

# メモリマネージャ

## ゲーム特化した3つのヒープ

2008-05-30

任天堂株式会社発行

このドキュメントの内容は、機密情報であるため、**厳重な取り扱い、管理を行ってください。**

# 目次

1	はじめに .....	6
2	拡張ヒープマネージャ .....	6
2.1	ヒープの作成 .....	6
2.2	メモリブロックの確保 .....	6
2.2.1	メモリブロックの確保と解放 .....	6
2.2.2	メモリブロックを確保する場合の最小単位 .....	7
2.2.3	メモリブロック確保の手法 .....	7
2.2.4	ヒープ領域後方からのメモリブロックの確保 .....	8
2.3	アライメントの指定 .....	9
2.4	メモリブロックのサイズ変更 .....	9
2.5	空き容量の取得 .....	10
2.6	グループID .....	10
2.7	メモリブロックに対する処理 .....	11
2.8	メモリブロックの情報の取得 .....	11
2.9	ヒープとメモリブロックのチェック .....	11
3	フレームヒープマネージャ .....	12
3.1	ヒープの作成 .....	12
3.2	メモリブロックの確保 .....	12
3.2.1	メモリブロックの確保と解放 .....	12
3.2.2	メモリブロックを確保する場合の最小単位 .....	13
3.3	アライメントの指定 .....	13
3.4	メモリブロックの解放 .....	13
3.5	メモリブロック確保状態の保存と復帰 .....	14
3.6	ヒープ領域の大きさの調整 .....	15
3.7	メモリブロックのサイズ変更 .....	15
3.8	確保可能なサイズの取得 .....	16
4	ユニットヒープ .....	17
4.1	ヒープの作成 .....	17
4.2	メモリブロックの確保 .....	17
4.2.1	メモリブロックの確保と解放 .....	17
4.2.2	メモリブロックを確保する場合の最小単位 .....	18
4.3	アライメントの指定 .....	18
4.4	確保可能なメモリブロック数の取得 .....	18
5	各ヒープで共通する機能 .....	19
5.1	ヒープへのオプション .....	19
5.2	デバッグ時に埋める値の変更 .....	19
5.3	ヒープの内部の表示 .....	20

5.4	ヒープ領域の取得 .....	20
6	マルチヒープの管理 .....	21
6.1	マルチヒープ .....	21
6.2	マルチヒープでのメモリ解放 .....	21
6.3	ヒープをツリー状に管理 .....	21

## 表

表 2-1	拡張ヒープの作成と破棄の関数 .....	6
表 2-2	メモリブロックの確保と解放を行う関数 .....	7
表 2-3	メモリブロック確保モード .....	7
表 2-4	確保モードの設定と取得の関数 .....	8
表 2-5	関数で用いる確保モードの種類 .....	8
表 2-6	メモリブロックのサイズ変更を行う関数 .....	10
表 2-7	空き容量などを取得する関数 .....	10
表 2-8	グループIDの設定と取得を行う関数 .....	10
表 2-9	メモリブロックの情報を取得する関数 .....	11
表 2-10	拡張ヒープとメモリブロックのチェックを行う関数 .....	11
表 3-1	ヒープの作成と破棄を行う関数 .....	12
表 3-2	メモリブロックの確保の関数 .....	13
表 3-3	メモリブロックの解放方法 .....	13
表 3-4	メモリブロックを解放する関数 .....	13
表 3-5	関数に指定するメモリブロック解放方法の値 .....	14
表 3-6	メモリブロックの確保状態の保存と復帰を行う関数 .....	15
表 3-7	ヒープ領域の大きさを縮小する関数 .....	15
表 3-8	メモリブロックのサイズ変更を行う関数 .....	16
表 3-9	確保可能なサイズを取得する関数 .....	16
表 4-1	ヒープの作成と破棄を行う関数 .....	17
表 4-2	メモリブロックの確保と解放を行う関数 .....	18
表 4-3	確保可能なメモリブロックの個数を取得する関数 .....	19
表 5-1	ヒープ作成時に指定できるオプション .....	19
表 5-2	デバッグ時に埋める値の設定と取得を行う関数 .....	20
表 5-3	値を埋めるときのヒープの操作の種類 .....	20
表 5-4	ヒープ内部の情報を表示する関数 .....	20
表 5-5	ヒープ領域を取得する関数 .....	20
表 6-1	メモリブロックを確保したヒープを検索する関数 .....	21

## 図

図 2-1	拡張ヒープのメモリブロック確保の手法 .....	7
図 2-2	メモリブロック細分化のメカニズム .....	8
図 2-3	拡張ヒープのメモリブロック細分化の対策 .....	9
図 2-4	拡張ヒープのメモリブロックのサイズ変更 .....	9
図 3-1	フレームヒープのメモリ確保 .....	13
図 3-2	フレームヒープのメモリブロック確保状態の保存と復帰メカニズム .....	14

図 3-3	フレームヒープの大きさの調整 .....	15
図 3-4	フレームヒープのメモリブロックのサイズ変更.....	16
図 4-1	ユニットヒープのメモリ確保 .....	18

