～249）。

＜イベント発生＞

スレーブノードから受信したサイクリックデータをコモンメモリにコミット（反映）させる時点で、受信した入力ステータス値が受信前の値と異なっているときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。ただし、ネットワーク未参加（モニタモード含む）の場合は、イベント通知しません。

＜詳細＞

本関数コールバック時に HFA_GetInputStatus 関数をコールすることで、「***APP***」は最新の入力ステータス値を読出すことができます。

本イベントと同時に *CommonRefresh* イベントも発生します。発生順序は、*CommonRefresh* イベント→ *InputStatusRefresh* イベントとなります。

＜使用例＞

１）入力ステータスの値を監視し、画面に表示する。

```
void CALLBACK InputStatusRefresh (long NodeNo){
    HFA_SLAVE_INPUT_STATUS xStatus;
    // 入力ステータス読出し
    if(HFA_GetInputStatus(NodeNo, &xStatus) == 0){
        printf(“%d¥n”, xStatus.StatusNo);          // 値を画面に表示する。
    }
}
```

＜関連事項＞

- HFA_GetInputStatus 関数
- HFA_SetCallback 関数，HFA_SetCallbackV3 関数
- *CommonRefresh* イベント

4.7.34. トークン保持時間測定状態変化

＜関数 I/F＞

void ChangeTokenTimeMeasureStatus(long Status, long Sender)

＜引数＞

型	変 数	I/O	内 容
long	Status	IN	トークン保持時間測定状態 ・0=測定中→停止中 ・1=停止中→測定中 ・2=測定中→測定中
long	Sender	IN	イベント発生元 ・0=自分(***APP*** 自身) ・1=自分以外(設定ツールなど)

＜イベント発生＞

「***APP***」または設定ツールからトークン保持時間測定の開始・停止要求を受けて、以下の状態遷移が起きたときにイベントが発生します。

・測定中→停止中

・停止中→測定中

・測定中→測定中

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

HFA_StartTokenTimeMeasure 関数をコールすると測定を開始し、HFA_StopTokenTimeMeasure 関数をコールすると測定を停止します。また、設定ツールからトークン保持時間測定開始フレームを受信すると測定を開始し、トークン保持時間測定終了フレームを受信すると測定を停止します。

トークン保持時間の測定結果はログ領域に保存されます。HFA_GetMyNodeLogV3 関数をコールして測定結果を取得できます。

＜使用例＞

１） トークン保持時間測定状態変化時の処理を行う。

```
void CALLBACK ChangeTokenTimeMeasureStatus(long Status, long Sender){
      HFA_LOG_V3 xLog;
      if(Sender == 0 ) return;   //自分がイベント発生元の場合は何もしない。

      switch(Stataus){
      case 0:   // 測定中→停止中の処理
            HFA_GetMyNodeLogV3(&xLog);        // 自ノードログ情報取得
            printf("トークン破棄回数=%d", xLog.Token.Destroy);
            break;
      case 1:   // 停止中→測定中の処理
            break;
      case 2:   // 測定中→測定中の処理
            break;
      }
}
```

＜関連事項＞

- HFA_StartTokenTimeMeasure 関数，HFA_StopTokenTimeMeasure 関数
- HFA_GetTokenTimeMeasureStatus 関数
- HFA_GetMyNodeLogV3 関数
- HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.35. 汎用通信データ送信元ログ測定状態変化

＜関数 I/F＞

void ChangeDataLogMeasureStatus(long Status, long Sender)

＜引数＞

型	変 数	I/O	内 容
long	Status	IN	汎用通信データ送信元ログ測定状態 ・0=測定中→停止中 ・1=停止中→測定中 ・2=測定中→測定中
long	Sender	IN	イベント発生元 ・0=自分(***APP*** 自身) ・1=自分以外(設定ツールなど)

＜イベント発生＞

「***APP***」または設定ツールから汎用通信データ送信元ログ測定の開始・停止要求を受け、以下、状態遷移が起きたときにイベントが発生します。

・測定中→停止中

・停止中→測定中

・測定中→測定中

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

HFA_StartDataLogMeasure 関数をコールすると測定を開始し、HFA_StopDataLogMeasure 関数をコールすると測定を停止します。また、設定ツールから汎用通信データ送信元ログ測定開始フレームを受信すると測定を開始し、汎用通信データ送信元ログ測定終了フレームを受信すると測定を停止します。

汎用通信データ送信元ログの測定結果はログ領域に保存されます。HFA_GetMyNodeLogV3 関数をコールして測定結果を取得できます。

＜使用例＞

１）汎用通信データ送信元ログ測定状態変化時の処理を行う。

```
void CALLBACK ChangeDataLogMeasureStatus(long Status, long Sender){
    HFA_LOG_V3 xLog;
    if(Sender == 0 ) return;   //自分がイベント発生元の場合は何もしない。

    switch(Stataus){
    case 0:   // 測定中→停止中の処理
        HFA_GetMyNodeLogV3(&xLog);        // 自ノードログ情報取得
        printf("IP1 アドレス=%x¥n", xLog.IP.Log[0].Address);
        break;
    case 1:   // 停止中→測定中の処理
        break;
    case 2:   // 測定中→測定中の処理
        break;
    }
}
```

＜関連事項＞

- HFA_StartDataLogMeasure 関数，HFA_StopDataLogMeasure 関数
- HFA_GetDataLogMeasureStatus 関数
- HFA_GetMyNodeLogV3 関数
- HFA_SetCallback 関数，HFA_SetCallbackV3 関数

4.7.36. 設定ツールからのバイトブロックリード要求

＜関数 I/F＞

*void RecvReqReadByteBlockFromSettingTool(unsigned long SettingToolAddr, unsigned long Addr, long Bytes, HFA_REPLY_DATA *Data)*

＜引数＞

型	変 数	I/O	内 容
unsigned long	SettingToolAddr	IN	設定ツールの IP アドレス（unsigned long 型）。
unsigned long	Addr	IN	バイトブロックの読出し開始アドレス（バイト単位）。
long	Bytes	IN	バイトブロックの読出しデータサイズ（バイト単位）。
HFA_REPLY_DATA *	Data	OUT	応答データ情報[23]のポインタ。構造体メンバに以下の値を設定してください。 メンバ：設定値 Result：応答結果（0=正常，1=異常，2=非実装） Data：応答データ。応答結果に応じて、以下の値を設定してください。 ・正常応答：バイトブロックの読出しデータ ・異常応答：エラー内容 ・非実装応答：設定不要 ErrBytes：異常応答時のエラー内容のバイト数（0～1024）

＜イベント発生＞

設定ツールからバイトブロックリード要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、Data 引数に結果を格納する必要があります。

[23] 応答データ情報の構造体は、4.8.24 項を参照してください。

＜使用例＞

1）バイトブロックリード要求受信時の処理を行う。

```
void CALLBACK RecvReqReadByteBlockFromSettingTool (unsigned long SettingToolAddr, unsigned long
        Addr, long Bytes, HFA_REPLY_DATA *Data){
    // 設定ツールの IP アドレス
    in_addr oIpAddress;
    oIpAddress.S_un.S_addr = SettingToolAddr;
    printf("設定ツールの IP アドレス=%s¥n", inet_ntoa(oIpAddress));

    if (…) {          // 通知される引数情報を判断
        // 正常応答
        Data->Result = 0;              // 正常
        memcpy(Data->Data, &G_ByteBlock[Addr], Bytes);          // リードデータコピー
    }
    else{
        // 異常応答
        Data->Result = 1;              // 異常
        Data->ErrBytes = 1;            // エラー情報のバイト数
        Data->Data[0] = 1;             // エラーコード
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.7.37. 設定ツールからのワードブロックリード要求

＜関数 I/F＞

*void RecvReqReadWordBlockFromSettingTool (unsigned long SettingToolAddr, unsigned long Addr, long Words, HFA_REPLY_DATA *Data)*

＜引数＞

<table>
<tr><th>型</th><th>変 数</th><th>I/O</th><th>内 容</th></tr>
<tr><td>unsigned long</td><td>SettingToolAddr</td><td>IN</td><td>設定ツールの IP アドレス（unsigned long 型）。</td></tr>
<tr><td>unsigned long</td><td>Addr</td><td>IN</td><td>ワードブロックの読出し開始アドレス（ワード単位）。</td></tr>
<tr><td>long</td><td>Words</td><td>IN</td><td>ワードブロックの読出しデータサイズ（ワード単位）。</td></tr>
<tr><td>HFA_REPLY_DATA *</td><td>Data</td><td>OUT</td><td>応答データ情報[24]のポインタ。構造体メンバに以下の値を設定してください。
<table>
<tr><th>メンバ</th><th>設定値</th></tr>
<tr><td>Result</td><td>応答結果（0=正常，1=異常，2=非実装）</td></tr>
<tr><td>Data</td><td>応答データ。応答結果に応じて、以下の値を設定してください。
・正常応答：ワードブロックの読出しデータ
・異常応答：エラー内容
・非実装応答：設定不要</td></tr>
<tr><td>ErrBytes</td><td>異常応答時のエラー内容のバイト数（0～1024）</td></tr>
</table>
</td></tr>
</table>

＜イベント発生＞

設定ツールからワードブロックリード要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、Data 引数に結果を格納する必要があります。

[24] 応答データ情報の構造体は、4.8.24 項を参照してください。

＜使用例＞

1）ワードブロックリード要求受信時の処理を行う。

```
void CALLBACK RecvReqReadWordBlockFromSettingTool (unsigned long SettingToolAddr, unsigned long
        Addr, long Words, HFA_REPLY_DATA *Data){
    if (…) {          // 通知される引数情報を判断
        // 正常応答
        Data->Result = 0;              // 正常
        memcpy(Data->Data, &G_WordBlock[Addr], Words*2);   // リードデータコピー
    }
    else{
        // 異常応答
        Data->Result = 1;              // 異常
        Data->ErrBytes = 1;            // エラー情報のバイト数
        Data->Data[0] = 1;             // エラーコード
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.7.38. 設定ツールからのバイトブロックライト要求

＜関数 I/F＞

*void RecvReqWriteByteBlockFromSettingTool (unsigned long SettingToolAddr, unsigned long Addr, long Bytes, unsigned char *InData, HFA_REPLY_DATA *OutData)*

＜引数＞

型	変 数	I/O	内 容
unsigned long	SettingToolAddr	IN	設定ツールの IP アドレス（unsigned long 型）。
unsigned long	Addr	IN	バイトブロックの書込み開始アドレス（バイト単位）。
long	Bytes	IN	バイトブロックの書込みデータサイズ（バイト単位）。
unsigned char *	InData	IN	書込みデータの先頭ポインタ
HFA_REPLY_DATA *	OutData	OUT	応答データ情報[25]のポインタ。構造体メンバに以下の値を設定してください。 メンバ：Result — 応答結果（0=正常，1=異常，2=非実装） メンバ：Data — 応答データ。応答結果に応じて、以下の値を設定してください。 ・正常応答：設定不要 ・異常応答：エラー内容 ・非実装応答：設定不要 メンバ：ErrBytes — 異常応答時のエラー内容のバイト数（0～1024）

＜イベント発生＞

設定ツールからバイトブロックライト要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、OutData 引数に結果を格納する必要があります。

[25] 応答データ情報の構造体は、4.8.24 項を参照してください。

＜使用例＞

1）バイトブロックライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteByteBlockFromSettingTool(unsigned long SettingToolAddr, unsigned long Addr,
        long Bytes, unsigned char *InData,HFA_REPLY_DATA *OutData){
    if (…) {                    // 通知される引数情報を判断
        // 正常応答
        OutData->Result = 0;        // 正常
        memcpy(&G_ByteBlock[Addr], InData, Bytes);   // ライトデータコピー
    }
    else{
        // 異常応答
        OutData->Result = 1;        // 異常
        OutData->ErrBytes = 1;      // エラー情報のバイト数
        OutData->Data[0] = 1;       // エラーコード
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.7.39. 設定ツールからのワードブロックライト要求

＜関数 I/F＞

*void RecvReqWriteWordBlockFromSettingTool(unsigned long SettingToolAddr, unsigned long Addr, long Words, unsigned char *InData, HFA_REPLY_DATA *OutData)*

＜引数＞

型	変 数	I/O	内 容
unsigned long	SettingToolAddr	IN	設定ツールの IP アドレス（unsigned long 型）。
unsigned long	Addr	IN	ワードブロックの書込み開始アドレス（ワード単位）。
long	Words	IN	ワードブロックの書込みデータサイズ（ワード単位）。
unsigned char *	InData	IN	書込みデータの先頭ポインタ **注）「*APP*」では、ワード単位（=2 バイト単位）に型変換してから、アクセスしてください。**
HFA_REPLY_DATA *	OutData	OUT	応答データ情報[26]のポインタ。構造体メンバに以下の値を設定してください。 メンバ：設定値 Result：応答結果（0=正常，1=異常，2=非実装） Data：応答データ。応答結果に応じて、以下の値を設定してください。 ・正常応答：設定不要 ・異常応答：エラー内容 ・非実装応答：設定不要 ErrBytes：異常応答時のエラー内容のバイト数（0～1024）

＜イベント発生＞

設定ツールからワードブロックライト要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、OutData 引数に結果を格納する必要があります。

[26] 応答データ情報の構造体は、4.8.24 項を参照してください。

＜使用例＞

1）ワードブロックライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteWordBlockFromSettingTool(unsigned long SettingToolAddr, unsigned long
        Addr, long Words, unsigned char *InData,HFA_REPLY_DATA *OutData){
    if (…) {                    // 通知される引数情報を判断
        // 正常応答
        OutData->Result = 0;        // 正常
        memcpy(&G_WordBlock[Addr], InData, Words*2);   // ライトデータコピー
    }
    else{
        // 異常応答
        OutData->Result = 1;        // 異常
        OutData->ErrBytes = 1;      // エラー情報のバイト数
        OutData->Data[0] = 1;       // エラーコード
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.7.40. 設定ツールからのネットワークパラメータライト要求

