応用記述

AT08569：ﾌﾗｯｼｭとRAM使用最小化のためのASFｺｰﾄﾞ量最適化

Atmel ﾏｲｸﾛ ｺﾝﾄﾛｰﾗ

要点

- ASF応用の大きさの考察
- ｺﾝﾊﾟｲﾗと構築任意選択
- ﾌﾗｯｼｭ必要量低減への検討
- RAM必要量低減への検討

序説

Atmel®ｿﾌﾄｳｪｱ枠組み(ASF:Atmel Software Framework)は広範囲なAtmelのﾏｲｸﾛ ｺﾝﾄﾛｰﾗで実行しｲﾝﾀｰﾌｪｰｽする応用ｺｰﾄﾞの迅速な開発を許す、ｿｰｽ ﾌｧｲﾙ、ﾗｲﾌﾞﾗﾘ、、参照基準ﾌﾟﾛｼﾞｪｸﾄの組です。

ASFは応用の作者が開発時間を殆ど中核機能の作成へ集中させ、応用を提供するのに必要な構築構成設定や周辺機能ﾄﾞﾗｲﾊﾞ枠組みを提供します。

主要なASF設計目標は統合されたﾏｲｸﾛ ｺﾝﾄﾛｰﾗ周辺機能の全特徴を実演することです。全機能が揃ったｻｰﾋﾞｽとﾄﾞﾗｲﾊﾞは複雑な応用必要条件にｿﾌﾄｳｪｱ必要条件の素早い一致を応用の作者に許します。それは度々ﾃﾞｰﾀｼｰﾄ調査の時間を減らし、難しいﾌﾟﾛｸﾞﾗﾑの問題の素早い試作を許します。

全機能が揃ったﾄﾞﾗｲﾊﾞの残念な副作用は、それらが特定機能に専用化されたものよりもより多くのﾒﾓﾘ資源を必要とすることです。総計では、最終的な応用と結合するこれらのﾄﾞﾗｲﾊﾞとｻｰﾋﾞｽは利用可能なﾒﾓﾘ資源を超えることができます。応用の構成を完成させることはｺｰﾄﾞをより小さな使用量に最適化するための多くの機会を与えます。ASFは多くの構造に渡って動く一般的な解決策で、Atmelﾏｲｸﾛ ｺﾝﾄﾛｰﾗが提供する特徴の殆どの使用法を例証するように書かれています。単一応用は代表的にこの機能の部分集合だけを必要とし、場所がきつくなる時には、Atmelｿﾌﾄｳｪｱ枠組み実装の調整(特徴省略)のために応用特有の必要条件を利用する幾つかの方法があります。

この応用記述はASF応用ﾒﾓﾘ必要条件を最小化するための様々な手法を調査します。

本書は一般の方々の便宜のため有志により作成されたもので、Atmel社とは無関係であることを御承知ください。しおりの[はじめに]での内容にご注意ください。

目次

1. ASFの大きさの考察 3
   1.1. ASFの特徴と大きさ 3
   1.2. ﾒﾓﾘ必要条件 3
      1.2.1. 静的ﾒﾓﾘ必要条件 3
      1.2.2. 動的ﾒﾓﾘ必要条件 5
         1.2.2.1. LwIP 6
         1.2.2.2. FreeRTOS 6
2. 道具でのASK応用ﾒﾓﾘ必要条件削減 7
   2.1. ﾏｲｸﾛ ｺﾝﾄﾛｰﾗ選択 7
   2.2. ﾂｰﾙﾁｪｰﾝ選択と調整 7
      2.2.1. ﾂｰﾙﾁｪｰﾝ ﾍﾞﾝﾁﾏｰｸ 8
      2.2.2. ﾂｰﾙﾁｪｰﾝ構築任意選択 8
      2.2.3. 組み込みCﾗｲﾌﾞﾗﾘ選択 9
3. 応用ﾌﾗｯｼｭ必要条件削減 9
   3.1. ASFｳｨｻﾞｰﾄﾞ部品除去 9
   3.2. 依存性追跡と機能調整 10
   3.3. ｺｰﾄﾞ単純化 11
   3.4. 例 12
4. 応用RAM必要条件削減 12
   4.1. ﾃﾞｰﾀ ｾｸﾞﾒﾝﾄと定数ﾃﾞｰﾀ 12
   4.2. ｽﾀｯｸ割り当て調整 12
   4.3. 先取り的 対 協調的の計画部 13
5. 改訂履歴 13

1. ASFの大きさの考察

望むｻｰﾋﾞｽやﾐﾄﾞﾙｳｪｱで応用を準備する時に、しばしば想像する大きさが選択されたﾏｲｸﾛ ｺﾝﾄﾛｰﾗで使用可能なﾒﾓﾘ資源を超える結果になる場合があります。それにはこの応用記述で調査した幾つかの改善策があります。特定応用に適合する単一手法はありませんが、この概略調査では、ASFの大きさを減らす時に記述された技術を手軽に試せます。読者はそれらｼｽﾃﾑの基本構造と必要条件の完全な知識を持つと仮定されます。大きさ削減は機能性、可搬性、性能に於ける二律背反の釣り合いの技で、最適な解決策を決めることは相当な試行錯誤を伴い得ます。この過程を手助けするため、最初に応用の大きさがどう決まり、そして何のASF部品がこの大きさの一因となるかに集中します。