## 改訂履歴

改訂日	改訂内容
2008-05-30	NITRO-Systemの名称変更による修正(NITRO-SystemからTWL-Systemに変更)。
2008-04-08	改訂履歴の書式を変更。
2007-06-15	1. 脱字の修正。
2005-01-05	1. NITROと言う表記をニンテンドーDSに統一。
2004-08-20	1. 拡張ヒープに「ヒープとメモリブロックのチェック」を追加
2004-08-02	1. フレームヒープに「メモリブロックのサイズ変更」を追加
2004-06-10	1. P17「ヒープへのオプション」の説明文を修正。
2004-04-12	1. 誤字の修正。
2004-03-30	1. タイトル、ヘッダーの変更。 2. 文全体のわかりにくい個所に説明を追加。 3. 誤字・脱字の修正
2004-03-29	1. 文全体の誤字・脱字の修正。 2. 「はじめに」の本文に2箇所削除。 3. 各ヒープの機能の説明の修正。 4. 「各ヒープで共通する機能」を追加。
2004-03-25	1. 拡張ヒープにAPIの説明を追加。 2. 拡張ヒープに「ヒープの作成」を追加。 3. 拡張ヒープの「メモリブロック確保の手法」の修正。 4. 拡張ヒープの「アライメントの指定」の修正。 5. 拡張ヒープの「特殊なメモリ解放」の削除。 6. 拡張ヒープに「空き容量の取得」を追加。 7. 拡張ヒープの「グループID」の値の変更。 8. 拡張ヒープに「メモリブロックの情報の取得」を追加。 9. フレームヒープに「ヒープの作成」を追加。 10. フレームヒープの「アライメントの処理」の修正。 11. フレームヒープに「確保可能なサイズの取得」を追加。 12. ユニットヒープに「ヒープの作成」を追加。 13. ユニットヒープの「メモリブロックの確保」の修正。 14. ユニットヒープに「メモリブロックを確保する場合の最小単位」を追加。 15. ユニットヒープに「アライメントの指定」を追加。 16. ユニットヒープに「確保可能なメモリブロック数の取得」を追加。
2004-02-06	1. 拡張ヒープの「グループIDによる解放」を削除。 2. 拡張ヒープに「メモリブロックに対する処理」を追加。 3. ユニットヒープのアルゴリズムを変更。

# 1 はじめに

TWL では 16Mバイトの大きさを持つメインメモリが存在します。また、ニンテンドーDS でも4Mバイトの大きさを持つメインメモリが存在します。これくらいの大きさのメモリとなりますと、プログラマがメモリマップを作成して管理すると言う事は、かなり困難な作業となります。このような場合、メモリマネージャを使用することで、動的にメモリの確保と解放が行えるようになり、メモリマップ等によるメモリ管理を行う必要はなくなります。

しかしながら、パソコンなどと比較すると、TWL やニンテンドーDS のメモリサイズはかなり小さいうえに、ゲーム機ならではの特殊な事情などがあり、一般的な `malloc()` と `free()` だけでは非力であると考えます。そこで、TWL-System では少し特殊なメモリマネージャを3つ提供します。これらは、ゲーム等でよく使われるヒープ機構に独自のアイデアを付加した物となっています。

## 2 拡張ヒープマネージャ

拡張ヒープマネージャは、C言語標準ライブラリの `malloc()` 関数や `free()` 関数と同様に、メモリの確保と解放を自由に行う事が出来るメモリマネージャです。メモリ確保と解放と言う基本機能に加え、ゲームソフトで使用することを考慮して、幾つかの付加機能を持っています。以下に拡張ヒープマネージャの概要を説明します。

### 2.1 ヒープの作成

拡張ヒープマネージャを利用するためには、まず拡張ヒープを作成する必要があります。拡張ヒープの作成と破棄を行う関数は次のとおりです。

**表 2-1 拡張ヒープの作成と破棄の関数**

関数	機能
<code>NNS_FndCreateExpHeap()</code>	拡張ヒープを作成します。
<code>NNS_FndCreateExpHeapEx()</code>	拡張ヒープを作成します。ヒープへのオプションが指定出来ます。
<code>NNS_FndDestroyExpHeap()</code>	拡張ヒープを破棄します。

### 2.2 メモリブロックの確保

#### 2.2.1 メモリブロックの確保と解放

メモリブロックの確保と解放を行う関数を次に示します。

表 2-2 メモリブロックの確保と解放を行う関数

関数	機能
NNS_FndAllocFromExpHeap()	拡張ヒープからメモリブロックを確保します。
NNS_FndAllocFromExpHeapEx()	拡張ヒープからメモリブロックを確保します。アライメント(次節で説明)を指定できます。
NNS_FndFreeToExpHeap()	メモリブロックを解放します。

### 2.2.2 メモリブロックを確保する場合の最小単位

拡張ヒープマネージャでは、メモリブロックの管理領域として16バイトを必要とします。また、確保するメモリブロックは、最低で4バイトの境界にアライメントされますので、1バイトのメモリブロックを確保する場合でも20バイトのメモリが消費されることになります。

### 2.2.3 メモリブロック確保の手法

拡張ヒープマネージャでは、メモリブロックを確保する空き領域を検索する方法に 2 種類のモードがあり、切り替えて使用できるようになっています。確保モードは次のとおりです。

表 2-3 メモリブロック確保モード

モード	内容
FIRST モード	確保しようとしているメモリブロックのサイズ以上の大きさを持つ、最初に見つかった空き領域からメモリブロックを確保します。
NEAR モード	確保しようとしているメモリブロックのサイズに一番近いサイズの空き領域を探し、その空き領域からメモリブロックを確保します。

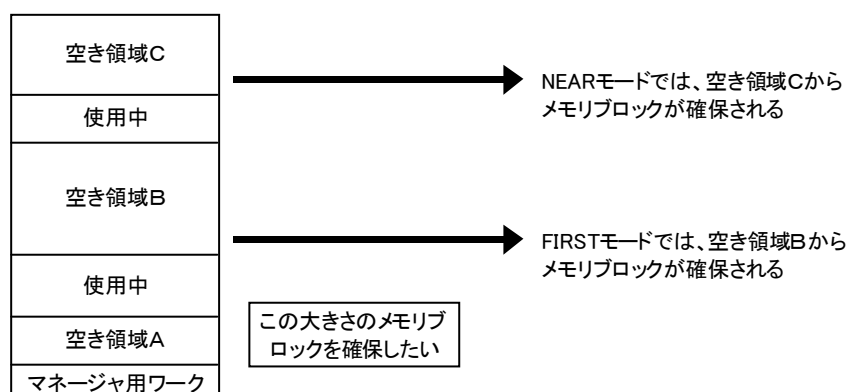


図 2-1 拡張ヒープのメモリブロック確保の手法

デフォルトでは、FIRST モード になっています。NEAR モードの方が FIRST モードに比べて、サイズの大きい空きブロックをなるべくそのまま温存するという特性があります。ただしこのモードでは、ピッタリとフィットする空き領域が見つからないかぎり、全ての空き領域を検索しますので、空き領域が細分化している場合には、メモリブロックの確保に時間