＜関数 I/F＞

void RecvReqWriteNetParamFromSettingTool (unsigned long SettingToolArea,
*HFA_COMMON_MEMORY *CommonMemory,*
*char *NodeName, long SetMask,*
*HFA_REPLY_DATA *Data, long *Join)*

＜引数＞

<table>
<tr><th>型</th><th>変 数</th><th>I/O</th><th>内 容</th></tr>
<tr><td>unsigned long</td><td>SettingToolAddr</td><td>IN</td><td>設定ツールの IP アドレス（unsigned long 型）。</td></tr>
<tr><td>HFA_COMMON_MEMORY *</td><td>CommonMemory</td><td>IN</td><td>コモンメモリ領域[27]のポインタ</td></tr>
<tr><td>char *</td><td>NodeName</td><td>IN</td><td>ノード名称。NULL 終端とします。</td></tr>
<tr><td>long</td><td>SetMask</td><td>IN</td><td>設定マスク。設定する項目に応じて以下の値を論理和演算で指定します。
・0x00000001：アドレスの設定
・0x00000002：ノード名称の設定</td></tr>
<tr><td>HFA_REPLY_DATA *</td><td>Data</td><td>OUT</td><td>応答データ情報[28]のポインタ。構造体メンバに以下の値を設定してください。
<table>
<tr><th>メンバ</th><th>設定値</th></tr>
<tr><td>Result</td><td>応答結果（0=正常，1=異常，2=非実装）</td></tr>
<tr><td>Data</td><td>応答データ。応答結果に応じて、以下の値を設定してください。
・正常応答：設定不要
・異常応答：エラー内容
・非実装応答：設定不要</td></tr>
<tr><td>ErrBytes</td><td>異常応答時のエラー内容のバイト数（0～1024）</td></tr>
</table></td></tr>
<tr><td>long *</td><td>Join</td><td>OUT</td><td>再参加（正常応答時有効）
・0＝再参加無し
・0≠再参加あり</td></tr>
</table>

＜イベント発生＞

設定ツールからネットワークパラメータライト要求メッセージを受信したときにイベントが発生します。

---

[27] コモンメモリ領域情報の構造体は、4.8.22 項を参照してください。

[28] 応答データ情報の構造体は、4.8.24 項を参照してください。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、Data 引数および Join 引数に結果を格納する必要があります。

正常応答（Data->Result=0）した場合で、ネットワークパラメータの設定値が現在値と異なる場合は、ネットワークパラメータを更新して、リンクから離脱します。ネットワークパラメータ更新後に、リンクに再参加する場合は、Join 引数に再参加ありを設定してください。

なお、モニタモード中はコモンメモリ領域の設定は出来ません。

＜使用例＞

1）ネットワークパラメータライト要求受信時の処理を行う。

```
void CALLBACK RecvReqWriteNetParamFromSettingTool(unsigned long SettingToolAddr,
        HFA_COMMN_MEMORY *CommonMemory, char *NodeName, long SetMask,
        HFA_REPLY_DATA *Data, long *Join){
    if (…) {          // 通知される引数情報を判断
        // 設定可
        Data->Result = 0;              // 正常
        *Join= 1;                      // 再参加あり
    }
    else{
        // 設定不可
        Data->Result = 1;              // 異常
        Data->ErrBytes = 1;            // エラー情報のバイト数
        Data->Data[0] = 1;             // エラーコード
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.7.41. 設定ツールからの運転/停止指令要求

＜関数 I/F＞

*void RecvReqControlEquipmentFromSettingTool (unsigned long SettingToolArea, long Command, HFA_REPLY_DATA *Data)*

＜引数＞

型	変 数	I/O	内 容
unsigned long	SettingToolAddr	IN	設定ツールの IP アドレス（unsigned long 型）。
long	Command	IN	運転/停止指令 ・0＝停止指令 ・0≠運転指令
HFA_REPLY_DATA *	Data	OUT	応答データ情報[29]のポインタ。構造体メンバに以下の値を設定してください。

メンバ	設定値
Result	応答結果（0=正常，1=異常，2=非実装）
Data	応答データ。応答結果に応じて、以下の値を設定してください。 ・正常応答：設定不要 ・異常応答：エラー内容 ・非実装応答：設定不要
ErrBytes	異常応答時のエラー内容のバイト数（0～1024）

＜イベント発生＞

設定ツールから運転/停止指令要求メッセージを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、Data 引数に結果を格納する必要があります。
正常応答（Data->Result=0）した場合は、Command 引数の設定に応じて、運転/停止状態を更新します。

[29] 応答データ情報の構造体は、4.8.24 項を参照してください。

＜使用例＞

１）運転/停止指令要求受信時の処理を行う。

```
void CALLBACK RecvReqControlEquipmentFromSettingTool(long SettingToolAddr, long Command,
        HFA_REPLY_DATA *Data, long *Result){
    if (…) {        // 通知される引数情報を判断
        // 設定可
        Data->Result = 0;           // 正常
    }
    else{
        // 設定不可
        Data->Result = 1;           // 異常
        Data->ErrBytes = 1;         // エラー情報のバイト数
        Data->Data[0] = 1;          // エラーコード
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.7.42. 設定ツールからの IO 割付設定要求

＜関数 I/F＞

*void RecvReqSetIOFromSettingTool (unsigned long SettingToolArea, HFA_SLAVES_INFO *IoInfo, long *Result)*

＜引数＞

型	変 数	I/O	内 容
unsigned long	SettingToolAddr	IN	設定ツールの IP アドレス（unsigned long 型）。
HFA_SLAVES_INFO *	IoInfo	IN	IO 割付情報[30]のポインタ
long *	Result	OUT	応答結果 ・0=正常 ・1=異常 ・2=非実装 ・その他=非実装

＜イベント発生＞

設定ツールから IO 割付設定要求フレームを全て正常に受信完了したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、Result 引数に応答結果を格納する必要があります。
正常応答（Result=0）した場合は、IO 割付設定を更新します。

＜使用例＞

１）IO 割付設定要求受信時の処理を行う。

```
void CALLBACK RecvReqSetIOFromSettingTool(unsigned long SettingToolAddr, HFA_SLAVES_INFO *IoInfo,
        long *Result){
    if (…) {        // 通知される引数情報を判断
        // 設定可
        *Result = 0;        // 正常
    }
    else{
        // 設定不可
        *Result = 1;        // 異常
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

[30] IO 割付情報の構造体は、4.8.17 項を参照してください。

4.7.43. 設定ツールからのコンフィギュレーション用パラメータ設定要求

＜関数 I/F＞

void RecvReqSetConfigParamFromSettingTool (unsigned long SettingToolArea,
*HFA_CONFIG_PARAM *Config, long *Result, long *Join)*

＜引数＞

型	変 数	I/O	内 容
unsigned long	SettingToolAddr	IN	設定ツールの IP アドレス（unsigned long 型）。
HFA_CONFIG_PARAM *	Config	IN	コンフィギュレーション用パラメータ情報[31]のポインタ
long	SetMask	IN	設定マスク。設定する項目に応じて以下の値を論理和演算で指定します。[bit 値：0=更新なし, 1=更新あり] ・0x00000001：コモンメモリ更新フラグ ・0x00000002：ノード名更新フラグ ・0x00000004：トークン監視時間更新フラグ ・0x00000008：最小許容フレーム間隔更新フラグ
long *	Result	OUT	応答結果 ・0=正常 ・1=異常 ・2=非実装 ・その他=非実装
long *	Join	OUT	再参加有無（Result=0 の場合に有効） ・0=再参加なし ・0≠再参加あり

＜イベント発生＞

設定ツールからコンフィギュレーション用パラメータ設定要求フレームを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、Result 引数および Join 引数に結果を格納する必要があります。

正常応答（Result=0）した場合は、自ノードを離脱して、コンフィギュレーション用パラメータを更新します。設定更新後、自ノードを再参加する場合、Join 引数に再参加ありを設定します。

[31] コンフィギュレーション用パラメータ情報の構造体は、4.8.23 項を参照してください。

＜使用例＞

１）コンフィギュレーション用パラメータ設定要求受信時の処理を行う。

```
void CALLBACK RecvReqSetConfigParamFromSettingTool(unsigned long SettingToolAddr,
        HFA_CONFIG_PARAM *Config, long SetMask, long *Result, long *Join){
    if (…) {                    // 通知される引数情報を判断
        // 設定可
        *Result = 0;            // 正常
        *Join = 1;              // 再参加あり
    }
    else if(...){
        // 設定可
        *Result = 0;            // 正常
        *Join = 0;              // 再参加なし
    }
    else{
        // 設定不可
        *Result = 1;            // 異常
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.7.44. 設定ツールからのノードリセット要求

＜関数 I/F＞

*void RecvReqResetNodeFromSettingTool (unsigned long SettingToolArea, long *Result, long *Join)*

＜引数＞

型	変 数	I/O	内 容
unsigned long	SettingToolAddr	IN	設定ツールの IP アドレス（unsigned long 型）。
long *	Result	OUT	応答結果 ・0=正常 ・1=異常 ・2=非実装 ・その他=非実装
long *	Join	OUT	再参加有無（Result=0 の場合に有効） ・0＝再参加なし ・0≠再参加あり

＜イベント発生＞

設定ツールからノードリセット要求フレームを受信したときにイベントが発生します。

＜イベント通知先＞

コールバック関数が登録されている「***APP***」。

＜詳細＞

「***APP***」側では、本イベントに対する応答結果を判断し、Result 引数および Join 引数に結果を格納する必要があります。

正常応答（Result=0）した場合は、自ノードを離脱します。自ノードを再参加する場合は、Join 引数に再参加ありを設定します。

＜使用例＞

１）ノードリセット要求受信時の処理を行う。

```
void CALLBACK RecvReqResetNodeFromSettingTool(unsigned long SettingToolAddr,
        long *Result, long *Join){
    if (…) {                    // 通知される引数情報を判断
        // リセット可
        *Result = 0;            // 正常
        *Join = 1;              // 再参加あり
    }
    else if(...){
        // リセット可
        *Result = 0;            // 正常
        *Join = 0;              // 再参加なし
    }
    else{
        // リセット不可
        *Result = 1;            // 異常
    }
}
```

＜関連事項＞

・ HFA_SetCallbackV3 関数

4.8. I/F 構造体

「***DLL***」の I/F 関数をコール時に使用する構造体の仕様を以下に示します。構造体の型定義は、ヘッダファイル（”FL_KE.h”）に含まれています。

I/F 構造体一覧

No	I/F 構造体名	説明
1	NODE	ノード情報管理パラメータ
2	NETWORK	ネットワーク管理情報パラメータ
3	CALLBACK	コールバック登録テーブル（FL-net Ver.2 専用）
4	LOG	ログ情報テーブル（FL-net Ver.2 専用）
5	LOG_Protocol	ログ情報－プロトコル情報部
6	LOG_Frame	ログ情報－フレーム情報部
7	LOG_Cyclic	ログ情報－サイクリック情報部
8	LOG_Message	ログ情報－メッセージ情報部
9	LOG_ACK	ログ情報－ACK 情報部
10	LOG_Token	ログ情報－トークン情報部
11	LOG_Link	ログ情報－リンク情報部
12	LOG_Node	ログ情報－参加ノード情報部
13	LOG_User	ログ情報－製造業者定義可能情報部
14	HFA_CALLBACKFUNC_V3	コールバック関数テーブル（FL-net Ver.3 対応版）
15	HFA_ADDRESS	アドレス情報
16	HFA_SLAVE_INFO	スレーブ登録情報
17	HFA_SLAVES_INFO	全スレーブ登録情報
18	HFA_SLAVE_STATUS	スレーブ状態
19	HFA_SLAVE_INPUT_STATUS	入力ステータス
20	HFA_SLAVE_OUTPUT_STATUS	出力ステータス
21	HFA_LOG_TOKENTIME	トークン保持時間測定項目
22	HFA_COMMON_MEMORY	コモンメモリ情報
23	HFA_CONFIG_PARAM	コンフィギュレーション用パラメータ
24	HFA_REPLY_DATA	応答データ
25	HFA_LOG_V3	ログ情報テーブル（FL-net Ver.3 対応版）
26	HFA_LOG_Cyclic_V3	ログ情報－サイクリック情報部（FL-net Ver.3 対応版）
27	HFA_LOG_Message_V3	ログ情報－メッセージ情報部（FL-net Ver.3 対応版）
28	HFA_LOG_Token_V3	ログ情報－トークン情報部（FL-net Ver.3 対応版）
29	HFA_IP	IP 情報
30	HFA_LOG_IP_V3	汎用通信データ送信元ログ

4.8.1. NODE 構造体

項目	内容	
要約	ノード情報管理パラメータ	
定義	typedef struct{ char VendorName[10]; char MakerType[10]; char NodeName[10]; long Common1Addr; long Common1Size; long Common2Addr; long Common2Size; long TokenTimeout; long MaxRefreshCycle; long RefreshCycle; long MinFrameInterval; long UpperStatus; long LinkStatus; long ProtocolVersion; long MyNodeStatus; } NODE;	
メンバ	VendorName	ベンダ名称
	MakerType	製造業者形式
	NodeName	ノード名称
	Common1Addr	コモンメモリ領域 1 先頭アドレス（バイト単位）
	Common1Size	コモンメモリ領域 1 データサイズ（バイト単位）
	Common2Addr	コモンメモリ領域 2 先頭アドレス（ワード単位）
	Common2Size	コモンメモリ領域 2 データサイズ（ワード単位）
	TokenTimeout	トークン監視時間
	MaxRefreshCycle	リフレッシュサイクル許容時間
	RefreshCycle	リフレッシュサイクル実測値
	MinFrameInterval	最小許容フレーム間隔
	UpperStatus	上位層の状態
	LinkStatus	FA リンクの状態
	ProtocolVersion	プロトコルタイプ
	MyNodeStatus	自ノードの状態
解説	HFA_GetNodeStatus 関数でノード管理情報パラメータを取得する際に使用します。	