1.1. ASFの特徴と大きさ

Atmel Studio内のASFｴｸｽﾌﾟﾛｰﾗ(Explorer)またはｳｨｻﾞｰﾄﾞ(Wizard)はAtmel Sudioﾌﾟﾛｼﾞｪｸﾄに含まれるべきASF機能を選択または選択解除するための主なｲﾝﾀｰﾌｪｰｽです。幾つかのASF部品がASF階層に於いてより低い他のASF部品に依存するのを強調することが重要です。このようなより高位のASF部品が選択されると、依存する全ての従属部品がﾌﾟﾛｼﾞｪｸﾄへ自動的に追加されます。これは幾つかの驚きになり得ます。1つのASF部品の包含は予期せぬ機能を引き擦り込むかもしれません。

Atmel Studioﾌﾟﾛｼﾞｪｸﾄで正しく構成される全ての部品を見るためにしっかりとASFｴｸｽﾌﾟﾛｰﾗを調査することによって始めてください。可能な処は何処でも応用要件を減らすことによって余分な部品を取り去るために出来るだけ努力をしてください。例えば、printf()支援の目的のためだけにｿﾌﾄｳｪｱ浮動小数点ｴﾐｭﾚｰﾄがｲﾝｸﾙｰﾄﾞされていませんか? 何らかの配慮なしにprintf()は浮動小数点形式(”%f”)を支援し、この機能を支援するのに必要なﾗｲﾌﾞﾗﾘを持ち込むでしょう。ANSI Cﾗｲﾌﾞﾗﾘ支援は更に以下が検討されます。

真直ぐな手法は単に機能の選択と選択解除を行うことで、どの応用機能を省略するか、それが厳密に必要とされるかどうかを決めます。ASF部品のﾁｪｯｸを外して解決策を再構築し、応用が簡単化し得るかどうかを決めてください。この試行錯誤手法は2つの方法で作業することができます。1つは完全な応用で開始し、ﾌﾟﾛｼﾞｪｸﾄの大きさを減らすために奮闘して機能の選択を解除し始めてください。代わりの1つは非常に基本的な開始位置で開始することができ、完全な応用を再作成するのに必要とされるASF機能を漸次追加してください。この後者の手法は増加するﾒﾓﾘ要件の機能単位評価を許します。

やがて来る章で検討される理由のため、与えられた何れかのASF部品の分離した大きさや、最終の顧客ﾌﾟﾛｼﾞｪｸﾄ構築の環境の外側で推測することは簡単な問題ではありません。構成設定定数、ｺﾝﾊﾟｲﾗ選択、最適化水準は少しを指定するだけで最終構築の大きさに影響を及ぼします。従ってAtmel Studioの組み込みﾂｰﾙの利用方法を学び、最終的なﾌﾟﾛｼﾞｪｸﾄ構築を学ぶために系統的な手法を採用し、そこに何があるかを決めるために必要な科学捜査を実行することがもっと大事です。