がかかってしまいます。

確保モードの設定と取得を行う関数は次のとおりです。

**表 2-4 確保モードの設定と取得の関数**

関数	機能
NNS_FndSetAllocModeForExpHeap()	確保モードを設定します。
NNS_FndGetAllocModeForExpHeap()	現在設定されている確保モードを取得します。

関数で用いる確保モードの種類は次のとおりです。

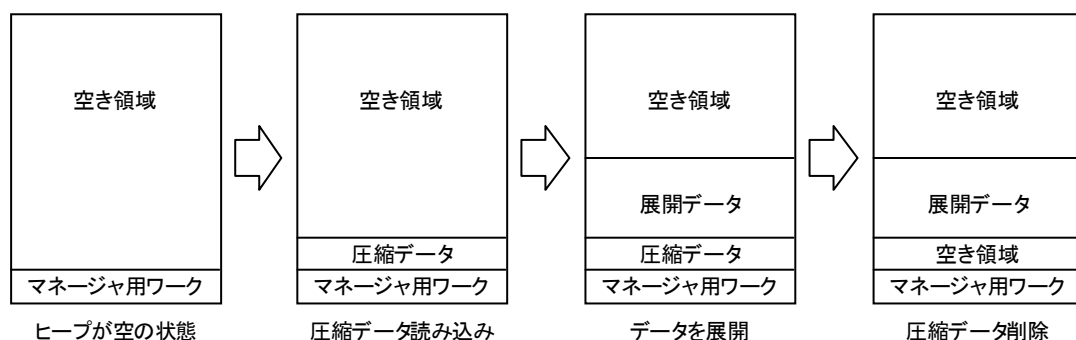
**表 2-5 関数で用いる確保モードの種類**

モード	関数で指定する値
FIRST モード	NNS_FND_EXPHEAP_ALLOC_MODE_FIRST
NEAR モード	NNS_FND_EXPHEAP_ALLOC_MODE_NEAR

## 2.2.4 ヒープ領域後方からのメモリブロックの確保

通常、拡張ヒープマネージャは、ヒープ領域の先頭から末尾に向かって空き領域を探し、見つかった空き領域の前方からメモリブロックを確保します。これとは逆に、ヒープ領域の末尾から先頭に向かって空き領域を探し、見つかった空き領域の後方からメモリブロックを確保することもできるようになっています。この機能は、寿命の長いメモリブロックは通常通りヒープ領域の前方から確保して、一時的なメモリブロックの場合はヒープ領域の後ろから確保するというように使い分けることで、ヒープのフラグメンテーション(細分化)を起こしにくくさせる事が出来るように考慮したものです。

例えば、圧縮されたデータをメモリ上に読み込んで、そのデータを展開した後に元の圧縮されたデータを削除する場合を考えてみます。この場合、通常通りにヒープ領域の前方からメモリブロックを確保して展開処理を行うと、空き領域が2つに分断されてしまいます。



**図 2-2 メモリブロック細分化のメカニズム**

そこで、一時的に読み込まれる圧縮データを、ヒープ領域の後ろから確保したメモリブロックに読み込むようにすると、空き領域が分断されなくなります。



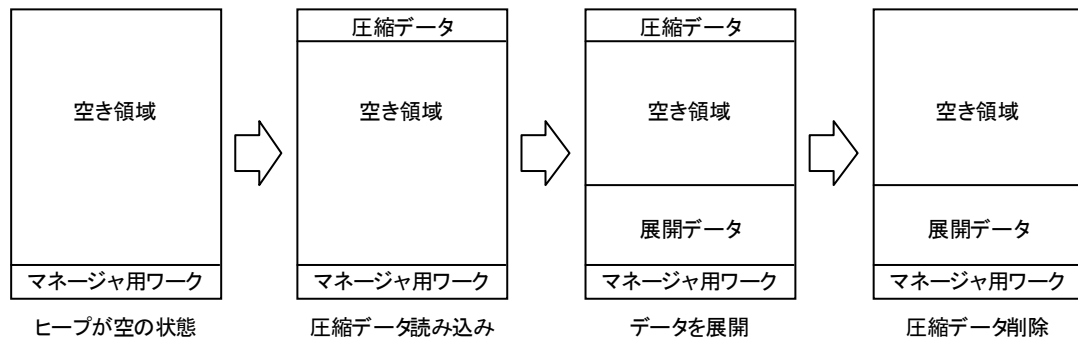


図 2-3 拡張ヒープのメモリブロック細分化の対策

メモリブロックをヒープ領域の後方から確保するには、メモリブロック確保関数 `NNS_FndAllocFromExpHeapEx0` を使用し、引数のアライメント(次節で説明)を負の値で与えるようにします。

## 2.3 アライメントの指定

拡張ヒープマネージャでは、メモリブロック確保時にアライメントを指定することができます。関数 `NNS_FndAllocFromExpHeapEx0` では、アライメントに 4,8,16,32 の値が指定できます。また、アライメントを負の値 (-4,-8,-16,-32) で指定すると、ヒープ領域の後方からメモリブロックを確保します。関数 `NNS_FndAllocFromExpHeap0` では、アライメントの指定は行えず、常に4になります。

## 2.4 メモリブロックのサイズ変更

拡張ヒープマネージャでは、確保したメモリブロックを移動させることなく、そのサイズを変更する事ができます。メモリブロックが今の大きさに比べて小さくなる場合には、縮小後に残る領域を空き領域として利用します。メモリブロックが今の大きさに比べて大きくなる場合には、メモリブロックの後ろに空き領域が存在する必要があります。メモリブロックの後ろが空き領域であれば、この空き領域を結合しサイズを大きくします。