4.8.2. NETWORK 構造体

項目	内容	
要約	ネットワーク管理情報パラメータ	
定義	typedef struct{ long TokenOwner; long MinFrameInterval; long MaxRefreshCycle; long RefreshCycle; long RefreshCycleHigh; long RefreshCycleLow; } NETWORK;	
メンバ	TokenOwner	トークン保持ノード番号
	MinFrameInterval	最小許容フレーム間隔
	MaxRefreshCycle	リフレッシュサイクル許容時間
	RefreshCycle	リフレッシュサイクル測定時間（現在値）
	RefreshCycleHigh	リフレッシュサイクル測定時間（最大値）
	RefreshCycleLow	リフレッシュサイクル測定時間（最小値）
解説	HFA_GetNetworkStatus 関数でネットワーク管理情報パラメータを取得する際に使用します。	

4.8.3. CALLBACK 構造体

項目	内容	
要約	コールバック登録テーブル（FL-net Ver.2 専用）	
定義	typedef struct{ CBF_LinkIn LinkIn; CBF_LinkOut LinkOut; CBF_CommonRefresh CommonRefresh; CBF_LogClear LogClear; CBF_RecvReqReadByte RecvReqReadByteBlock; CBF_RecvReqWriteByte RecvReqWriteByteBlock; CBF_RecvReqReadWord RecvReqReadWordBlock; CBF_RecvReqWriteWord RecvReqWriteWordBlock; CBF_RecvRepReadByte RecvRepReadByteBlock; CBF_RecvRepWriteByte RecvRepWriteByteBlock; CBF_RecvRepReadWord RecvRepReadWordBlock; CBF_RecvRepWriteWord RecvRepWriteWordBlock; CBF_RecvRepReadNet RecvRepReadNetParam; CBF_RecvRepWriteNet RecvRepWriteNetParam; CBF_RecvRepControl RecvRepControlEquipment; CBF_RecvRepReadProfile RecvRepReadProfile; CBF_RecvRepReadLog RecvRepReadLog; CBF_RecvRepClearLog RecvRepClearLog; CBF_RecvRepEchoMessage RecvRepEchoMessage; CBF_RecvTransparency RecvTransparency; CBF_LinkInTimeout LinkInTimeout; CBF_SendTimeout SendTimeout; CBF_ReplyTimeout ReplyTimeout; CBF_SendComplete SendComplete; CBF_Error Error; CBF_RecvReqVendor RecvReqVendorMessage; CBF_RecvRepVendor RecvRepVendorMessage; } CALLBACK;	
メンバ	LinkIn	ノードが参加したとき
	LinkOut	ノードが離脱したとき
	CommonRefresh	コモンメモリが変化したとき
	LogClear	自ノードのログデータがクリアされたとき
	RecvReqReadByteBlock	バイトブロックリード要求を受信

	RecvReqWriteByteBlock	バイトブロックライト要求を受信
	RecvReqReadWordBlock	ワードブロックリード要求を受信
	RecvReqWriteWordBlock	ワードブロックライト要求を受信
	RecvRepReadByteBlock	バイトブロックリード応答を受信
	RecvRepWriteByteBlock	バイトブロックライト応答を受信
	RecvRepReadWordBlock	ワードブロックリード応答を受信
	RecvRepWriteWordBlock	ワードブロックライト応答を受信
	RecvRepReadNetParam	ネットワークパラメータのリード要求を受信
	RecvRepWriteNetParam	ネットワークパラメータのライト要求を受信
	RecvRepControlEquipment	運転／停止指令応答を受信
	RecvRepReadProfile	プロファイルリード応答を受信
	RecvRepReadLog	ログデータリード応答を受信
	RecvRepClearLog	ログデータクリア応答を受信
	RecvRepEchoMessage	メッセージ折り返し応答を受信
	RecvTransparency	透過形メッセージを受信
	LinkInTimeout	リンク参加タイムアウトが発生したとき
	SendTimeout	送信タイムアウトが発生したとき
	ReplyTimeout	応答受信タイムアウトが発生したとき
	SendComplete	メッセージを送信したとき
	Error	エラーが発生したとき
	RecvReqVendorMessage	ベンダ固有要求メッセージを受信
	RecvRepVendorMessage	ベンダ固有応答メッセージを受信
解説	HFA_SetCallback 関数でコールバック関数を登録する際に使用します。 ※各メンバの型については、対応するコールバック関数のプロトタイプ（本書 4.7 節）をご参照ください。	

4.8.4. LOG 構造体

項目	内容	
要約	LOG 情報テーブル（FL-net Ver.2 専用）	
定義	typedef struct{ LOG_Protocol Protocol; LOG_Frame Frame; LOG_Cyclic Cyclic; LOG_Message Message; LOG_ACK Ack; LOG_Token Token; LOG_Link Link; LOG_Node Node; LOG_User User; } LOG;	
メンバ	Protocol	プロトコル情報
	Frame	フレーム情報
	Cyclic	サイクリック情報
	Message	メッセージ情報
	Ack	ACK 情報
	Token	トークン情報
	Link	リンク情報
	Node	参加ノード情報
	User	製造業者定義可能情報
解説	HFA_GetMyNodeLog 関数で自ノードのログ情報を読出す場合に使用します。	

4.8.5. LOG_Protocol 構造体

項目	内容	
要約	ログ情報－プロトコル情報部	
定義	typedef struct{ long SendData; long SendSockErr; long SendNetErr; long RecvData; long RecvSockErr; long RecvNetErr; } LOG_Protocol	
メンバ	SendData	ソケット送信通算回数
	SendSockErr	ソケット送信エラー回数
	SendNetErr	Ether 送信エラー回数（未使用）
	RecvData	ソケット受信通算回数
	RecvSockErr	ソケット受信エラー回数
	RecvNetErr	Ether 受信エラー回数（未使用）
解説	LOG 構造体のメンバです。	

4.8.6. LOG_Frame 構造体

項目	内容	
要約	ログ情報－フレーム情報部	
定義	typedef struct{ long SendToken; long SendCyclic; long SendPeerToPeer; long SendBroadcast; long RecvToken; long RecvCyclic; long RecvPeerToPeer; long RecvBroadcast; } LOG_Frame;	
メンバ	SendToken	トークン送信回数
	SendCyclic	サイクリックフレーム送信回数
	SendPeerToPeer	1 対 1 メッセージ送信回数
	SendBroadcast	1 対 n メッセージ送信回数
	RecvToken	トークン受信回数
	RecvCyclic	フレーム受信回数
	RecvPeerToPeer	1 対 1 メッセージ受信回数
	RecvBroadcast	1 対 n メッセージ受信回数
解説	LOG 構造体のメンバです。	

4.8.7. LOG_Cyclic 構造体

項目	内容	
要約	ログ情報－サイクリック情報部	
定義	typedef struct{ long RecvCyclicErr; long CyclicAddrErr; long CyclicCbnErr; long CyclicTbnErr; long CyclicBsizeErr; } LOG_Cyclic;	
メンバ	RecvCyclicErr	送受信エラー回数
	CyclicAddrErr	アドレスサイズエラー回数
	CyclicCbnErr	CBN エラー回数
	CyclicTbnErr	TBN エラー回数
	CyclicBsizeErr	Bsize エラー回数
解説	LOG 構造体のメンバです。	

4.8.8. LOG_Message 構造体

項目	内容	
要約	ログ情報－メッセージ情報部	
定義	typedef struct{ long Resend; long RetryOut; long RecvErr; long SeqVerErr; long SeqVerify; } LOG_Message;	
メンバ	Resend	再送回数
	RetryOut	再送オーバー回数
	RecvErr	受信エラー回数
	SeqVerErr	通番バージョンエラー回数
	SeqVerify	通番再送認識回数
解説	LOG 構造体のメンバです。	

4.8.9. LOG_ACK 構造体

項目	内容	
要約	ログ情報－ACK 情報部	
定義	typedef struct{ long AckErr; long VersionErr; long SeqNoErr; long NodeNoErr; long TransactionCodeErr; } LOG_ACK;	
メンバ	AckErr	ACK エラー回数
	VersionErr	通番バージョンエラー回数
	SeqNoErr	通番番号エラー回数
	NodeNoErr	ノード番号エラー回数（未使用）
	TransactionCodeErr	メッセージ番号（TCD）エラー回数
解説	LOG 構造体のメンバです。	

4.8.10. LOG_Token 構造体

項目	内容	
要約	ログ情報－トークン情報部	
定義	typedef struct{ long Conflict; long Destroy; long Retry; long KeepTimeout; long WatchTimeout; } LOG_Token;	
メンバ	Conflict	多重化回数
	Destroy	破棄回数
	Retry	再発行回数
	KeepTimeout	保持タイムアウト回数
	WatchTimeout	監視タイムアウト回数
解説	LOG 構造体のメンバです。	

4.8.11. LOG_Link 構造体

項目	内容	
要約	ログ情報－リンク情報部	
定義	typedef struct{ long ElapseTime; long WaitFrame; long LinkIn; long LinkOut; long LinkOutBySkip; long AnotherLinkOut; } LOG_Link;	
メンバ	ElapseTime	通算稼働時間（秒単位）
	WaitFrame	フレーム待ち回数
	LinkIn	加入回数
	LinkOut	離脱回数
	LinkOutBySkip	離脱回数(ノードスキップによる)
	AnotherLinkOut	他ノード離脱回数
解説	LOG 構造体のメンバです。	

4.8.12. LOG_Node 構造体

項目	内容	
要約	ログ情報－参加ノード情報部	
定義	typedef struct{ char LinkIn[256]; } LOG_Node;	
メンバ	LinkIn	参加認識ノード一覧
解説	LOG 構造体のメンバです。	

4.8.13. LOG_User 構造体

項目	内容	
要約	ログ情報－製造業者定義可能情報部	
定義	typedef struct{ long Data[16]; } LOG_User;	
メンバ	Data	製造業者定義可能領域（未使用）
解説	LOG 構造体のメンバです。	

4.8.14. HFA_CALLBACKFUNC_V3 構造体

項目	内容	
要約	コールバック登録テーブル（FL-net Ver.3 対応版）	
定義	typedef struct _HFA_CALLBACKFUNC_V3{ CBF_LinkIn LinkIn; CBF_LinkOut LinkOut; CBF_CommonRefresh CommonRefresh; CBF_LogClear LogClear; CBF_RecvReqReadByte RecvReqReadByteBlock; CBF_RecvReqWriteByte RecvReqWriteByteBlock; CBF_RecvReqReadWord RecvReqReadWordBlock; CBF_RecvReqWriteWord RecvReqWriteWordBlock; CBF_RecvRepReadByte RecvRepReadByteBlock; CBF_RecvRepWriteByte RecvRepWriteByteBlock; CBF_RecvRepReadWord RecvRepReadWordBlock; CBF_RecvRepWriteWord RecvRepWriteWordBlock; CBF_RecvRepReadNet RecvRepReadNetParam; CBF_RecvRepWriteNet RecvRepWriteNetParam; CBF_RecvRepControl RecvRepControlEquipment; CBF_RecvRepReadProfile RecvRepReadProfile; CBF_RecvRepReadLog RecvRepReadLog; CBF_RecvRepClearLog RecvRepClearLog; CBF_RecvRepEchoMessage RecvRepEchoMessage; CBF_RecvTransparency RecvTransparency; CBF_LinkInTimeout LinkInTimeout; CBF_SendTimeout SendTimeout; CBF_ReplyTimeout ReplyTimeout; CBF_SendComplete SendComplete; CBF_Error Error; CBF_RecvReqVendor RecvReqVendorMessage; CBF_RecvRepVendor RecvRepVendorMessage; CBF_RecvReqWriteNetParam RecvReqWriteNetParam; CBF_RecvReqControlEquipment RecvReqControlEquipment; CBF_LinkInSlave LinkInSlave; CBF_LinkOutSlave LinkOutSlave; CBF_InputDataRefresh InputDataRefresh; CBF_InputStatusRefresh InputStatusRefresh; CBF_ChangeTokenTimeMeasureStatus ChangeTokenTimeMeasureStatus; CBF_ChangeDataLogMeasureStatus ChangeDataLogMeasureStatus; CBF_RecvReqReadByteBlockFromSettingTool RecvReqReadByteBlockFromSettingTool; CBF_RecvReqReadWordBlockFromSettingTool RecvReqReadWordBlockFromSettingTool; CBF_RecvReqWriteByteBlockFromSettingTool RecvReqWriteByteBlockFromSettingTool; CBF_RecvReqWriteWordBlockFromSettingTool RecvReqWriteWordBlockFromSettingTool; CBF_RecvReqWriteNetParamFromSettingTool RecvReqWriteNetParamFromSettingTool; CBF_RecvReqControlEquipmentFromSettingTool RecvReqControlEquipmentFromSettingTool; CBF_RecvReqSetIOFromSettingTool RecvReqSetIOFromSettingTool; CBF_RecvReqSetConfigParamFromSettingTool RecvReqSetConfigParamFromSettingTool; CBF_RecvReqResetNodeFromSettingTool RecvReqResetNodeFromSettingTool; } HFA_CALLBACKFUNC_V3, *LPHFA_CALLBACKFUNC_V3;	
メンバ	LinkIn	ノードが参加
	LinkOut	ノードが離脱
	CommonRefresh	コモンメモリの値が変化
	LogClear	自ノードのログデータがクリア
	RecvReqReadByteBlock	バイトブロックリード要求メッセージを受信
	RecvReqWriteByteBlock	バイトブロックライト要求メッセージを受信
	RecvReqReadWordBlock	ワードブロックリード要求メッセージを受信
	RecvReqWriteWordBlock	ワードブロックライト要求メッセージを受信
	RecvRepReadByteBlock	バイトブロックリード応答メッセージを受信
	RecvRepWriteByteBlock	バイトブロックライト応答メッセージを受
	RecvRepReadWordBlock	ワードブロックリード応答メッセージを受信
	RecvRepWriteWordBlock	ワードブロックライト応答メッセージを受信
	RecvRepReadNetParam	ネットワークパラメータリード応答メッセージを受信
	RecvRepWriteNetParam	ネットワークパラメータライト応答メッセージを受信
	RecvRepControlEquipment	運転／停止指令応答メッセージを受信
	RecvRepReadProfile	プロファイルリード応答メッセージを受信
	RecvRepReadLog	ログデータリード応答メッセージを受信
	RecvRepClearLog	ログデータクリア応答メッセージを受信
	RecvRepEchoMessage	メッセージ折り返し応答を受信
	RecvTransparency	透過形メッセージメッセージを受信
	LinkInTimeout	リンク参加タイムアウトが発生
	SendTimeout	メッセージ送信タイムアウトが発生