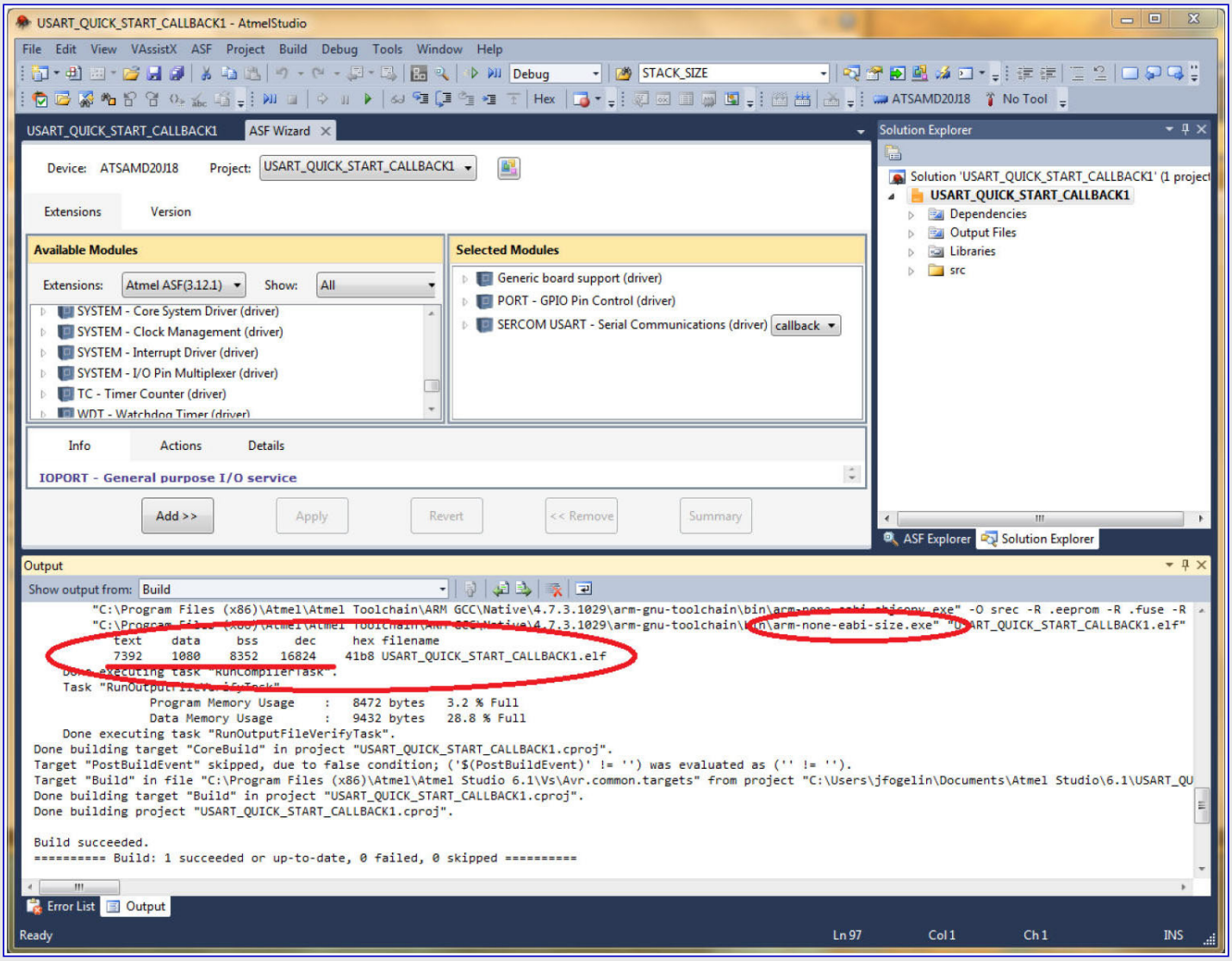
1.2. ﾒﾓﾘ必要条件

1.2.1. 静的ﾒﾓﾘ必要条件

一旦ﾌﾟﾛｼﾞｪｸﾄが構築されると、静的ﾒﾓﾘ必要条件を決めるために結果として生じる値域を分析することができます。応用はﾌﾟﾛｸﾞﾗﾑｺｰﾄﾞと定数ﾃﾞｰﾀ(例えばtext領域)用にﾌﾗｯｼｭ ﾒﾓﾘを必要とします。また揮発性ﾃﾞｰﾀ(例えばdata領域)、未初期化変数(例えばbss領域)、緩衝部、ﾌﾟﾛｸﾞﾗﾑ用ｽﾀｯｸ用にRAMを必要とします。ASFの場合、ｽﾀｯｸの位置と大きさはﾌﾞｰﾄ ﾛｰﾀﾞによって決められるのではなく、むしろ応用のﾃﾞｰﾀ領域内に構築されます。

この情報はAtmel Studio構築出力(build ouput)ｳｨﾝﾄﾞｳで提供されます。これの例の基は呼び戻し動作でのUSARTに関するAtmel SAMD20即時開始の手引きです。図1-1.をご覧ください。

図1-1. Atmel Studioで静的メモリ必要条件を探す場所



dataとbssの領域は結合されてﾃﾞｰﾀ ﾒﾓﾘ(RAM)の範囲内に割り当てられます。図1-1.の例を観察してください。未初期化ﾃﾞｰﾀ領域(bss)は8352ﾊﾞｲﾄです。何故こう大きいのですか? 答えはﾌﾟﾛｸﾞﾗﾑ実行用ｽﾀｯｸの静的構成設定であるからです。ｽﾀｯｸはその頂上で割り込み処理呼び出しを扱うために或る程度余分な空間を持ち、最大限可能な深い関数呼び出しの入れ子を処理するのに充分大きくなければなりません。この大きさの推測は挑戦になり得、大いに応用固有です。Atmel StudioとASFは適切なｽﾀｯｸの大きさ見積もりを試みません。それはﾘﾝｶ ｽｸﾘﾌﾟﾄ内のｼﾝﾎﾞﾙによって明示的に定義されます。利用可能なRAMの1/4である既定の大きさを上書きするために、図1-2.で実演されるようにｼﾝﾎﾞﾙの__stack_size__を再定義しなければなりません。