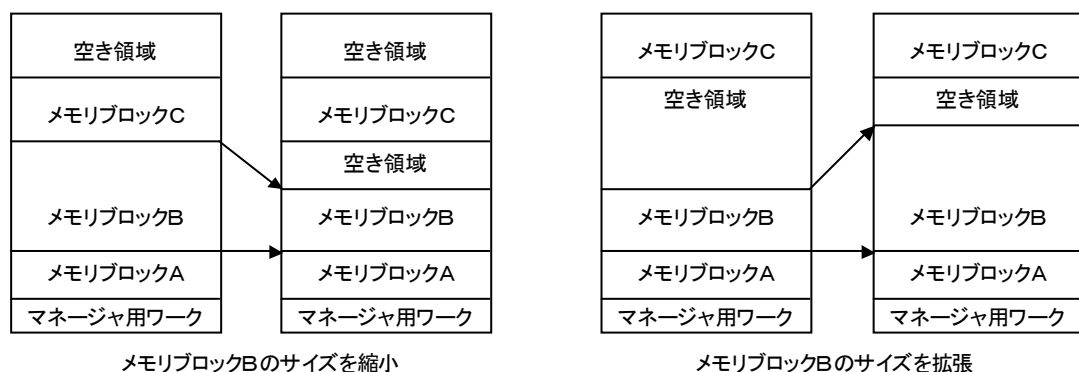


図 2-4 拡張ヒープのメモリブロックのサイズ変更

メモリブロックのサイズ変更で使用する関数は次のとおりです。

表 2-6      メモリブロックのサイズ変更を行う関数

関数	機能
NNS_FndResizeForMBlockExpHeap()	メモリブロックを拡張あるいは縮小します。戻り値として、変更されたメモリブロックのサイズが返ります。

メモリブロックを縮小する時、指定されたメモリブロックのサイズと現状のメモリブロックのサイズの差が小さい場合には、縮小してできる空き領域を有効に利用できない場合があります。この場合は、メモリブロックのサイズの変更は行われず、関数 NNS\_FndResizeForMBlockExpHeap() は現在のメモリブロックのサイズを返します。メモリブロックを拡張しようとして、メモリブロックの直後に空き領域が無かったり、空き領域を結合しても要求したサイズに拡張できない場合は、関数 NNS\_FndResizeForMBlockExpHeap() は失敗し 0 を返します。

## 2.5      空き容量の取得

拡張ヒープマネージャでは、空き領域の合計を取得することが出来ます。また、確保可能なもっとも大きなメモリブロックのサイズを求めることも出来ます。次にこれらの関数を示します。

表 2-7      空き容量などを取得する関数

関数	機能
NNS_FndGetTotalFreeSizeForExpHeap()	拡張ヒープ内の空き領域の合計サイズを取得します。
NNS_FndGetAllocatableSizeForExpHeap()	確保できるメモリブロックの最大サイズを取得します。アライメントは 4 固定です。
NNS_FndGetAllocatableSizeForExpHeapEx()	確保できるメモリブロックの最大サイズを取得します。アライメントを指定できます。

## 2.6      グループID

拡張ヒープマネージャは、メモリブロックを確保する時に、メモリブロックの管理領域内に 0～255 のグループID を格納します。このグループIDは、ユーザが任意に変更することができます。グループIDを変更すると、次のメモリブロックの確保から、新しく設定されたグループIDが使用されます。このグループIDは、以下のような用途に利用できます。

- 特定のグループIDを持ったメモリブロックだけを、一括して解放する。
- グループ ID ごとのメモリ使用量を調べる。用途や使用者ごとにグループ ID を管理することで、何にメモリを費やしているかを把握しやすくなる。

グループ ID の設定と取得を行う関数を示します。

表 2-8      グループ ID の設定と取得を行う関数

関数	機能
NNS_FndSetGroupIDForExpHeap()	拡張ヒープのグループ ID をセットします。
NNS_FndGetGroupIDForExpHeap()	拡張ヒープのグループ ID を取得します。

## 2.7 メモリブロックに対する処理

拡張ヒープマネージャでは、確保されているメモリブロックに対してユーザーが指定した処理を行わせる事が出来ます。この機能により、拡張ヒープマネージャでは用意されていない、ヒープに対する様々な処理を行う事が出来ます。以下に、その例を記します。

- 一時的な利用のためにヒープ領域の後方から確保したメモリブロックだけを、一括して解放する機能を実現する。
- 特定のグループIDを持つメモリブロックの容量の合計を求める。

指定した処理を行わせる関数は次のとおりです。

関数	機能
NNS_FndVisitAllocatedForExpHeap()	確保されたメモリブロックおのおのに対してユーザが指定した関数を呼び出します。

## 2.8 メモリブロックの情報の取得

拡張ヒープマネージャでは、確保されたメモリブロックに対しての情報、メモリブロックのサイズ、グループ ID、先頭から確保されたか、末尾から確保されたかを取得することが出来ます。メモリブロックの情報を取得する関数は次のとおりです。

表 2-9      メモリブロックの情報を取得する関数

関数	機能
NNS_FndGetSizeForMBlockExpHeap()	メモリブロックのサイズを取得します。
NNS_FndGetGroupIDForMBlockExpHeap()	メモリブロックのグループ ID を取得します。
NNS_FndGetAllocDirForMBlockExpHeap()	メモリブロックの確保方向を取得します。

## 2.9 ヒープとメモリブロックのチェック

拡張ヒープマネージャでは、拡張ヒープや拡張ヒープから確保されたメモリブロックが破壊されていないかどうかのチェックを行うことが出来ます。拡張ヒープとメモリブロックのチェックを行う関数は次のとおりです。

表 2-10      拡張ヒープとメモリブロックのチェックを行う関数

関数	機能
NNS_FndCheckExpHeap ()	拡張ヒープが破壊されていないかチェックします。
NNS_FndCheckForMBlockExpHeap ()	メモリブロックが破壊されていないかチェックします。

## 3 フレームヒープマネージャ

フレームヒープマネージャは、非常に単純なメモリマネージャで、指定された大きさのメモリブロックを確保することと、確保されたメモリブロックの全てを同時に解放することしかできません。その代わりに、メモリブロックに管理情報を一切持たないため、メモリ効率が良いものとなっています。以下にフレームヒープマネージャの概要を説明します。