	ReplyTimeout	メッセージ応答受信タイムアウトが発生
	SendComplete	メッセージの送信が完了（正常，異常）
	Error	エラーが発生
	RecvReqVendorMessage	ベンダ固有要求メッセージを受信
	RecvRepVendorMessage	ベンダ固有応答メッセージを受信
	RecvReqWriteNetParam	ネットワークパラメータライト要求メッセージ受信
	RecvReqControlEquipment	運転/停止指令要求メッセージ受信
	LinkInSlave	スレーブノード参加
	LinkOutSlave	スレーブノード離脱
	InputDataRefresh	入力データ変更
	InputStatusRefresh	入力ステータス変更
	ChangeTokenTimeMeasureStatus	トークン保持時間測定状態変化
	ChangeDataLogMeasureStatus	汎用通信データ送信元ログ測定状態変化
	RecvReqReadByteBlockFromSettingTool	設定ツールからのバイトブロックリード要求受信
	RecvReqReadWordBlockFromSettingTool	設定ツールからのワードブロックリード要求受信
	RecvReqWriteByteBlockFromSettingTool	設定ツールからのバイトブロックライト要求受信
	RecvReqWriteWordBlockFromSettingTool	設定ツールからのワードブロックライト要求受信
	RecvReqWriteNetParamFromSettingTool	設定ツールからのネットワークパラメータライト要求受信
	RecvReqControlEquipmentFromSettingTool	設定ツールからの運転/停止指令要求受信
	RecvReqSetIOFromSettingTool	設定ツールからの IO 割付設定要求受信
	RecvReqSetConfigParamFromSettingTool	設定ツールからのコンフィギュレーション用パラメータ設定要求受信
	RecvReqResetNodeFromSettingTool	設定ツールからのノードリセット要求受信
解説	HFA_SetCallbackV3 関数で，コールバック関数を登録する際に使用します。	

4.8.15. HFA_ADDRESS 構造体

項目	内容	
要約	アドレス情報	
定義	typedef struct _HFA_ADDRESS{ unsigned char Area; unsigned short Address; } HFA_ADDRESS, *LPHFA_ADDRESS;	
メンバ	Area	領域［1=領域 1，2=領域 2］
	Address	アドレス［領域 1：0x0000～0x01FF，領域 2：0x0000～0x1FFF］
解説	コモンメモリアドレス情報を定義します。	

4.8.16. HFA_SLAVE_INFO 構造体

項目	内容	
要約	スレーブ登録情報	
定義	typedef struct _HFA_SLAVE_INFO{ unsigned char NodeNo; HFA_ADDRESS InDataAddress; unsigned short InDataSize; unsigned long InputPoint; HFA_ADDRESS OutDataAddress; unsigned short OutDataSize; unsigned long OutputPoint; HFA_ADDRESS InStatusAddress; HFA_ADDRESS OutStatusAddress; } HFA_SLAVE_INFO, *LPHFA_SLAVE_INFO;	
メンバ	NodeNo	ノード番号［1～249］
	InDataAddress	IO 入力データアドレス
	InDataSize	IO 入力データサイズ（ワード単位）
	InputPoint	入力点数
	OutDataAddress	IO 出力データアドレス
	OutDataSize	IO 出力データサイズ（ワード単位）
	OutputPoint	出力点数
	InStatusAddress	入力ステータスアドレス
	OutStatusAddress	出力ステータスアドレス
解説	HFA_SetIO 関数で IO 割付情報を設定する際に使用します。	

4.8.17. HFA_SLAVES_INFO 構造体

項目	内容	
要約	全スレーブ登録情報	
定義	typedef struct _HFA_SLAVES_INFO{ unsigned char Number; HFA_SLAVE_INFO Infos[249]; } HFA_SLAVES_INFO, *LPHFA_SLAVES_INFO;	
メンバ	Number	個数
	Infos	スレーブ登録情報
解説	HFA_GetIO 関数で全スレーブ情報を取得する際に使用します。 Infos 配列は添え字 0 から Number 個数分スレーブ情報が入ります。	

4.8.18. HFA_SLAVE_STATUS 構造体

項目	内容	
要約	スレーブ状態	
定義	typedef struct _HFA_SLAVE_STATUS{ unsigned long InputPoint; unsigned long OutputPoint; unsigned short MasterNodeNo; unsigned char MasterOffOutput; unsigned char RemoteOffOutput; unsigned char RemoteOffInput; unsigned short FreeData[6]; unsigned short GeneralPurpose[6]; } HFA_SLAVE_STATUS, *LP HFA_SLAVE_STATUS;	
メンバ	InputPoint	入力点数
	OutputPoint	出力点数
	MasterNodeNo	マスタノード番号
	MasterOffOutput	マスタ離脱時の IO 出力［0=クリア，1=ホールド］
	RemoteOffOutput	リモート制御フラグ OFF 時の IO 出力［0=クリア，1=ホールド］
	RemoteOffInput	リモート制御フラグ OFF 時の IO 入力［0=クリア，1=ホールド］
	FreeData	マスタからスレーブ毎に指定する設定
	GeneralPurpose	汎用ステータス
解説	スレーブ状態ステータスの入力・出力で共通部分を定義します。	

4.8.19. HFA_SLAVE_INPUT_STATUS 構造体

項目	内容	
要約	入力ステータス	
定義	typedef struct _HFA_SLAVE_INPUT_STATUS{ unsigned short StatusNo; HFA_SLAVE_STATUS InputStatus; } HFA_SLAVE_INPUT_STATUS, *LP HFA_SLAVE_INPUT_STATUS;	
メンバ	StatusNo	スレーブ状態番号［0=未加入，1=停止中，2=接続処理中，3=動作中，4=マスタ不在，5=自ノード離脱，6=設定異常］
	InputStatus	スレーブ状態
解説	HFA_GetInputStatus 関数で入力ステータスを取得する際に使用します。	

4.8.20. HFA_SLAVE_OUTPUT_STATUS 構造体

項目	内容	
要約	出力ステータス	
定義	typedef struct _HFA_SLAVE_OUTPUT_STATUS{ unsigned short RemoteControl; HFA_SLAVE_STATUS OutputStatus; } HFA_SLAVE_OUTPUT_STATUS, *LP HFA_SLAVE_OUTPUT_STATUS;	
メンバ	RemoteControl	リモート制御［0=リモート停止，1=リモート動作］
	OutputStatus	スレーブ状態
解説	HFA_SetOutputStatus 関数・HFA_GetOutputStatus 関数で、出力ステータスを設定・取得する際に使用します。	

4.8.21. HFA_LOG_TOKENTIME 構造体

項目	内容	
要約	トークン保持時間測定項目	
定義	typedef struct _HFA_LOG_TOKENTIME{ long Destroy; long DestroyTime; long Retry; long RetryTime; long KeepTimeout; long KeepTimeoutTime; long WatchTimeout; long WatchTimeoutTime; long MaxHoldTime; long MinHoldTime; long MaxHoldTimeDetect; long HoldTimeMeasure; long HoldTimeMeasureCount; long RefreshCycleTime; long RecvCyclic; long RecvCyclicErr; long RecvCyclicErrTime; long RecvMessageErr; long RecvMessageErrTime; }HFA_LOG_TOKENTIME, *LPHFA_LOG_TOKENTIME;	
メンバ	Destroy	トークン破棄回数
	DestroyTime	トークン破棄検出直近の時間
	Retry	トークン再発行回数
	RetryTime	トークン再発行直近の時間
	KeepTimeout	トークン保持タイムアウト回数
	KeepTimeoutTime	トークン保持タイムアウト直近の時間
	WatchTimeout	トークン監視タイムアウト回数
	WatchTimeoutTime	トークン監視タイムアウト直近の時間
	MaxHoldTime	トークン保持時間最大値
	MinHoldTime	トークン保持時間最小値
	MaxHoldTimeDetect	トークン保持時間最大値検出時間
	HoldTimeMeasure	トークン保持時間測定時間
	HoldTimeMeasureCount	トークン保持時間測定中のトークン回数
	RefreshCycleTime	リフレッシュサイクル最大値検出時の時間
	RecvCyclic	サイクリックフレーム受信回数
	RecvCyclicErr	サイクリック伝送受信エラー回数
	RecvCyclicErrTime	サイクリック伝送受信エラー検出時間
	RecvMessageErr	メッセージ伝送受信エラー回数
	RecvMessageErrTime	メッセージ伝送受信エラー検出時間
解説	HFA_StopTokenTimeMeasure 関数で、トークン保持時間測定情報を取得する際に使用します。	

4.8.22. HFA_COMMON_MEMORY 構造体

項目	内容	
要約	コモンメモリ情報	
定義	typedef struct _HFA_COMMON_MEMORY { long Addr1; long Size1; long Addr2; long Size2; } HFA_COMMON_MEMORY, *LPHFA_COMMON_MEMORY;	
メンバ	Addr1	領域 1 アドレス（ワード単位）
	Size1	領域 1 サイズ（ワード単位）
	Addr2	領域 2 アドレス（ワード単位）
	Size2	領域 2 サイズ（ワード単位）
解説	コモンメモリ領域 1，2 の定義です。	

4.8.23. HFA_CONFIG_PARAM 構造体

項目	内容	
要約	コンフィギュレーション用パラメータ	
定義	typedef struct _HFA_CONFIG_PARAM { long NodeNo; long Common1Addr; long Common1Bytes; long Common2Addr; long Common2Words; long TokenWatchTime; long MinFrameInterval; char NodeName[11]; char LocalIP[20]; } HFA_CONFIG_PARAM, *LPHFA_CONFIG_PARAM;	
メンバ	NodeNo	ノード番号
	Common1Addr	領域 1 アドレス（バイト単位）
	Common1Bytes	領域 1 サイズ（バイト単位）
	Common2Addr	領域 2 アドレス（ワード単位）
	Common2Words	領域 2 サイズ（ワード単位）
	TokenWatchTime	トークン監視時間（ms 単位）
	MinFrameInterval	最小許容フレーム間隔（100μs 単位）
	NodeName	ノード名(設備名)（ASCII 文字で最大 10 文字）
	LocalIP	ローカル IP アドレス
解説	HFA_SetConfigParam 関数・HFA_GetConfigParam 関数で、コンフィギュレーション用パラメータを設定・取得する際に使用します。	

4.8.24. HFA_REPLY_DATA 構造体

項目	内容	
要約	応答データ	
定義	typedef struct _HFA_REPLY_DATA { long Result; unsigned char Data[1024]; long ErrBytes; } HFA_REPLY_DATA, *LPHFA_REPLY_DATA;	
メンバ	Result	応答種別［0=正常, 1=異常, その他=非実装］
	Data	応答時のデータ
	ErrBytes	エラー情報のバイト数［0～1024］
解説	要求メッセージに応答メッセージを返す際の応答データを定義します。	

4.8.25. HFA_LOG_V3 構造体

項目	内容	
要約	自ノードログ情報（FL-net Ver.3 対応版）	
定義	typedef struct{ LOG_Protocol Protocol; LOG_Frame Frame; HFA_LOG_Cyclic_V3 Cyclic; HFA_LOG_Message_V3 Message; LOG_ACK Ack; HFA_LOG_Token_V3 Token; LOG_Link Link; LOG_Node Node; LOG_User User; HFA_LOG_IP_V3 IP; } HFA_LOG_V3, *LPHFA_LOG_V3;	
メンバ	Protocol	プロトコル情報
	Frame	フレーム情報
	Cyclic	サイクリック情報（FL-net Ver.3 対応）
	Message	メッセージ情報（FL-net Ver.3 対応）
	Ack	ACK 情報
	Token	トークン情報（FL-net Ver.3 対応）
	Link	リンク情報
	Node	参加ノード情報
	User	製造業者定義可能情報
	IP	IP 情報（FL-net Ver.3 対応）
解説	FL-net Ver.3 に対応したログ構造体の定義です。 HFA_接頭辞が付いていない構造体のメンバは Ver.2 と同じです。	

4.8.26. HFA_LOG_Cyclic_V3 構造体

項目	内容	
要約	サイクリックログ情報（FL-net Ver.3 対応版）	
定義	typedef struct{ long RecvCyclicErr; long CyclicAddrErr; long CyclicCbnErr; long CyclicTbnErr; long CyclicBsizeErr; long RecvCyclicErrTime; } HFA_LOG_Cyclic_V3, LPHFA_LOG_Cyclic_V3;	
メンバ	RecvCyclicErr	受信エラー回数
	CyclicAddrErr	アドレスサイズエラー回数
	CyclicCbnErr	CBN エラー回数
	CyclicTbnErr	TBN エラー回数
	CyclicBsizeErr	Bsize エラー回数
	RecvCyclicErrTime	受信エラー検出時間（FL-net Ver.3 対応）
解説	HFA_LOG_V3 構造体のメンバです。 LOG_Cyclic 構造体の FL-net Ver.3 対応版で、一部のメンバが追加されています。	