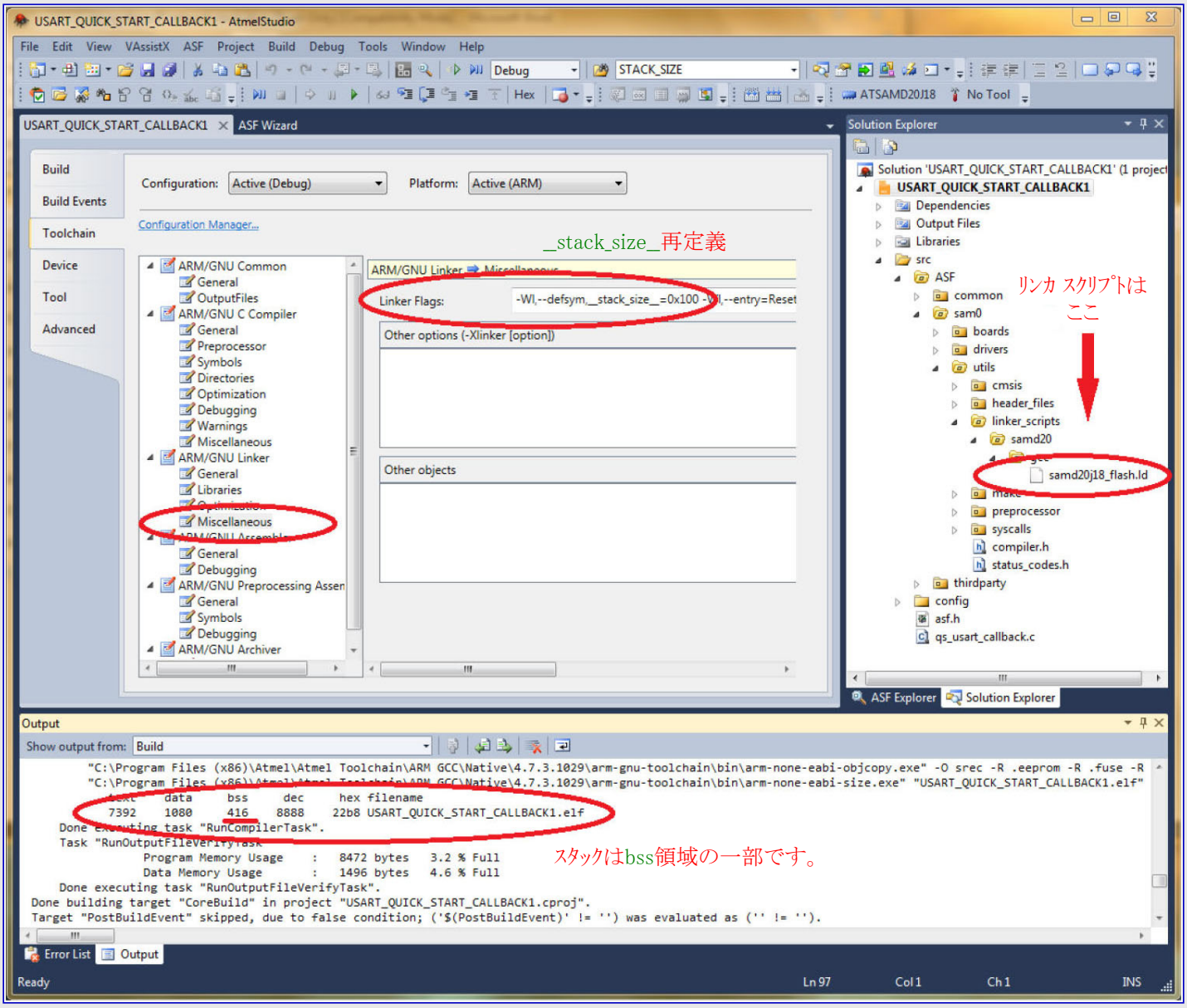
ﾛｰﾀﾞ ｽｸﾘﾌﾟﾄはASF樹状構造の一部で、__stack_size__の既定定義を含みます。それらを変更することができ、またはToolchain任意選択下のﾌﾟﾛｼﾞｪｸﾄ ﾌﾟﾛﾊﾟﾃｨで変更することもできます。以下の文を追加してください。

“-Wl,--defsym,__stack_size__=0x1000”

0x1000ﾊﾞｲﾄのｽﾀｯｸ量のためにmiscellaneous副標題下のlinker flagsへ。先頭と後尾の2連続下線に注意してください。

最適なｽﾀｯｸ量を決めるための技法はここでは網羅されませんが、ｽﾀｯｸが小さすぎる場合に応用がｽﾀｯｸ破壊の危険になる事実を認識してください。この状況ではｽﾀｯｸ ﾎﾟｲﾝﾀが割り当てられた領域を向こうへ行き、その進路内の何をも上書きします。ｽﾀｯｸ利用が常に確定的または完全に既知ではないので、これらのﾊﾞｸﾞは断続になり得て追跡が難しくなります。殆どの開発者はこれらの種類の応用異常を避けるために保守的な安全緩和策で設計します。

図1-2. Atmel Studioで既定スタック量の上書き方法



リンカ スクリプトはスタック量、プログラムとデータが共に選択したデバイスで利用可能なメモリを超える場合に誤りを生成します。もっと高度な応用では、リンカ スクリプトの更なる特注化によってメモリを除外することができます。これは無線ネットワーク通信での更新支援や特殊緩衝部用空間予約にメモリの塊を取り除いておくことができます。

応用の静的メモリ必要条件はプログラム、データ、そして(スタックを含む)未初期化変数で構成されます。使用情報はAtmel Studioの出力ウィンドウで毎回の構築の最後で要約されます。各々がどの程度応用の全体量に寄与するかをより良く理解するために、ASF機能の内包と除外を実験してください。以下で、最終構築像の正確な内容を決めるための追加の道具と技術を調査します。

1.2.2. 動的メモリ必要条件

多くの基盤では、RAMの領域は動的メモリ割り当てに置かれます。動的メモリ割り当ては応用が走行してやって来るデータや入力を扱うためにいくつかの追加メモリを必要とする時に起きます。このRAMの領域は一般的にヒープとして参照されます。Atmelソフトウェア枠組み(ASF)はヒープを持たず、このようなメモリは置かれません。全てのメモリ資源は通常のC変数と構造体の定義を用いて予め定義され、構築時に静的に置かれます。この簡単化は動的メモリ割り当てから来る落とし穴を避けます。期間上でメモリが割り当てられ、そして解放される時に、このような1つの落とし穴が発生します。ヒープは断片化になり、要求を満たすのに十分な大きさの塊を見つける時間遅くなり得て、或る場合では失敗になり得ます!。組み込みプログラミングに於けるこの状況に対する救済策は応用をリセットして再開するだけことだけで、これはとても受け入れられません。

動的メモリ管理に関してASFには注目に値する2つの例外があります。1つ目は軽量インターネット規約(LwIP:lightweight internet protocol)です。