### 3.1 ヒープの作成

フレームヒープマネージャを利用するためには、まずフレームヒープを作成する必要があります。フレームヒープの作成と破棄を行う関数は次のとおりです。

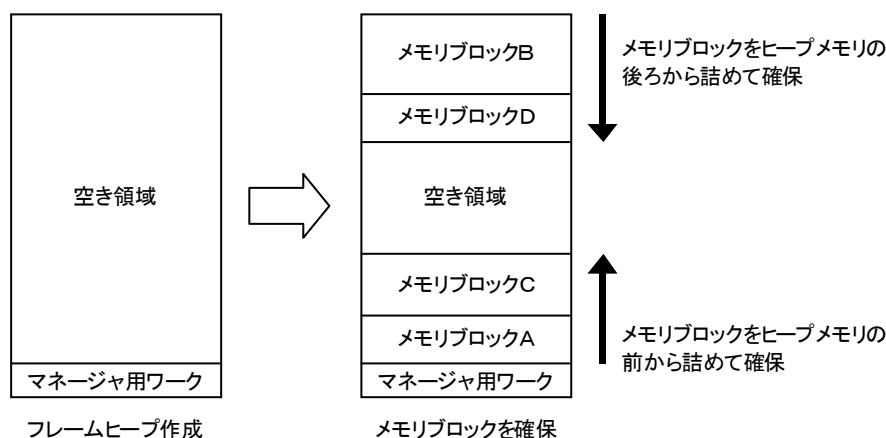
表 3-1 ヒープの作成と破棄を行う関数

関数	機能
NNS_FndCreateFrmHeap()	フレームヒープを作成します。
NNS_FndCreateFrmHeapEx()	フレームヒープを作成します。ヒープへのオプションが指定出来ます。
NNS_FndDestroyFrmHeap()	フレームヒープを破棄します。

### 3.2 メモリブロックの確保

#### 3.2.1 メモリブロックの確保と解放

フレームヒープマネージャは、ヒープ領域の前後からメモリブロックを隙間なく詰めて確保します。このような方法でメモリブロックを確保する為、ヒープのフラグメンテーション(細分化)は発生しません。また、フレームヒープマネージャが確保するメモリブロックには管理領域が全く無いため、メモリ使用効率は良く、メモリブロックの確保にかかる処理も軽くなっています。



### 図 3-1 フレームヒープのメモリ確保

メモリブロックを確保する関数は次のとおりです。

表 3-2 メモリブロックの確保の関数

関数	機能
NNS_FndAllocFromFrmHeap()	フレームヒープからメモリブロックを確保します。
NNS_FndAllocFromFrmHeapEx()	フレームヒープからメモリブロックを確保します。アライメント(次節で説明)を指定できます。

ヒープ領域の後ろからメモリを確保する場合は、関数 NNS\_FndAllocFromFrmHeapEx()を用いて、アライメントを負の値で指定します。

### 3.2.2 メモリブロックを確保する場合の最小単位

フレームヒープマネージャは、メモリブロックに管理領域がありませんが、確保するメモリブロックは最低で4バイトの境界にアライメントされますので、1バイトのメモリブロックを確保する場合でも4バイトのメモリが消されます。

## 3.3 アライメントの指定

フレームヒープマネージャでは、メモリブロック確保時にアライメントを指定することができます。関数 NNS\_FndAllocFromFrmHeapEx()では、アライメントに 4,8,16,32 の値が指定できます。また、アライメントを負の値 (-4,-8,-16,-32) で指定すると、ヒープ領域の後方からメモリブロックを確保します。関数 NNS\_FndAllocFromFrmHeap()では、アライメントの指定は行えず、常に4になります。

## 3.4 メモリブロックの解放

フレームヒープマネージャでは、確保した個々のメモリブロックを管理していない為、確保されたメモリブロックを個別に解放する事が出来ません。メモリブロックを解放する場合には次の3つの方法があります。

表 3-3 メモリブロックの解放方法

解放方法	内容
前方解放	ヒープ領域の前方から確保したメモリブロックを一括して解放します。
後方解放	ヒープ領域の後方から確保したメモリブロックを一括して解放します。
全解放	ヒープから確保した全てのメモリブロックを一括して解放します。

メモリブロックを解放する関数は次のとおりです。

表 3-4 メモリブロックを解放する関数

関数	機能
----	----

NNS_FndFreeToFrmHeap()	メモリブロックを指定された方法で一括して解放します。
------------------------	----------------------------

関数 NNS\_FndFreeToFrmHeap() に指定する解放方法は次のとおりです。

**表 3-5 関数に指定するメモリブロック解放方法の値**

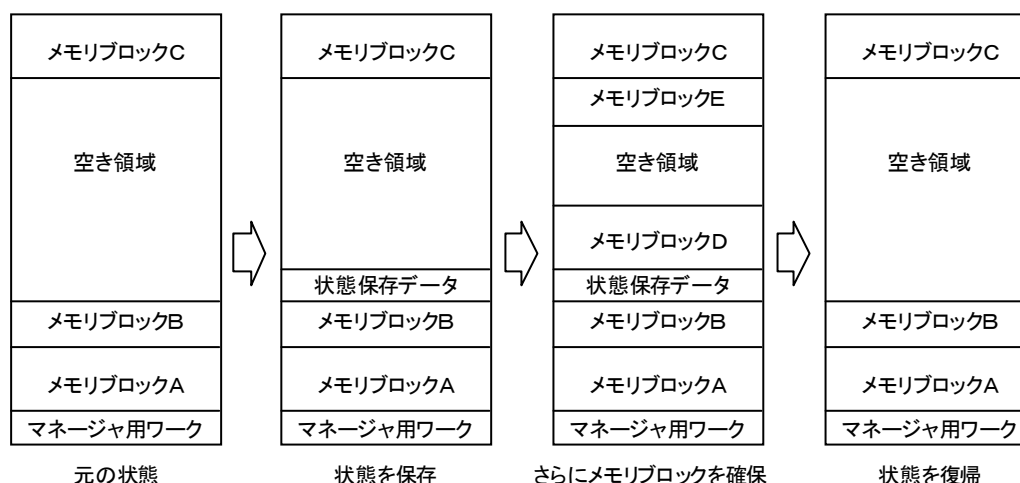
解放方法	関数に指定する値
前方解放	NNS_FND_FRMHEAP_FREE_HEAD
後方解放	NNS_FND_FRMHEAP_FREE_TAIL
全解放	NNS_FND_FRMHEAP_FREE_ALL
	(NNS_FND_FRMHEAP_FREE_HEAD と NNS_FND_FRMHEAP_FREE_TAIL を同時に指定した場合と同じです。)