4.8.27. HFA_LOG_Message_V3 構造体

項目	内容	
要約	メッセージログ情報（FL-net Ver.3 対応版）	
定義	typedef struct{ long Resend long RetryOut; long RefreshCycleTime; long RecvErr; long SeqVerErr; long SeqVerify; long RecvMessageErrTime; } HFA_LOG_Message_V3, LPHFA_LOG_Message_V3;	
メンバ	Resend	再送回数
	RetryOut	再送オーバー回数
	RefreshCycleTime	リフレッシュサイクル最大値検出時の時間（FL-net Ver.3 対応）
	RecvErr	受信エラー回数
	SeqVerErr	通番バージョンエラー回数
	SeqVerify	通番再送認識回数
	RecvMessageErrTime	メッセージ伝送受信エラー検出時間（FL-net Ver.3 対応）
解説	HFA_LOG_V3 構造体のメンバです。 LOG_Message 構造体の FL-net Ver.3 対応版で、一部のメンバが追加されています。	

4.8.28. HFA_LOG_Token_V3 構造体

項目	内容	
要約	トークンログ情報（FL-net Ver.3 対応版）	
定義	typedef struct{ long Conflict; long Destroy; long Retry; long DestroyTime; long RetryTime; long KeepTimeoutTime; long KeepTimeout; long WatchTimeout; long WatchTimeoutTime; long MaxHoldTime; long MinHoldTime; long MaxHoldTimeDetect; long HoldTimeMeasure; long HoldTimeMeasureCount; } HFA_LOG_Token_V3, LPHFA_LOG_Token_V3;	
メンバ	Conflict	多重化回数
	Destroy	破棄回数
	Retry	再発行回数
	DestroyTime	トークン破棄検出直近の時間（FL-net Ver.3 対応）
	RetryTime	トークン再発行直近の時間（FL-net Ver.3 対応）
	KeepTimeoutTime	トークン保持タイムアウト直近の時間（FL-net Ver.3 対応）
	KeepTimeout	保持タイムアウト回数
	WatchTimeout	監視タイムアウト回数
	WatchTimeoutTime	トークン監視タイムアウト直近の時間（FL-net Ver.3 対応）
	MaxHoldTime	トークン保持時間最大値（FL-net Ver.3 対応）
	MinHoldTime	トークン保持時間最小値（FL-net Ver.3 対応）
	MaxHoldTimeDetect	トークン保持時間最大値検出時間（FL-net Ver.3 対応）
	HoldTimeMeasure	トークン保持時間測定時間（FL-net Ver.3 対応）
	HoldTimeMeasureCount	トークン保持時間測定中のトークン回数（FL-net Ver.3 対応）
解説	HFA_LOG_V3 構造体のメンバです。 LOG_Token 構造体の FL-net Ver.3 対応版で、一部のメンバが追加されています。	

4.8.29. HFA_IP 構造体

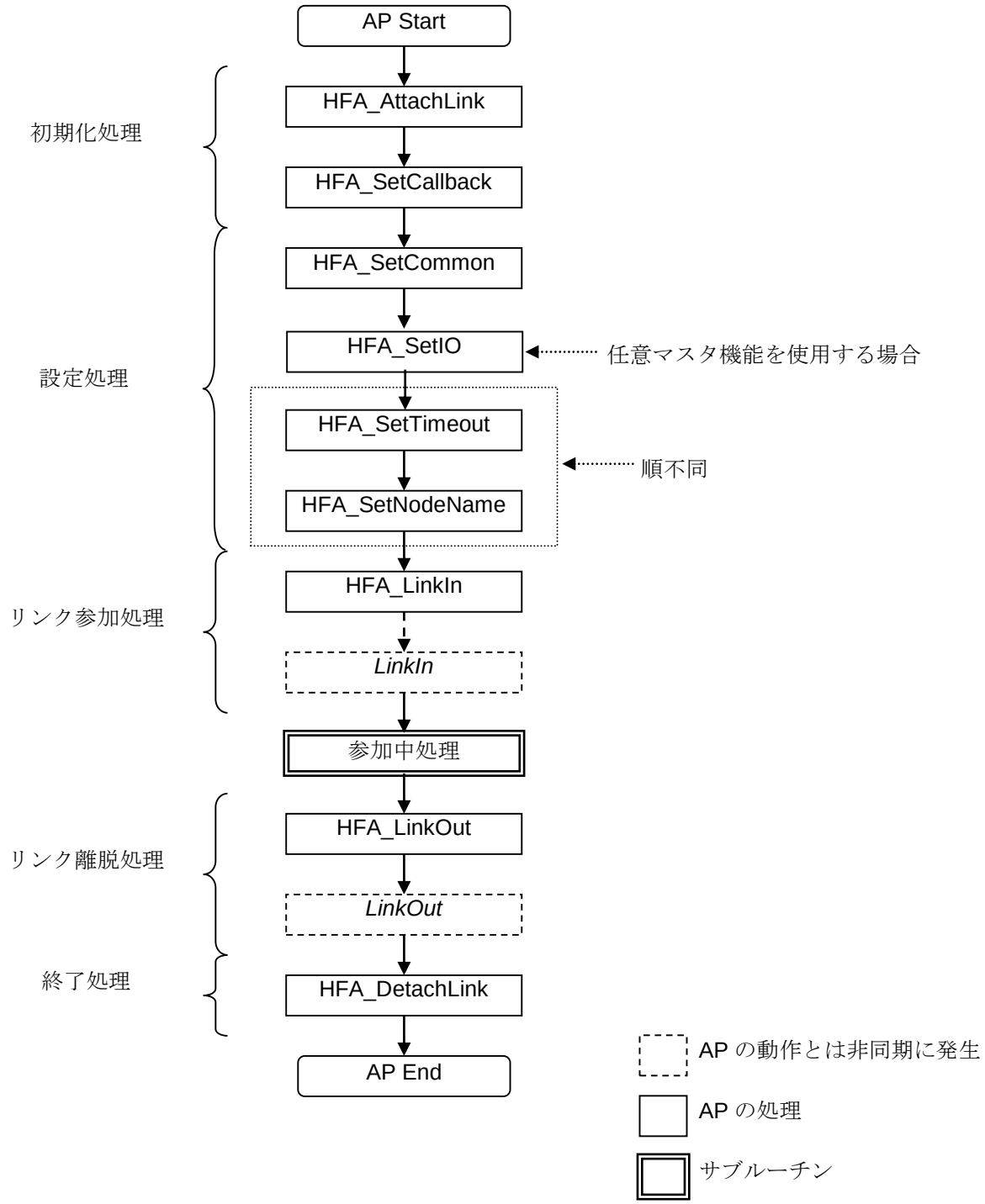
項目	内容	
要約	IP 情報	
定義	typedef struct _HFA_IP { unsigned long Address; unsigned long Counter; } HFA_IP, *LPHFA_IP;	
メンバ	Address	IP アドレス［0=無］
	Counter	受信カウンタ
解説	IP 情報の定義です。	

4.8.30. HFA_LOG_IP_V3 構造体

項目	内容	
要約	汎用通信データ送信元ログ	
定義	typedef struct _HFA_LOG_IP_V3 { unsigned long Time; HFA_IP Log[10]; } HFA_LOG_IP_V3, *LPHFA_LOG_IP_V3;	
メンバ	Time	測定時間
	Log	ログ情報
解説	HFA_LOG_V3 構造体のメンバです。 HFA_StopDataLogMeasure 関数で汎用通信データ送信元ログ情報を取得する際に使用します。	

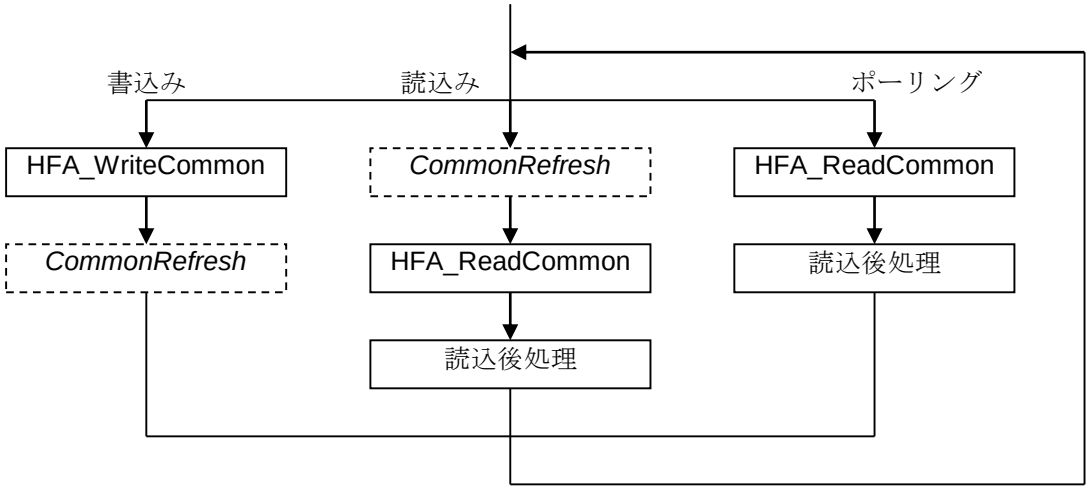
5. I/F 関数使用例

「***DLL***」の I/F 関数の使用例を以下に示します。

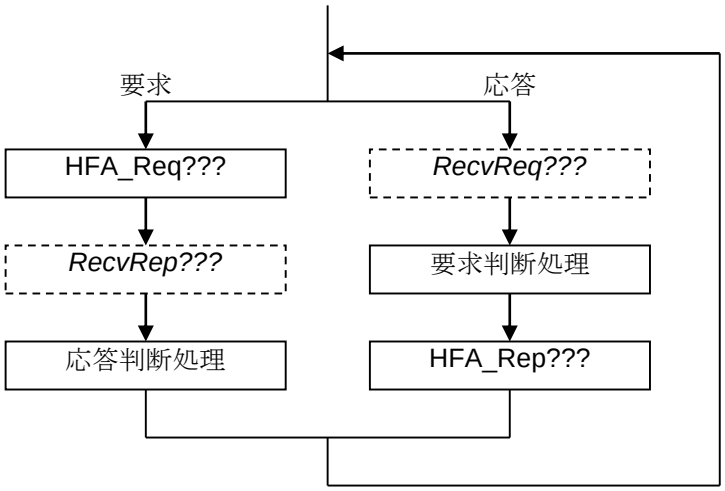


参加中処理として行う内容の例として、以下の組合せが考えられます。

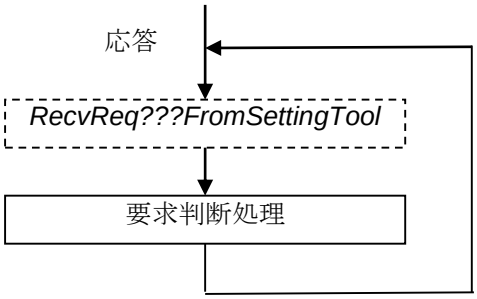
1）コモンメモリアクセス



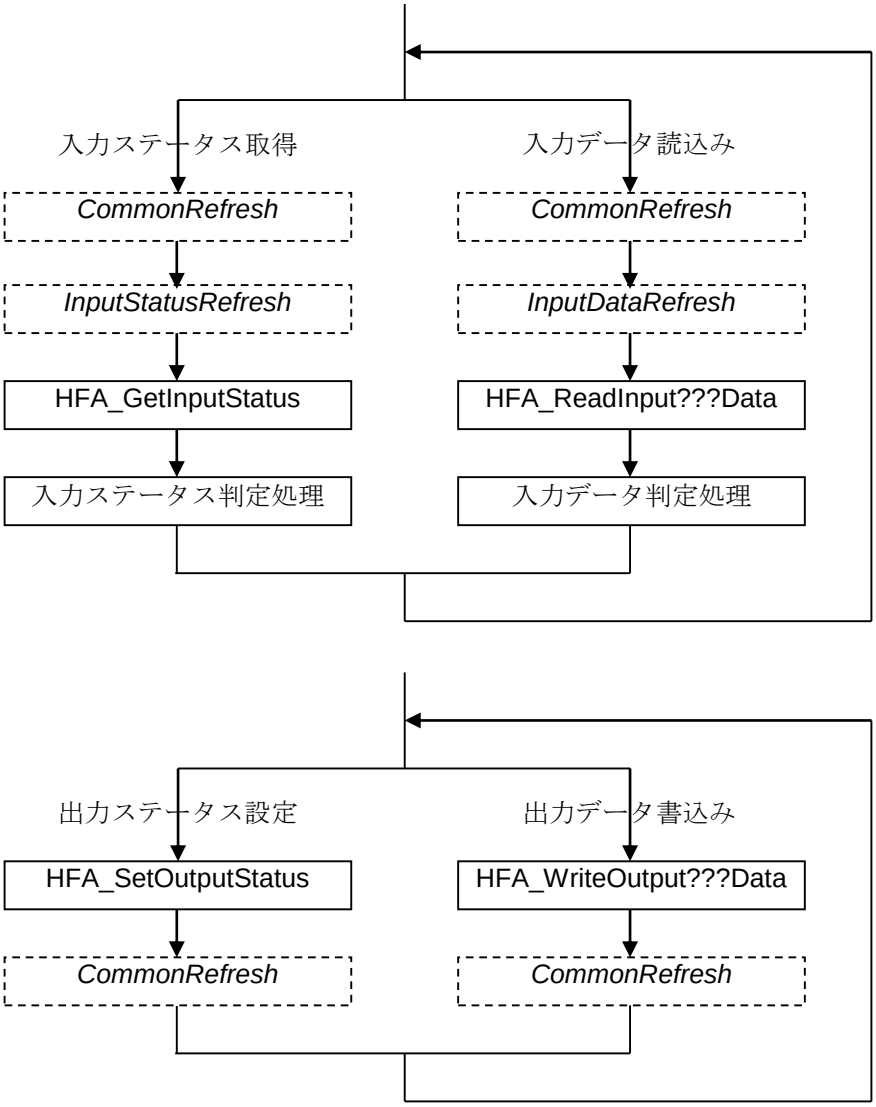
2）メッセージ送受信



3）コマンドサーバメッセージ受信



４）デバイスレベルネットワーク（IO データ交換）



6. DLL 内部仕様

6.1. WinSock ポート番号

FA リンクプロトコル通信では、以下に示す通信ポート番号を使用します。他のアプリケーション等で以下の通信ポート番号を使用しないでください。

通信ポート番号	用途
55000	トークンフレーム受信 サイクリック伝送受信
55001	メッセージ伝送受信
55002	トリガフレーム受信 参加要求フレーム受信 勧誘フレーム受信
55003	FA リンクプロトコル送信 勧誘フレーム送信
55004	汎用コマンド送受信