1.2.2.1. LwIP

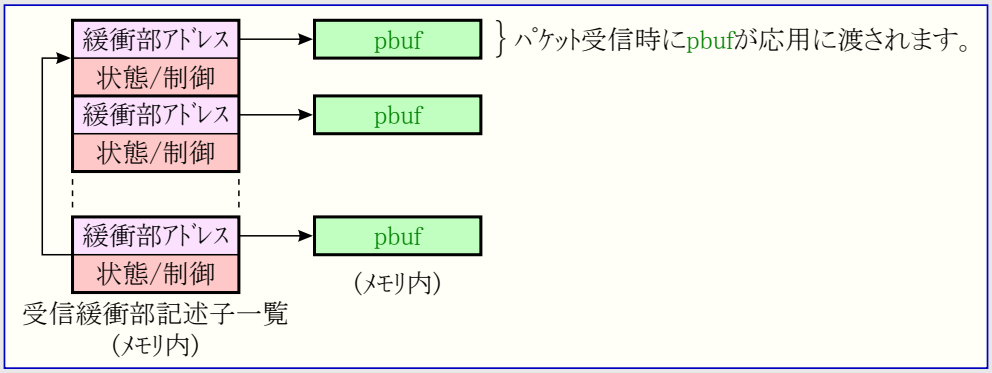
LwIPではﾒﾓﾘの一部がpbufとして知られるものに置かれます。LwIPのpbufﾃﾞｰﾀ構造体は1つのﾈｯﾄﾜｰｸ ﾊﾟｹｯﾄを表すように設計された緩衝部の一覧に繋げられます。実際に、使用者応用が使用者ﾃﾞｰﾀと共にtcp_write()関数呼び出しを実行するとき、LwIP階層は対応するﾊﾟｹｯﾄを構築するために様々な規約先頭部をｶﾌﾟｾﾙ化する(埋め込む)ように先頭へ追加(そして或いは他のpbufを追加)します。pbuf割り当てに依存して、IwIP階層は規約先頭部を追加するのに充分な空間を残す、または単にpbufの接続された一覧に別のpbufを追加することができます。pbufﾃﾞｰﾀ構造体を使い、緩衝部全体の複写なしに先頭部を追加することができます。

このﾃﾞｰﾀ構造体はﾊﾟｹｯﾄ ﾃﾞｰﾀを保持するための動的ﾒﾓﾘ割り当てと、静的ﾒﾓﾘ割り当てでの参照の支援を提供します。pbufの接続された一覧はpbuf連鎖として参照され、従って完全なﾊﾟｹｯﾄは多数のpbufに及ぶかもしれません。

0複写受信緩衝部は大きな量のﾃﾞｰﾀを転送する時に複写された緩衝部に渡ってｼｽﾃﾑ性能を改善することができます。また、これはIwIPﾈｯﾄﾜｰｸ階層によって割り当てられたpbufを直接的に使い、従ってﾒﾓﾘ構造設定を簡単化するため、より少ないﾒﾓﾘを使います。ﾊﾟｹｯﾄ受信を許可するために、GMACﾈｯﾄﾜｰｸ ｲﾝﾀｰﾌｪｰｽ初期化段階中にpbufが予め割り当てられます。

ﾊﾟｹｯﾄ受信時にGMAC IPは受信したﾃﾞｰﾀをpbufの本体領域に置きます。その後にﾈｯﾄﾜｰｸ ｲﾝﾀｰﾌｪｰｽﾄﾞﾗｲﾊﾞはそれの記述子からこのpbufを削除し、それを余分な複写なしにIwIPﾈｯﾄﾜｰｸ階層へ渡します。一旦ﾃﾞｰﾀが使用者応用によって使われてしまうと、IwIPﾈｯﾄﾜｰｸ階層が自由になるため、ﾈｯﾄﾜｰｸ ｲﾝﾀｰﾌｪｰｽはこのpbufを保持しません。最後に、新しいpbufは今更新されて使用準備が整った直前の緩衝部記述子に割り当てられます。

図1-3. 0複写受信記述子一覧と緩衝部



LwIPの使用ﾒﾓﾘ全体はMEM_SIZE定義の構成設定によって管理されます。この意味で、ﾒﾓﾘ配置は静的です。けれどもpbufはこの静的領域内で動的に割り当てられます。受信(pbuf)緩衝部によって使われるﾒﾓﾘ量は定数でGMAC_RX_BUFFERS×PBUF_POOL_BUFSIZEに等しく、IwIPの合計ﾒﾓﾘ量MEM_SIZEがそれによって設定されなければならないことを確実にしてください。

1.2.2.2. FreeRTOS

RTOSｶｰﾈﾙはﾀｽｸ、待ち行列(ｷｭｰ)、相互排除(ﾐｭｰﾃｯｸｽ)、ｿﾌﾄｳｪｱ計時器、ｾﾏﾌｫが作成される毎にRAMを割り当てます。FreeRTOS™はそれの可搬層でﾒﾓﾘ割り当てAPIを保ちます。可搬層は実時間ｼｽﾃﾑに適合する応用特有実装を許します。FreeRTOSは単純にmalloc()とfree()に頼りません。