また、メモリブロックの確保状態を保存しておき、以降に確保されたメモリブロックを一括して解放し、保存される直前の状態に戻す機能もあります。これについては次節で説明します。

### 3.5 メモリブロック確保状態の保存と復帰

フレームヒープマネージャには、ヒープ領域内のメモリブロックの確保状態を保存したり、後にその状態に復帰させたりする機能を持っています。

メモリブロック確保状態を1回保存するためには20バイトのメモリが必要です。メモリブロック確保状態の保存は、そのヒープの容量が許す限り何度でも保存できます。メモリブロック確保状態を保存する場合には、4バイトのタグを付けることが出来ます。メモリブロック確保状態を復帰させる場合には、1つ前の状態か、またはタグにより指定した状態に復帰させる事が出来ます。



**図 3-2 フレームヒープのメモリブロック確保状態の保存と復帰メカニズム**

メモリブロックの確保状態の保存と復帰を行う関数は次のとおりです。

表 3-6      メモリブロックの確保状態の保存と復帰を行う関数

関数	機能
NNS_FndRecordStateForFrmHeap()	メモリブロックの確保状態を保存します。
NNS_FndFreeByStateToFrmHeap()	メモリブロックの確保状態を復帰します。

### 3.6 ヒープ領域の大きさの調整

フレームヒープマネージャは、ヒープ領域の大きさをヒープ領域の内容に合わせて縮小することができます。ただし、この機能が使えるのはヒープ領域の後ろから確保されたメモリブロックが1つも無い場合に限られます。

この機能は、大きさや個数が不定のデータをメモリに隙間無く詰めこみたい場合に使用できます。まず、十分大きなサイズでフレームヒープを作成し、ヒープ領域の前方からメモリブロックを確保し、データを格納していきます。必要なデータを全て格納した後に、ヒープの内容に合わせて、ヒープ領域の大きさを縮小します。

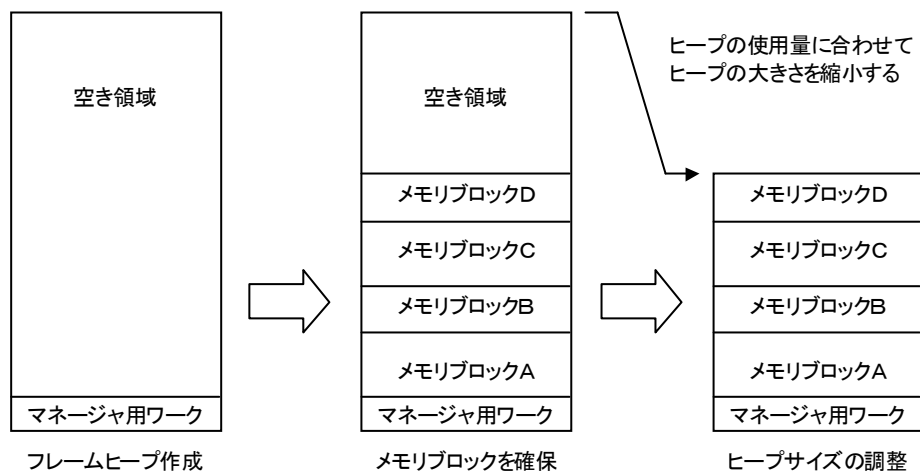


図 3-3      フレームヒープの大きさの調整

ヒープ領域の大きさを縮小する関数は次のとおりです。

表 3-7      ヒープ領域の大きさを縮小する関数

関数	機能
FndAdjustFrmHeap()	ヒープ領域の後方にある空き領域をヒープ領域から解放し、ヒープ領域の大きさを縮小します。

### 3.7 メモリブロックのサイズ変更

フレームヒープマネージャでは、ヒープの空き領域の前方から確保されたメモリブロックの中で最後に確保されたメモリブロックに限り、そのメモリブロックのサイズを変更する事ができます。メモリブロックが今の大きさに比べて小さくなる場合には、縮小後に残る領域は後方の空き領域に吸収されます。メモリブロックが今の大きさに比べて大きくなる場合に

は、後方の空き領域を縮小して、メモリブロックを拡張します。

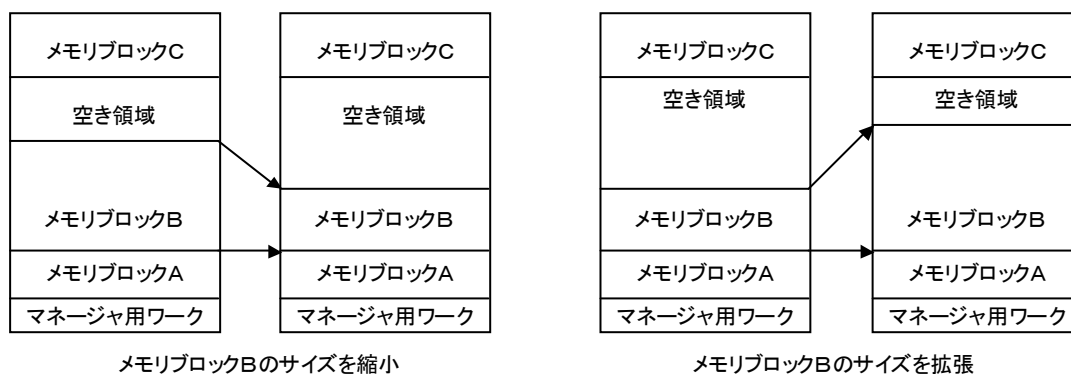


図 3-4 フレームヒープのメモリブロックのサイズ変更

メモリブロックのサイズ変更で使用する関数は次のとおりです。

表 3-8 メモリブロックのサイズ変更を行う関数

関数	機能
NNS_FndResizeForMBlockFrmHeap()	メモリブロックを拡張あるいは縮小します。戻り値として、変更されたメモリブロックのサイズが返ります。