6.2. DLL のデフォルト値

「***DLL***」内部で管理する情報のデフォルト値を以下に示します。

1） コモンメモリ送信領域

項 目	設定値
コモンメモリ領域１アドレス	0x0000
コモンメモリ領域１サイズ	0x0000
コモンメモリ領域２アドレス	0x0000
コモンメモリ領域２サイズ	0x0000

デフォルト状態では、コモンメモリ送信領域はありません。送信領域を設定する場合は、HFA_SetCommon 関数をコールしてください。

2） 自ノード管理情報

項 目	設定値
ベンダ名称	HITACHI-KE
製造業者形式	KE-SFL3WIN
ノード名称	FLnet△CTRL
運転／停止状態	運転状態

△：スペース

ノード名称を変更する場合は、HFA_SetNodeName 関数をコールしてください。

運転／停止状態を変更する場合は、HFA_SetControlEquipment 関数をコールしてください。

ベンダ名称および製造業者形式は固定値のため、「***APP***」から変更することはできません。

3）コンフィギュレーション用パラメータ

項 目	設定値
ノード番号	デフォルト NIC アダプタの IP アドレスの 4 バイト目
トークン監視時間	255
最小許容フレーム間隔	0
使用する NIC アダプタの IP アドレス	""（NIC アダプタ特定なし）

ノード番号を変更する場合は、HFA_SetNodeNo 関数をコールしてください。

トークン監視時間を変更する場合は、HFA_SetTokenWatchTime 関数をコールしてください。

最小許容フレーム間隔を変更する場合は、HFA_SetMinFrameInterval 関数をコールしてください。

IP アドレスを変更する場合は、HFA_SetIP 関数をコールしてください。

4）デバイスレベルネットワーク用パラメータ

項 目	設定値
IO 割付情報	無し（任意マスタ機能無し）

IO 割付情報を設定する場合は、HFA_SetIO 関数をコールしてください。

5）イベント通知

デフォルト状態では、イベント通知は全て無効です。イベント通知を有効にする場合は、HFA_SetCallback 関数または HFA_SetCallbackV3 関数をコールしてください。

6）タイムアウト値

項 目	設定値
ネットワーク参加タイムアウト時間	0 ms
メッセージ送信完了タイムアウト時間	0 ms
メッセージ応答受信タイムアウト時間	0 ms

デフォルト状態では、タイムアウトイベントは発生しません。設定値を変更する場合は、HFA_SetTimeout 関数をコールしてください。

7）コモンメモリ更新イベント検知範囲

項 目	設定値
コモンメモリ領域１アドレス	0x0000
コモンメモリ領域１サイズ	0x0000
コモンメモリ領域２アドレス	0x0000
コモンメモリ領域２サイズ	0x0000

デフォルト状態では、コモンメモリ全領域を対象とします。範囲を変更する場合は、HFA_SetCommonRefreshDegree 関数をコールしてください。

8） ロギング出力項目

項 目	設定値
関数コール結果正常時のロギング	無し
イベント通知時のロギング	無し
サイクリック通信のアナライズ	無し
メッセージ通信のアナライズ	無し
参加要求通信のアナライズ	無し
デバッグ情報	無し
「***DLL***」管理情報種別のロギング	有り
警告種別のロギング	有り
エラー種別のロギング	有り
関数コール結果警告時のロギング	有り
関数コール結果異常時のロギング	有り

設定値を変更する場合は、HFA_DebugLog 関数をコールしてください。

6.3. ノード状態

「***DLL***」内部で管理するノード状態には、上位層の状態、FA リンクの状態、自ノードの状態があります。それぞれの状態に含まれる項目、構成の仕様および自ノード管理情報の値を以下に示します。

1） 上位層の状態（ULS）

＜構成＞

ビット	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
値	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**?**

＜項目仕様＞

ビット	項目	内容	自ノード管理情報の値
0～11	上位層のエラー内容	上位層で定義可能	全ビット 0 固定
12	予備	0 固定	0 固定
13,14	上位層のエラー情報	00=NORMAL 01=WARNING 10=ALARM 11=ALARM	00 固定
15	上位層の動作情報	0=STOP 1=RUN	HFA_SetControlEquipment 関数の設定値

2） FA リンクの状態

＜構成＞

ビット	0	1	2	3	4	5	6	7
値	**?**	**?**	**0**	**0**	**0**	**?**	**1**	**?**

＜項目仕様＞

ビット	項目	内容	自ノード管理情報の値
0	ノードの参加離脱	0=離脱 1=参加中	HFA_GetNodeStatus 関数で、ノード管理情報を読出した場合、本ビットはノードの参加状態を示します。ただし、ネットワークに送信するフレームについては、本ビットの値は 0 固定となります。
1	通信無効検知	0=検知なし 1=検知あり	HFA_GetNodeStatus 関数で、ノード管理情報を読出した場合、本ビットはノードの通信無効検知の有無を示します。ただし、ネットワークに送信するフレームについては、本ビットの値は 0 固定となります。
2,3	予備	0 固定	0 固定
4	上位層動作信号エラー	0=エラーなし 1=エラーあり	0 固定
5	コモンメモリ･データ有効通知	0=無効 1=有効	コモンメモリ送信領域の範囲に応じます。 ・有効：送信領域=有の場合 ・無効：送信領域=無（受信専用）の場合
6	コモンメモリ（アドレス・サイズ）設定完了	0=未完了 1=完了	1 固定
7	アドレス重複検知	0=重複なし 1=重複あり	アドレス重複検知状態に応じます。

3）自ノードの状態

＜構成＞

ビット	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
値	?	?	?	?	?	0	0	0	?	?	?	?	?	?	?	?

＜項目仕様＞

ビット	項目	値
0	ノード番号重複フラグ	0=重複なし，1=重複あり
1	トークン監視時間エラー	0=エラーなし，1=エラーあり
2	受信待ち状態	0=待ちなし，1=受信待ち
3	初期化エラー	0=エラーなし，1=エラーあり
4	参加状態	0=離脱，1=参加中 ※モニタモードを実行中の場合は、トークンモード=1 のネットワーク確立状態（0=未確立，1=確立）を示します。なお、トークンモード=0（Ver.1.00 版）ネットワークが確立している場合は、0 となります。
5～7	予備	0 固定
8～15	自ノードステータス番号	FA リンクプロトコル仕様の状態遷移番号。 詳細は、以下をご参照ください。

※自ノードステータス番号の値

値	状 態	内 容
0	モニタモード中	モニタモード実行中の状態。
1	上位層からの初期化待ち	ネットワーク加入に必要なパラメータの設定を上位層から設定されるのを待つ状態。
2	加入トークン検出待ち	ネットワークが稼働中かチェックを行う状態。
3	トリガ送信または、 受信待ち	新規加入の同期をとるためのトリガによる同期を取るための状態。
4	参加要求受付	新規加入時、トリガによって同期をとった参加要求受付時間に参加要求フレームによるネットワーク情報の確立を行う状態。
5	トークン周回の３周待ち	途中加入時、トークンによってネットワークの情報を収集する状態。
6	参加要求送信待ち	途中加入時、ネットワークの情報の収集が終了後、参加要求フレームの送信を行う状態。
7	トークン待ち	自ノード宛のトークンの受信を待ち、他ノードの監視を行う状態。
8	トークン保持	自ノード宛のトークンを受信してから、次ノード宛のトークンを送信するまでの状態。

6.4. メッセージ受信時の処理

他ノードからメッセージを受信した場合、「***DLL***」は以下の処理を行います。

1）メッセージ受付処理

・ 不明なメッセージ番号を受信した場合は、メッセージを破棄します。
・ 1 対 1 メッセージを受信した場合は、ACK を送信します。（1 対 n メッセージの場合は、送信しません。）
・ 受信したメッセージのフォーマットをチェックします。（チェック内容は、メッセージ内容によって異なります。）フォーマット異常を検知した場合は、異常 ACK（R_STS=6）を送信します。「***APP***」への受信イベントは発生しません。
・ 同一通番の再送メッセージを受信した場合は、正常 ACK を送信します。「***APP***」への受信イベントは発生しません。

2）要求メッセージ受信処理

メッセージ内容に応じて以下の 2 種類の動作を行います。

・ 「***DLL***」で自動応答可能なメッセージの場合は、「***DLL***」で自動的に応答メッセージを送信します。
・ 「***DLL***」で応答不可能なメッセージの場合は、「***APP***」に受信イベントを通知します。「***APP***」は、受信内容を判断して、応答メッセージを送信する必要があります。（ただし、「***APP***」側へのコールバック関数が未登録の場合は、「***DLL***」が自動的に非実装（M_RLT=2）の応答メッセージを送信します。）

3）応答メッセージ受信処理

他ノードから応答メッセージを受信した場合、「***APP***」に受信イベントを通知します。「***APP***」は、要求メッセージとの関連性を判断する必要があります。なお、「***APP***」側へのコールバック関数が未登録の場合は、受信イベントは発生しません。

メッセージ内容毎の詳細を以下に示します。

6.4.1. バイトブロックリード

1）要求受信（TCD=65003）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② *RecvReqReadByteBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Addr* = M_ADD

・*Bytes* = M_SZ

③ 応答送信

RecvReqReadByteBlock コールバック関数の登録有無で動作が異なります。

・未登録の場合：バイトブロックリード非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：「***APP***」で HFA_RepReadByteBlock 関数をコールしてください。

2）応答受信（TCD=65203）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0 の場合：0≧TFL－64、1024＜TFL－64（データ無、サイズオーバー）

・M_RLT=1 の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

・M_RLT=2 の場合：TFL≠64（データ有）

② *RecvRepReadByteBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*Addr* = M_ADD

・*Bytes* = M_SZ

・*Data* = 受信データアドレス（M_RLT=0）、NULL（M_RLT=1, 2）

・*ErrBytes* = TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.2. バイトブロックライト

１）要求受信（TCD=65004）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvReqWriteByteBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Addr* = M_ADD

・*Bytes* = M_SZ

・*Data* = 受信データアドレス（「***DLL***」内メモリのポインタ）

③ 応答送信

RecvReqWriteByteBlock コールバック関数の登録有無で動作が異なります。

・未登録の場合：バイトブロックライト非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：「***APP***」で HFA_RepWriteByteBlock 関数をコールしてください。

２）応答受信（TCD=65204）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0, 2 の場合：TFL≠64（データ有）

・M_RLT=1　の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvRepWriteByteBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*Addr* = M_ADD

・*Bytes* = M_SZ

・*ErrBytes* =TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.3. ワードブロックリード

1）要求受信（TCD=65005）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② *RecvReqReadWordBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Addr* = M_ADD

・*Words* = M_SZ

③ 応答送信

RecvReqReadWordBlock コールバック関数の登録有無で動作が異なります。

・未登録の場合：ワードブロックリード非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：「***APP***」で HFA_RepReadWordBlock 関数をコールしてください。

2）応答受信（TCD=65205）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0 の場合：0≧TFL－64、1024＜TFL－64（データ無、サイズオーバー）

・M_RLT=1 の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

・M_RLT=2 の場合：TFL≠64（データ有）

② *RecvRepReadWordBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*Addr* = M_ADD

・*Words* = M_SZ

・*Data* = 受信データアドレス（M_RLT=0）、NULL（M_RLT=1, 2）

・*ErrBytes* = TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.4. ワードブロックライト

１）要求受信（TCD=65006）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvReqWriteWordBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Addr* = A_ADD

・*Bytes* = W_SZ

・*Data* = 受信データアドレス（「***DLL***」内メモリのポインタ）

③ 応答送信

RecvReqWriteWordBlock コールバック関数の登録有無で動作が異なります。

・未登録の場合：ワードブロックライト非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：「***APP***」で HFA_RepWriteWordBlock 関数をコールしてください。

２）応答受信（TCD=65206）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0, 2 の場合：TFL≠64（データ有）

・M_RLT=1　の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvRepWriteWordBlock* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*Addr* = M_ADD

・*Words* = M_SZ

・*ErrBytes* =TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.5. ネットワークパラメータリード

1）要求受信（TCD=65007）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② イベント通知

なし

③ 応答送信

ノード管理情報パラメータ（「***DLL***」内部メモリ）を参照して、ネットワークパラメータリード正常応答（M_RLT=0）を自動的に送信します。

2）応答受信（TCD=65207）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0 の場合：TFL≠120（サイズ不一致）

・M_RLT=1 の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

・M_RLT=2 の場合は、異常

② *RecvRepReadNetParam* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*ErrBytes* =TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.6. ネットワークパラメータライト

1）要求受信（TCD=65008）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠92（サイズ不一致）