FreeRTOSは4つの試供ﾒﾓﾘ割り当て実装(各々、heap_1.c, heap_2.c, heap_3.c, heap_4.c)を含み、ASF third party FreeRTOS副ﾌｫﾙﾀﾞのSource¥Portable¥MemMangﾌｫﾙﾀﾞに配置され得ます。

heap_1.c計画は全ての最も単純な実装です。これは一旦割り当てられてしまうと、解放されるべきﾒﾓﾘを許しません。多くの徹底した組み込み応用はｼｽﾃﾑ ﾌﾞｰﾄ時に必要とする全てのﾀｽｸ、待ち行列(ｷｭｰ)、ｾﾏﾌｫなどを作成し、その後に(応用が再びOFFにされる、またはﾘﾌﾞｰﾄされるまで、)ﾌﾟﾛｸﾞﾗﾑの存続期間の間、それらの物の全てを使います。決して何も削除されません。配列の総量(ﾋｰﾌﾟの総量)はFreeRTOSConfig.hで定義されるTOTAL_HEAP_SIZEの構成によって設定されます。

heap_2.c計画は計画1と異なり、最良適合算法を使い、解放されるべき以前に割り当てられた塊を許します。けれどもこれは隣接した解放塊を単一の大きな塊に結合しません(合体算法を含みません)。利用可能なﾋｰﾌﾟ空間の総量はFreeRTOSConfig.hで定義されるTOTAL_HEAP_SIZEの構成によって設定されます。

heap_3.c計画は殆どの場合であなたが選んだｺﾝﾊﾟｲﾗで供給される、標準Cﾗｲﾌﾞﾗﾘのmalloc()とfree()の関数用の簡単なﾗｯﾊﾟｰを実装します。このﾗｯﾊﾟｰは単位ｽﾚｯﾄﾞ ｾｰﾌなmalloc()とfree()の関数を作ります。

注: FreeRTOSConfig.hで設定するTOTAL_HEAP_SIZE構成設定はheap_3が使われる時に無効です。

heap_4.c計画は計画2と異なり、先頭適合算法を使い、隣接した解放ﾒﾓﾘを単一の大きな塊に結合します(合体算法を含みます)。利用可能なﾋｰﾌﾟ空間の総量はFreeRTOSConfig.hで定義されるTOTAL_HEAP_SIZEの構成によって設定されます。

LwIPとの時に、FreeRTOSの全体ﾒﾓﾘ構成設定は構築時に静的に置かれます。けれども、ﾒﾓﾘ資源の管理は動的です。それはFreeRTOS内で構成されたﾋｰﾌﾟ算法によって管理されます。これは例えば、多すぎるﾀｽｸやｶｰﾈﾙ ｵﾌﾞｼﾞｪｸﾄを作成することによってFreeRTOSﾋｰﾌﾟを使い尽くすことが起こり得ます。故に応用によってどの位の動的ﾒﾓﾘが実際に使われかを決めるためにFreeRTOSﾋｰﾌﾟ利用の概略を描くことが重要です。xPortGetFreeHeapSize()ﾙｰﾁﾝはこれに関して有用になります。この関数は未割当に留まるﾋｰﾌﾟ空間の総量を返し、最適化されるべきTOTAL_HEAP_SIZE設定の構成を許します。

2. 道具でのASK応用メモリ必要条件削減

今や記述されたASKに基づく応用に関する静的と動的なメモリ必要条件で、応用メモリ必要条件削減を手助けする道具と技術に注意を向けます。この調査で網羅される各々とあらゆる手法はあらゆる応用に適用するとは限りませんが、少しの調査と試行錯誤が正しい組み合わせの決定を手助けするでしょう。