メモリブロックを拡張する場合、要求したサイズに拡張するのに十分な空き領域が無い場合、関数 NNS\_FndResizeForMBlockExpHeap() は失敗し 0 を返します。

### 3.8 確保可能なサイズの取得

フレームヒープマネージャでは、確保可能なもっとも大きなメモリブロックのサイズを求めることができます。次にこれらの関数を示します。

表 3-9 確保可能なサイズを取得する関数

関数	機能
NNS_FndGetAllocatableSizeForFrmHeap()	確保できるメモリブロックの最大サイズを取得します。アライメントは 4 固定です。
NNS_FndGetAllocatableSizeForFrmHeapEx()	確保できるメモリブロックの最大サイズを取得します。アライメントを指定できます。



## 4 ユニットヒープ

ユニットヒープマネージャは非常に単純なメモリマネージャで、メモリブロックの確保はユニットヒープ作成時に指定したサイズのみとなります。即ち固定サイズのメモリブロックの確保と解放を行う為のメモリマネージャです。ユニットヒープは、メモリブロックに管理領域がありませんので、メモリの使用効率が良くなっています。以下にユニットヒープメモリマネージャの概要を説明します。

### 4.1 ヒープの作成

ユニットヒープマネージャを利用するためには、まずユニットヒープを作成する必要があります。次にユニットヒープの作成と破棄を行う関数を示します。

**表 4-1      ヒープの作成と破棄を行う関数**

関数	機能
NNS_FndCreateUnitHeap()	ユニットヒープを作成します。
NNS_FndCreateUnitHeapEx()	ユニットヒープを作成します。アライメントとヒープへのオプションが指定出来ます。
NNS_FndDestroyUnitHeap()	ユニットヒープを破棄します。

### 4.2 メモリブロックの確保

#### 4.2.1 メモリブロックの確保と解放

ユニットヒープマネージャでは、ヒープ領域内を予め指定された単位のチャンクに区切って管理しています。メモリブロックの確保は、このチャンクの確保ということになります。

空きチャンクは、片方向リスト(フリーチャンクリスト)として連結されています。次の空きチャンクへのポインタは、チャンクの先頭に置かれます。使用中のチャンクには、ポインタは置かれません。(管理領域はありません。)メモリブロックを確保する時には、この空きチャンクリストの先頭に繋がれているメモリブロックを返します。使用中のメモリブロックを解放する時には、そのメモリブロックをフリーチャンクリストの先頭に繋がめます。

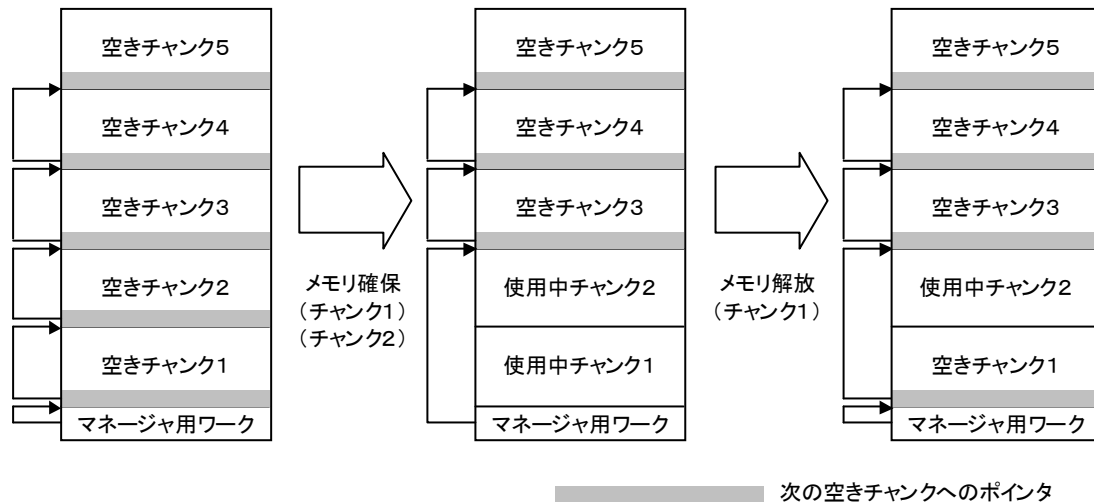


図 4-1 ユニットヒープのメモリ確保

メモリブロックの確保と解放を行う関数は次のとおりです。

表 4-2 メモリブロックの確保と解放を行う関数

関数	機能
NNS_FndAllocFromUnitHeap()	ユニットヒープからメモリブロックを確保します。
NNS_FndFreeToUnitHeap()	メモリブロックを解放します。

#### 4.2.2 メモリブロックを確保する場合の最小単位

ユニットヒープマネージャは、メモリブロックに管理領域がありませんが、確保するメモリブロックは最低で4バイトの境界にアライメントされますので、1バイトのメモリブロックを割り当てる場合でも4バイトのメモリが消費されます。

### 4.3 アライメントの指定

ユニットヒープマネージャでは、ヒープ作成時にアライメントを指定することができます。確保するメモリブロック単位毎には行えません。確保されるメモリブロックは全て同じアライメントになります。関数 NNS\_FndCreateUnitHeapEx ()では、アライメントに 4,8,16,32 の値が指定できます。関数 NNS\_FndCreateUnitHeap()では、アライメントの指定は行えず、常に4になります。

### 4.4 確保可能なメモリブロック数の取得

ユニットヒープマネージャでは、確保可能なメモリブロックの数、すなわち空きチャンクの数を取得することが出来ます。次に関数を示します。

表 4-3 確保可能なメモリブロックの個数を取得する関数

関数	機能
NNS_FndCountFreeBlockForUnitHeap()	確保可能なメモリブロックの個数を返します。

## 5 各ヒープで共通する機能

ここでは、拡張ヒープ、フレームヒープ、ユニットヒープそれぞれに共通の機能について説明します。

### 5.1 ヒープへのオプション

ヒープを作成する関数の中で、NNS\_FndCreateExpHeapEx()、NNS\_FndCreateFrmHeapEx()、NNS\_FndCreateUnitHeapEx()は、ヒープへのオプションを指定できます。指定できるオプションは次のとおりです。