② イベント通知

・*NodeNo* = SA

・*Common1Addr*= コモンメモリ 1 アドレス

・*Common1Bytes*= コモンメモリ 1 サイズ

・*Common2Addr*= コモンメモリ 2 アドレス

・*Common2Words*= コモンメモリ 2 サイズ

・*NodeName*= ノード名称

・*SetMask*= 設定マスク

③ 応答送信

RecvReqWriteNetParam コールバック関数の登録有無で動作が異なります。

・未登録の場合：ネットワークパラメータライト非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：「***APP***」で HFA_RepWriteNetParam 関数をコールしてください。

2）応答受信（TCD=65208）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0, 2 の場合：TFL≠64（データ有）

・M_RLT=1　の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvRepWriteNetParam* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*ErrBytes* =TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.7. 停止指令

1）要求受信（TCD=65009）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② イベント通知

・*NodeNo* = SA

・*Command*= 制御指令

③ 応答送信

RecvReqControlEquipment コールバック関数の登録有無で動作が異なります。

・未登録の場合：停止指令非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：「***APP***」で HFA_RepControlEquipment 関数をコールしてください。

2）応答受信（TCD=65209,）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0, 2 の場合：TFL≠64（データ有）

・M_RLT=1　の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvRepControlEquipment* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Command* = 0

・*Result* = M_RLT

・*ErrBytes* =TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.8. 運転指令

1）要求受信（TCD=65010）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② イベント通知

・*NodeNo* = SA

・*Command*= 制御指令

③ 応答送信

RecvReqControlEquipment コールバック関数の登録有無で動作が異なります。

・未登録の場合：運転指令非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：「***APP***」で HFA_RepControlEquipment 関数をコールしてください。

2）応答受信（TCD=65210）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0, 2 の場合：TFL≠64（データ有）

・M_RLT=1　の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvRepControlEquipment* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Command* = 1

・*Result* = M_RLT

・*ErrBytes* =TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.9. プロファイルリード

1）要求受信（TCD=65011）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② イベント通知

なし

③ 応答送信

プロファイル情報（付録 5 参照）を元に、プロファイルリード正常応答（M_RLT=0）を自動的に送信します。

2）応答受信（TCD=65211）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0, 1 の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

・M_RLT=　2 の場合は、異常

② *RecvRepReadProfile* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*Size* = TFL – 64（M_RLT=0, 1）、0（M_RLT=2）

・*Data* = 受信したデータアドレス（M_RLT=0, 1）、NULL（M_RLT=2）

6.4.10. ログデータリード

1）要求受信（TCD=65013）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② イベント通知

なし

③ 応答送信

自ノードログ情報（「***DLL***」内部メモリ）を参照して、ログデータリード正常応答（M_RLT=0）を自動的に送信します。

2）応答受信（TCD=65213）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0 の場合：TFL≠576（サイズ不一致）

・M_RLT=1 の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

・M_RLT=2 の場合は、異常

② *RecvRepReadLog* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*Size* = 512（M_RLT=0）、TFL－64（M_RLT=1）、0（M_RLT=2）

・*Log* = 受信データアドレス（M_RLT=0, 1）、NULL（M_RLT=2）

6.4.11. ログデータクリア

1）要求受信（TCD=65014）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・TFL≠64（データ有）

② イベント通知

なし

③ 応答送信

自ノードログ情報（「***DLL***」内部メモリ）をクリアし、ログデータクリア正常応答（M_RLT=0）を自動的に送信します。

2）応答受信（TCD=65214）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・M_RLT=0 の場合：TFL≠64（データ有）

・M_RLT=1 の場合：0＞TFL－64、1024＜TFL－64（サイズオーバー）

・M_RLT=2 の場合は、異常

② *RecvRepClearLog* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Result* = M_RLT

・*ErrBytes* = TFL－64（M_RLT=1）、0（M_RLT=0, 2）

・*Error* = 受信データアドレス（M_RLT=1）、NULL（M_RLT=0, 2）

6.4.12. メッセージ折り返し

1）要求受信（TCD=65015）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・0＞TFL－64、1024＜TFL－64（サイズオーバー）

② イベント通知

なし

③ 応答送信

要求データと同一の内容で、メッセージ折返し正常応答（M_RLT=0）を自動的に送信します。

2）応答受信（TCD=65215）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・0＞TFL－64、1024＜TFL－64（サイズオーバー）

・M_RLT≠0（正常応答以外）

② *RecvRepEchoMessage* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Bytes* = TFL－64

・*Message* = 受信データアドレス（ローカルメモリのポインタ）

6.4.13. ベンダ固有メッセージ

１）要求受信（TCD=65016）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・0＞TFL－80、1024＜TFL－80（サイズオーバー）

② *RecvReqVendorMessage* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*Multi* = 0（DA≠255 の場合），1（DA=255 の場合）

・*VendorName* = VNAME（サイズ=10 バイト）

・*SubCode* = SCODE（サイズ=6 バイト）

・*Bytes* = TFL－80（M_RLT=0, 1 の場合），0（M_RLT=2 の場合）

・*Data* = 受信データアドレス（サイズ=TFL－80，M_RLT=0, 1 の場合），NULL（M_RLT=2）

③ 応答送信

RecvReqVendorMessage コールバック関数の登録有無で動作が異なります。

・未登録の場合：DA≠255 の場合、ベンダ固有メッセージ非実装応答（M_RLT=2）を自動的に送信します。

・登録済の場合：Multi≠0 の場合、「***APP***」で HFA_RepVendorMessage 関数をコールしてください。

２）応答受信（TCD=65216）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・0＞TFL－80、1024＜TFL－80（サイズオーバー）

② *RecvRepVendorMessage* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*VendorName* = VNAME（サイズ=10 バイト）

・*SubCode* = SCODE（サイズ=6 バイト）

・*Bytes* = TFL－80（M_RLT=0, 1 の場合），0（M_RLT=2 の場合）

・*Data* = 受信データアドレス（サイズ=TFL－80，M_RLT=0, 1 の場合），NULL（M_RLT=2）

6.4.14. 透過形メッセージ

１）要求・応答受信（TCD=0～59999）

① ACK 送信（DA≠255 の場合）

以下の場合、R_STS=6 となります。

・0＞TFL－64、1024＜TFL－64（サイズオーバー）

② *RecvTransparency* イベント通知

通知する引数内容は、以下の通りです。

・*NodeNo* = SA

・*TransactionCode* = TCD

・*Bytes* = TFL – 64

・*Message* = 受信データアドレス（ローカルメモリのポインタ）

7. 制限事項

1） 本製品は通信ポート 55000 から 55004 を使用します。他のアプリケーション等がこれらのポートを使用している場合、本製品はエラーとなり動作しません。他のアプリケーションを終了してください。

2） Windows ファイアウォールが有効になっている場合、本製品は通信ポート 55000 から 55004 を使用しますので、ファイアウォールのブロックを解除する必要があります。

3） デバッグ用のブレイクポイントで停止させた場合、イベント抜けや動作が不安定になることがあります。これらのことを十分考慮した上でデバッグを行ってください。動作が不安定になった場合は、コンピュータのリセットを行ってください。

4） 表示処理、印字処理やその他の処理によって CPU 負荷が高くなると通信エンジンのレスポンスは低下します。通信処理のレスポンスが低下すると、FL-net ネットワークから離脱したり、その他のノード影響を与える可能性があります。ネットワーク障害が多発する場合は、ネットワークの設定（トークン監視時間など）を変更して、ネットワークが正しく動作するように調整してください。ネットワークの設定を変更しても正常に動作しない場合は、パソコンの CPU 負荷が高くなるような表示処理、印字処理やその他の処理の負荷を減らしてください。

5） 本製品を使用するパソコンに WinSock Proxy Client を導入している場合は、ネットワーク参加時にエラーを検知して、ネットワークに参加できない場合があります。ネットワークに参加できない場合は、WinSock Proxy Client の使用を無効にしてください。

6） コールバック関数の使用方法によっては、イベントを取りこぼす可能性があります。

7） メッセージ送信中に他ノードから自動応答を伴う要求メッセージを受信した場合に、要求メッセージを取りこぼす可能性があります。

8） 本製品内でネットワークエラー等を検知した場合、OS が管理するイベントログにエラーの内容が出力されます。イベントログの設定によっては、イベントログがオーバーフローして、「アプリケーションログが満杯です。」のエラーダイアログが表示される場合があります。イベントビューア画面にて、イベントログの容量を大きく設定してください。

付録１　メッセージ番号

メッセージ伝送で使用するメッセージ番号（トランザクションコード）の一覧を以下に示します。メッセージ番号は、「***APP***」にメッセージ送信完了（*SendComplete*），メッセージ送信タイムアウト（*SendTimeout*），メッセージ応答タイムアウト（*ReplyTimeout*）のイベントを通知する際に使用します。

メッセージ種類	要 求	応 答
リザーブ（透過形メッセージ）	0～9999	0～9999
透過形メッセージ	10000～59999	10000～59999
バイトブロックリード	65003	65203
バイトブロックライト	65004	65204
ワードブロックリード	65005	65205
ワードブロックライト	65006	65206
ネットワークパラメータリード	65007	65207
ネットワークパラメータライト	65008	65208
停止指令	65009	65209
運転指令	65010	65210
プロファイルリード	65011	65211
ログデータリード	65013	65213
ログデータクリア	65014	65214
メッセージ折り返し	65015	65215
ベンダ固有	65016	65216
IO 割付設定	65018	65218
IO 割付読出し	65019	65219
トークン保持時間測定開始	65020	65220
トークン保持時間測定停止	65021	65221
汎用通信データ送信元ログ測定開始	65022	65222
汎用通信データ送信元ログ測定停止	65023	65223
コンフィギュレーション用パラメータ設定	65024	65224
参加ノード管理情報パラメータ読出し	65025	65225
自ノード管理情報パラメータ読出し	65026	65226
自ノード設定情報パラメータ読出し	65027	65227
ノードリセット	65028	65228

付録 2　システムエラー詳細コード

「***DLL***」内部でシステムエラーを検知した場合、*Error* イベントの ErrorCode（エラー種別）引数に 1 が通知されます。SpecCode（詳細コード）引数に通知される値は、GetLastError 関数の戻り値です。内容につきましては、開発コンテナのオンラインヘルプ等でご確認ください。

付録３　ネットワークエラー詳細コード

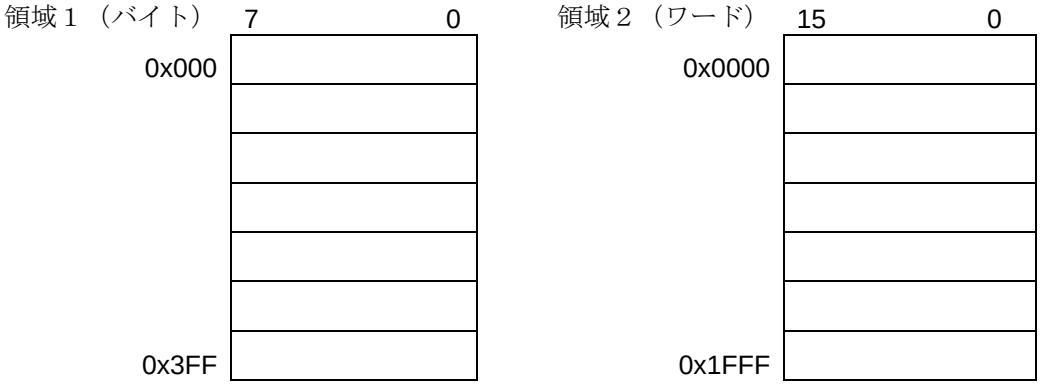
「***DLL***」内部でネットワークエラーを検知した場合、*Error* イベントの ErrorCode（エラー種別）引数に 2 が通知されます。SpecCode（詳細コード）引数に通知される値は、WSAGetLastError 関数の戻り値です。主なエラーコードに対する意味および対応方法を、以下に示します。

詳細コード	別名定義	意味
10024	WSAEMFILE	使用可能なリソースが不足しています。(接続しているコネクションが多い等) →他のプログラムを終了する、他のコネクションを切断する等の後再実行してください。
10042	WSAENOPROTOOPT	ソケットオプションの指定が不正。 →WindowsNT に付属の TCP/IP 以外の TCP/IP システムが組み込まれている可能性もあります。組み込まれている TCP/IP を確認してください。
10048	WSAEADDRINUSE	指定したアドレスは既に利用されています。 それ以前に使用していたコネクションが正常に終了できなかった場合にも発生することがあります。 同じアドレス、ポートを使う接続を連続して実行している場合にも発生することがあります。時間をおいて実行してください。
10049	WSAEADDRNOTAVAIL	指定したアドレスは無効です。 →指定されたアドレスが間違っています。アドレスの値を確認して再実行してください。
10050	WSAENETDOWN	ネットワークシステムに障害を検出しました。 → TCP/IP プロトコルの設定を確認してください。
10055	WSAENOBUFS	システムに使用可能な空きバッファが不足しています。 接続コネクション数や実行プログラムが多い場合にもエラーになることがあります。 →他のプログラムを終了する、他のコネクションを切断する等の後再実行してください。
10058	WSAESHUTDOWN	既に、システムがシャットダウンしているため、送信または受信ができない状態になっています。
10091	WSASYSNOTREADY	ネットワークサブシステムが利用できません。 →TCP/IP プロトコルが正しく組み込まれているか確認してください。
10092	WSAVERNOTSUPPORTED	要求されたバージョンをサポートしていません。 →WindowsNT の TCP/IP 以外の TCP/IP システムが組み込まれていないか確認してください。
11001	WSA_HOST_NOT_FOUND	ホスト名称が正しくありません。 →指定されたアドレスが間違っています。アドレスの値を確認して再実行してください。