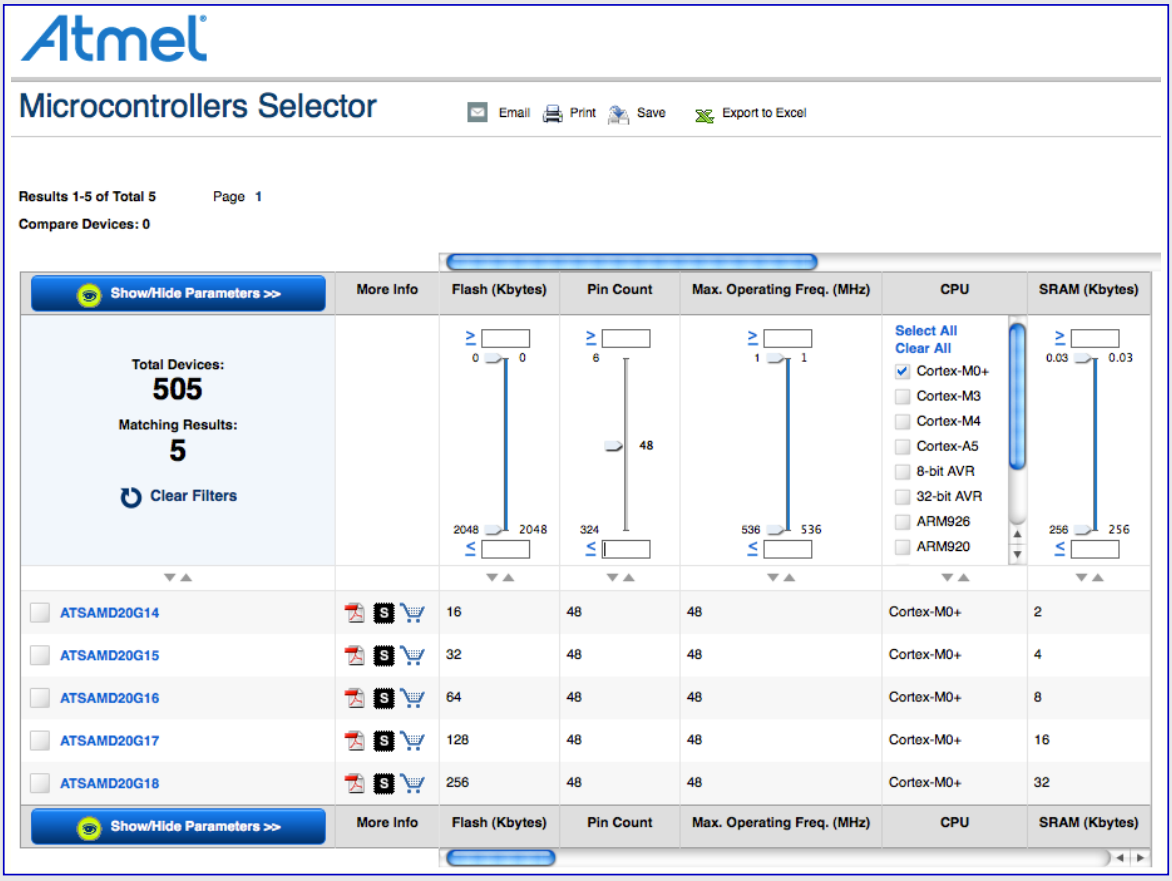
2.1. マイクロ コントローラ選択

応用の大きさのどんな重大な調査も適切な大きさのマイクロ コントローラの選択で始まって終わります。応用の大きさの問題に対する明白な解決策と同時に、製品開発寿命間にマイクロ コントローラが選択されるべき時について戦略上考える価値があります。

最初に指定されたマイクロ コントローラはかなり度々大きさと費用の仮見積りに基づきます。その後、開発期間で、様々な機能の大きさ対恩恵の最終決定を実行することができます。或る場合、同じマイクロ コントローラ系列のより大きなメモリ派生品を指定することは適合する応用を得るための最適時の解決策です。他の場合、応用の機能削減が唯一の救済策です。或る開発者は開発中に最大メモリの派生品を意図的に選び、その後に最後の製品でより小さなメモリの大きさの派生品に低下指定します。

図2-1.は内部メモリの大きさの範囲を未だ提供すると同時にピン互換の部品指定を容易にする、要素単位マイクロ コントローラ選択部を示します。

図2-1. 要素単位メモリ別種選択: http://www.atmel.com/products/selector_overview.aspx



2.2. ツールチェーン選択と調整

コンパイラの選択は結果コードの大きさに於いて重要な衝撃を持ち得ます。コンパイラの選択は多数の要素に関係します。いくつかは商用料金で利用可能で、クライアント毎単位で売られます。いくつかのコンパイラ、特にGNU gccは無料で利用可能です。全てのコンパイラはここ数年に渡って改善されており、実行量またはコード量効率のいずれかに於けるコンパイラの最適化より優れているような手作業コード ルーチンは益々難しくなります。1つのコンパイラでの同じコードはもっと簡潔になるかもしれないし、ならないかもしれず、ベンチマークは一般的な指標だけを提供します。

コンパイラの性能を正しく評価するにはツールチェーンの構築任意選択を注意深く構成設定しなければなりません。利用可能な最適化はコンパイラによって変わります。結局、そのコードの完全な理解はコードの大きさに於ける著しい減少が生じ得る、かなりの経験とで一致されたコンパイルです。或る最適化レベルに於いて、信頼に足らなくなるためにIDE内のソース レベル デバッグが珍しくないことに気付いてください。それはデバッガに対して命令と連携する正確なソース行の確立を難しくし得ます。強力な開発環境が優先なら、プログラムを過度に最適化しないでください。

Atmel Studioでは、様々な構築構成設定、例えばrelease build(公開用構築)やdebug build(デバッグ用構築)を保存することが可能です。実際問題として、これはより低く最適化されたデバッグ版が利用可能なメモリに入らない、またはより高く最適化した公開版が機能せず、信頼に足るデバッグができない時に挑戦することができます。この問題はそれにも拘わらず経験が鍵です。コンパイラ任意選択を綿密に読み、増加する侵略性に基づくコードに渡す一連の最適化を作ってください。開発環境が障害になるなら、最適化レベルを減らしてください。

2.2.1. ツールチェーン ベンチマーク

執筆時点に於いて、IAR™とKeilはコード量効率に於いて未だGNU gccよりも全般的に性能が優れています。結果がかなり変わる一方で、コード量での20%削減は適度な大きさの応用に対する妥当な期待です。IARとKeilは評価版を提供し、長期的な商用料金が受け入れられるなら、単純にベンチマークを信頼するよりも単にこれらのコンパイラを試みることが最良です。

2.2.2. ツールチェーン構築任意選択

Atmel Studioではコンパイラ最適化レベルに対するインターフェースがプロジェクト プロパティ ダイアログの方法によって管理されます。ARM®/GNU Cの場合、Compiler cinfigurationのOptimization副項目下で得られます。基本的な最適化レベルは引き落とし選択で、他で一般的に使われるコマンド行スイッチはチェックリストとして提供されます。利用可能な更に多くのフラグがあり、これらは手動によってOther optimization sテキスト枠に入力することができます。

図2-2.では-OsのGNU最適化レベル(大きさ最適化)が選ばれています。これが図1-1.で使われた同じ例で、大きさで控え目の2%削減なことに注意してください。応用の大きさ全体が小さく、大きさが0x2000の_stack_size_に左右されるので、この場合の結果は幾分迫力に欠けます。プログラム用textの大きさがより小さくなる一方で、dataとbssの両領域が実際により大きくなることにも注意してください。これは結果を注意深く学び、応用の状況に於ける最適化を考慮する必要の説明を手助けします。この場合、開始のためのより良い場所は最悪な場合のスタック使用法の分析になるとの結論を下します。スタック削減手法は後で検討されます。