表 5-1 ヒープ作成時に指定できるオプション

フラグ	内容
NNS_FND_HEAP_OPT_0_CLEAR	ヒープからメモリを確保したときに、確保したメモリブロックを 0 で埋めます。
NNS_FND_HEAP_OPT_DEBUG_FILL	ヒープ作成時および、メモリブロック確保時と解放時にそれぞれ異なった 32bit 値でメモリ領域を埋めます。

フラグ NNS\_FND\_HEAP\_OPT\_DEBUG\_FILL はデバッグ時に使用することを想定しており、メモリの初期化忘れや無効なメモリ領域を指しているポインタを通じてのメモリアクセスのバグを見つけるのに活用できます。最終 ROM 版 (FINALROM) ライブラリでは機能しないようになっています。

メモリ領域に埋める値は、デフォルトでは次のようになっていますが、変更することができます。変更の方法については次節で説明します。

- ヒープ作成時      0xC3C3C3C3
- メモリ確保時      0xF3F3F3F3
- メモリ解放時      0xD3D3D3D3

### 5.2 デバッグ時に埋める値の変更

ヒープ作成時にフラグ NNS\_FND\_HEAP\_OPT\_DEBUG\_FILL を指定することで、ヒープ作成時および、メモリブロック確保時と解放時にそれぞれ異なった 32bit 値でメモリ領域を埋めることができるようになりますが、この埋める値を変更することができます。埋める値の設定と取得を行う関数は次のとおりです。

表 5-2 デバッグ時に埋める値の設定と取得を行う関数

関数	機能
NNS_FndSetFillValForHeap()	埋める値を設定します。
NNS_FndGetFillValForHeap()	埋める値を取得します。

埋める値は、ヒープ作成時、メモリブロック確保時、メモリブロック解放時のそれぞれにおいて異なる値を設定できます。値の設定や取得を行うときは、どのヒープ操作時の値を対象とするかを指定します。次に関数に指定するヒープの操作の種類を示します。

表 5-3 値を埋めるときのヒープの操作の種類

関数に指定する値	ヒープの操作
NNS_FND_HEAP_FILL_NOUSE	ヒープ作成時
NNS_FND_HEAP_FILL_ALLOC	メモリブロック確保時
NNS_FND_HEAP_FILL_FREE	メモリブロック解放時

## 5.3 ヒープの内部の表示

デバッグ用として、ヒープ内部の情報を表示する機能があります。これらの事を行う関数を次に示します。

表 5-4 ヒープ内部の情報を表示する関数

関数	機能
NNS_FndDumpHeap	ヒープ内部の情報を表示します。

## 5.4 ヒープ領域の取得

ヒープが使用しているメモリ領域の開始アドレスと終了アドレスを取得する機能があります。これらの事を行う関数を次に示します。

表 5-5 ヒープ領域を取得する関数

関数	機能
NNS_FndGetHeapStartAddress()	ヒープの使用するメモリ領域の開始アドレスを取得します。
NNS_FndGetHeapEndAddress()	ヒープの使用するメモリ領域の終了アドレス(+1)を取得します。

## 6 マルチヒープの管理

ここでは、ゲームソフトウェアで複数個のヒープを作成し、利用する場合について考えています。

### 6.1 マルチヒープ

ゲーム中で使用されるデータには、グラフィックス用データや音楽用データ、システム用データなどの様に、色々な種類のデータがあります。これらのデータを管理しやすくする為に、ゲームヒープ、サウンドヒープ、システムヒープなどの様に、複数個のヒープを利用することがあります。このように複数のヒープを使用することを、マルチヒープと呼ぶことにします。

### 6.2 マルチヒープでのメモリ解放

ヒープからメモリブロックを確保したい場合、プログラマは、どのヒープからメモリブロックを確保すべきなのかは知っているはずで。よってプログラマは、あるヒープを指定して、そのヒープからメモリブロックを確保する事が出来ます。

メモリブロックを解放する場合はどうでしょうか。自分で確保したメモリブロックであれば、どのヒープに対してメモリブロックを返却すれば良いかは判ると思います。では、他のプログラマから受け取ったメモリブロックを解放する時は、どのヒープにメモリブロックを返却すれば良いのでしょうか。この場合でも、複数存在するヒープの用途がハッキリしている場合には、どのヒープに対してメモリブロックを返却すれば良いのかは判断がつくかも知れません。しかし、メモリブロックを返却すべきヒープの候補が複数考えられる時、どのようにすれば良いのでしょうか。

### 6.3 ヒープをツリー状に管理

どこから確保されたかが判らないメモリブロックを解放する場合、そのメモリブロックを確保したヒープを探す機構があれば便利です。これは、ヒープをツリー状に管理する事で可能となります。なぜツリー状なのかと言いますと、あるヒープから確保したメモリブロックをヒープ用のメモリとして利用する事が出来るからです(階層ヒープ構造)。

このように、ヒープがツリー状に管理されていますと、ヒープが占めるメモリの範囲を再帰的にチェックすることで、どのヒープから確保されたメモリブロックかを調べる事ができるようになります。

TWL-System のメモリマネージャでは、ヒープを作成するごとに内部でヒープの階層構造が作られています。そして、メモリブロックを確保したヒープを検索する関数が用意されています。次にこの関数を示します。

**表 6-1      メモリブロックを確保したヒープを検索する関数**

関数	機能
NNS_FndFindContainHeap()	指定されたメモリブロックを確保したヒープを検索し、見つかったヒープのハンドルを返します。

© 2004-2008 Nintendo

任天堂株式会社の許諾を得ることなく、本書に記載されている内容の一部あるいは全部を無断で複製・複写・転写・頒布・貸与することを禁じます。