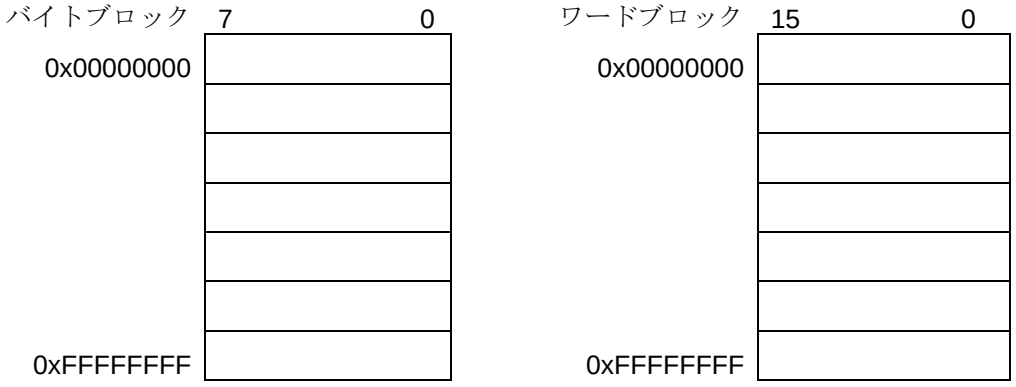
付録4　メモリマップ

コモンメモリとブロックデータ（共にバイトエリア及びワードエリア）のメモリマップを以下に示します。なお、「***DLL***」はコモンメモリの実データを内部メモリにて管理しますが、ブロックデータの実データは管理しません。ブロックデータを使用する場合は、「***APP***」にて実データを管理する必要があります。

1）コモンメモリ



2）ブロックデータ



1 回のリード／ライトで指定できるサイズは、バイトブロックの場合 1024 バイト、ワードブロックの場合 512 ワードとなります。

付録 5　プロファイル情報

「***KE-SFL3WIN***」には、以下に示すプロファイル情報（固定値）が設定されています。「***APP***」がプロファイル情報を変更することはできません。他ノードから自ノードあてにプロファイルリード要求メッセージを受信した場合は、以下の値で自動的にプロファイルリード応答を送信します。

32bit 版

パラメータ名称	名称文字 （長さ），（文字）	データタイプ[型]	パラメータ内容 （長さ），（内容）
デバイスプロファイル 共通仕様バージョン	6, “COMVER”	INTEGER	1, 1
システムパラメータ 識別文字	2, “ID”	PrintableString	7, “SYSPARA”
システムパラメータ 改変番号	3, “REV”	INTEGER	1, 1
システムパラメータ 変更日付	7, “REVDATE”	[INTEGER] [INTEGER] [INTEGER]	2, 2011 1, 11 1, 01
デバイス種別	10, “DVCATEGORY”	PrintableString	8, “COMPUTER”
ベンダ名	6, “VENDOR”	PrintableString	10, “HITACHI-KE”
製品形名	7, “DVMODEL”	PrintableString	10, “KE-SFL3WIN”

64bit 版

パラメータ名称	名称文字 （長さ），（文字）	データタイプ[型]	パラメータ内容 （長さ），（内容）
デバイスプロファイル 共通仕様バージョン	6, "COMVER"	INTEGER	1, 1
システムパラメータ 識別文字	2, "ID"	PrintableString	7, "SYSPARA"
システムパラメータ 改変番号	3, "REV"	INTEGER	1, 1
システムパラメータ 変更日付	7, "REVDATE"	[INTEGER] [INTEGER] [INTEGER]	2, 2018 1, 08 1, 01
デバイス種別	10, "DVCATEGORY"	PrintableString	8, "COMPUTER"
ベンダ名	6, "VENDOR"	PrintableString	10, "HITACHI-KE"
製品形名	7, "DVMODEL"	PrintableString	10, "KE-SFL3WIN"

付録 6　ログ情報

「***KE-SFL3WIN***」で実装するログ情報の一覧を以下に示します。（実装する項目のみ、項目名を記入します。項目名が空白の欄は、実装しません。）

「***DLL***」は、該当する項目のイベントが発生したタイミングで、値を+1 インクリメントします。値の範囲は、符号なしの長整数型（0～4,294,967,295）とし、範囲をオーバーした場合は、インクリメントを中止します。

ログ情報を読出す場合は、HFA_GetMyNodeLog 関数または HFA_GetMyNodeLogV3 関数をコールしてください。また、他ノードから自ノードあてにログデータのリード要求メッセージを受信した場合、以下の形式で応答します。（各項目は、それぞれ 4 バイトとし、4×128=512 バイトとなります。）

なお、ログ情報は以下のタイミングでクリアされます。

① 「***DLL***」起動時。

② 他ノードからログデータのクリア要求を受信した場合。

③ HFA_ClearMyNodeLog 関数がコールされた場合。

ｵﾌｾｯﾄ	項 目
0	通算ソケット部送信回数
4	通算ソケット部送信エラー回数
8	
12	
16	
20	
24	通算ソケット部受信回数
28	通算ソケット部受信エラー回数
32	
36	
40	
44	
48	トークン送信回数
52	サイクリックフレーム送信回数
56	1 対 1 メッセージ送信回数
60	1 対 n メッセージ送信回数
64	
68	
72	トークン受信回数
76	サイクリックフレーム受信回数
80	1 対 1 メッセージ受信回数
84	1 対 n メッセージ受信回数

ｵﾌｾｯﾄ	項 目
88	
92	
96	サイクリック伝送受信エラー回数
100	サイクリックアドレスサイズエラー回数
104	サイクリック CBN エラー回数
108	サイクリック TBN エラー回数
112	サイクリック BSIZE エラー回数
116	サイクリック伝送受信エラー検出時間
120	
124	
128	
132	
136	
140	
144	メッセージ伝送再送回数
148	メッセージ伝送再送オーバ回数
152	リフレッシュサイクル最大値検出時の時間
156	
160	
164	
168	メッセージ伝送受信エラー回数
172	メッセージ通番バージョンエラー回数

（↓次頁に続く。）

ｵﾌｾｯﾄ	項 目
176	メッセージ通番再送認識回数
180	メッセージ伝送受信エラー検出時間
184	
188	
192	ACK エラー回数
196	ACK 通番バージョンエラー回数
200	ACK 通番番号エラー回数
204	
208	ACK TCD エラー回数
212	
216	
220	
224	
228	
232	
236	
240	トークン多重化認識回数
244	トークン破棄回数
248	トークン再発行回数
252	トークン破棄検出直近の時間
256	トークン再発行直近の時間
260	トークン保持タイムアウト直近の時間
264	トークン保持タイムアウト回数
268	トークン監視タイムアウト回数
272	トークン監視タイムアウト直近の時間
276	トークン保持時間最大値
280	トークン保持時間最小値
284	トークン保持時間最大値検出時間
288	通算稼働時間 [s]
292	フレーム待ち状態回数
296	加入回数
300	自己離脱回数
304	スキップによる離脱回数
308	他ノード離脱認識回数
312	トークン保持時間測定時間

ｵﾌｾｯﾄ	項 目
316	トークン保持時間測定中のトークン回数
320	
324	
328	
332	汎用通信データ送信元ログ測定時間
336	参加認識ノード一覧（1～32）
340	参加認識ノード一覧（33～64）
344	参加認識ノード一覧（65～96）
348	参加認識ノード一覧（97～128）
352	参加認識ノード一覧（129～160）
356	参加認識ノード一覧（161～192）
360	参加認識ノード一覧（193～224）
364	参加認識ノード一覧（225～254）
368	IP1
372	IP1 受信カウンタ
376	IP2
380	IP2 受信カウンタ
384	IP3
388	IP3 受信カウンタ
392	IP4
396	IP4 受信カウンタ
400	IP5
404	IP5 受信カウンタ
408	IP6
412	IP6 受信カウンタ
416	IP7
420	IP7 受信カウンタ
424	IP8
428	IP8 受信カウンタ
432	IP9
436	IP9 受信カウンタ
440	IP10
444	IP10 受信カウンタ
448	
452	

（↓次頁に続く。）

ｵﾌｾｯﾄ	項 目
456	
460	
464	
468	
472	
476	
480	

ｵﾌｾｯﾄ	項 目
484	
488	
492	
496	
500	
504	
508	

付録 7　モニタモード

「***DLL***」は、FL-net ネットワークには参加せずに、ネットワーク上で巡回しているサイクリックデータパケットをモニタリングする機能をもちます。この機能により、ネットワーク監視することをモニタモードと称します。モニタモードは、HFA_LinkIn 関数にて自ノード番号を 0 で設定することで実行されます。なお、モニタモードで参加する際の機能制限を以下の表に示します。（ここで、”○”は機能有りを示し、”×”は機能無しを示します。）

1） 関数呼び出しによる機能制限

関数名	機 能
HFA_LinkIn	○
HFA_LinkInDefault	○
HFA_LinkOut	○
HFA_WriteCommon	×
HFA_ReadCommon	○
HFA_GetNodeStatus	注 1
HFA_GetNetworkStatus	○
HFA_GetMyNodeLog	○
HFA_GetMyNodeLogV3	○
HFA_ClearMyNodeLog	○
HFA_SetCommon	×
HFA_GetCommon	○
HFA_SetNodeName	○
HFA_GetNodeName	○
HFA_SetNodeNo	×
HFA_GetNodeNo	○
HFA_SetTokenWatchTime	×
HFA_GetTokenWatchTime	○
HFA_SetMinFrameInterval	×
HFA_GetMinFrameInterval	○
HFA_SetIP	×
HFA_GetIP	○
HFA_SetConfigParam	×
HFA_GetConfigParam	○
HFA_SetCommonRefreshDegree	○
HFA_ReqReadByteBlock	×

関数名	機 能
HFA_ReqWriteByteBlock	×
HFA_ReqReadWordBlock	×
HFA_ReqWriteWordBlock	×
HFA_ReqReadNetParam	×
HFA_ReqWriteNetParam	×
HFA_ReqControlEquipment	×
HFA_ReqReadProfile	×
HFA_ReqReadLog	×
HFA_ReqClearLog	×
HFA_ReqEchoMessage	×
HFA_ReqVendorMessage	×
HFA_RepReadByteBlock	×
HFA_RepWriteByteBlock	×
HFA_RepReadWordBlock	×
HFA_RepWriteWordBlock	×
HFA_RepWriteNetParam	×
HFA_RepControlEquipment	×
HFA_RepReadProfile	×
HFA_RepClearLog	×
HFA_RepVendorMessage	×
HFA_SendTranparency	×
HFA_AttachLink	○
HFA_DetachLink	○
HFA_SetCallback	○
HFA_SetCallbackV3	○
HFA_SetTimeout	×

(次頁へ続く)

注 1） 取得可能な項目は、HFA_GetNodeStatus 関数の詳細をご参照ください。

(続き)

関数名	機 能
HFA_DebugLog	○
HFA_StartCmdServerUdp	○
HFA_StopCmdServerUdp	○
HFA_StartCmdServerTcp	○
HFA_StopCmdServerTcp	○
HFA_StartTokenTimeMeasure	○
HFA_StopTokenTimeMeasure	○
HFA_GetTokenTimeMeasureStatus	○
HFA_StartDataLogMeasure	○
HFA_StopDataLogMeasure	○
HFA_GetDataLogMeasureStatus	○
HFA_SetIO	×
HFA_GetIO	○
HFA_GetSlaveNodeLinkStatus	○
HFA_SetOutputStatus	○
HFA_GetOutputStatus	○
HFA_GetInputStatus	○

関数名	機 能
HFA_ReadInputData	○
HFA_ReadInputBitData	○
HFA_ReadInputWordData	○
HFA_ReadInputRandomBitData	○
HFA_ReadInputRandomWordData	○
HFA_ReadOutputData	○
HFA_ReadOutputBitData	○
HFA_ReadOutputWordData	○
HFA_ReadOutputRandomBitData	○
HFA_ReadOutputRandomWordData	○
HFA_WriteOutputBitData	○
HFA_WriteOutputWordData	○
HFA_WriteOutputRandomBitData	○
HFA_WriteOutputRandomWordData	○

2） コールバック関数によるイベント発生有無

コールバック関数	機 能
LinkIn	注 2
LinkOut	注 2
CommonRefresh	○
LogClear	○
RecvReqReadByteBlock	×
RecvReqWriteByteBlock	×
RecvReqReadWordBlock	×
RecvReqWriteWordBlock	×
RecvReqWriteNetParam	×
RecvReqControlEquipment	×
RecvReqReadProfile	×
RecvReqClearLog	×
RecvReqVendorMessage	×
RecvRepReadByteBlock	×
RecvRepWriteByteBlock	×
RecvRepReadWordBlock	×
RecvRepWriteWordBlock	×
RecvRepReadNetParam	×
RecvRepWriteNetParam	×
RecvRepControlEquipment	×
RecvRepReadProfile	×
RecvRepReadLog	×
RecvRepClearLog	×
RecvRepEchoMessage	×
RecvRepVendorMessage	×
RecvTransparency	×

コールバック関数	機 能
LinkInTimeout	×
SendTimeout	×
ReplyTimeout	×
SendComplete	×
Error	○
LinkInSlave	×
LinkOutSlave	×
InputDataRefresh	×
InputStatusRefresh	×
ChangeTokenTimeMeasureStatus	○
ChangeDataLogMeasureStatus	○
RecvReqReadByteBlockFromSettingTool	○
RecvReqReadWordBlockFromSettingTool	○
RecvReqWriteByteBlockFromSettingTool	○
RecvReqWriteWordBlockFromSettingTool	○
RecvReqWriteNetParamFromSettingTool	○
RecvReqControlEquipmentFromSettingTool	○
RecvReqSetIOFromSettingTool	○
RecvReqSetConfigParamFromSettingTool	○
RecvReqResetNodeFromSettingTool	○

注 2）他ノードのリンク参加／離脱のみ、コールバック通知されます。