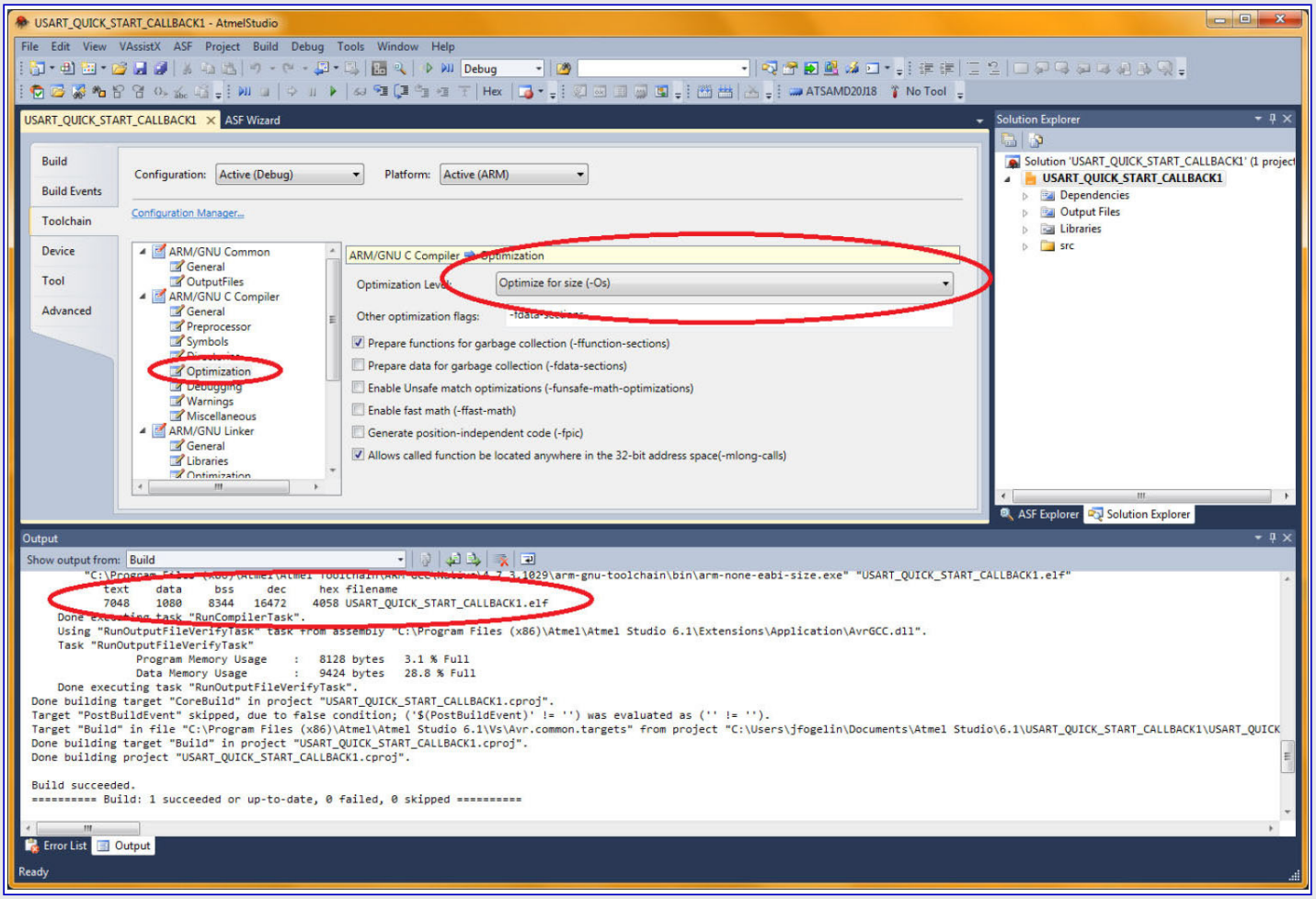
コンパイラ最適化フラグの詳細な検討の例(この場合はこの応用記述で使われるGNUコンパイラ)はここhttp://gcc.gnu.org/onlinedocs/gcc-4.7.3/gcc/Optimize-Options.htmlで得られます。この資料の短時間の走り読みはきめ細かいコンパイラ最適化がどう制御されるかを実演します。それはまた意気地なし用ではありません!。実際問題として、殆どの開発者は単に-Oレベルの1つから選んで終わりにします。

1つのARM/GNU C最適化フラグは更なる検討に値します。"Prepare functions for garbage collection(塵掃除用関数準備)"チェック枠(-ffunction-sections)はコード量削減に至る時に重量運搬船です。チェックされると、このフラグはELFオブジェクト ファイルのそれ自身の区域で各ファイル内に各関数を置きます。リンク時が来ると、(一般的に重荷または不実行コードとして参照される)参照されないこれらの関数は最後の姿にリンクされません。これは実質的節約を実現することができます。libm.aとlibc.aもこのフラグに対する価値ある候補であることに注意してください。Atmel配布版のnewlib-nanoのlibc_s.aは以下のフラグでコンパイルされます。

```
"-g -Os -ffunction-sections -fdata-sections -fno-exceptions … "
```

Cライブラリでもっと下を読んでください。あなたのリンク行に他のライブラリが有る場合、潜在的に更にもっと節約を実現するためにそこで最適化レベルを調査してください。このフラグの許可の良い副産物、それは定義によってtest-coverage.Dead code(不実行コード適用範囲検査)のcode-coverage(コード適用範囲)統計を改善し得ることで、実行されません。

図2-2. GNU -OsのAtmel Studio最適化レベル選択



2.2.3. 組み込みCライブラリ選択

AVR®の場合、供給されるCライブラリは考えに於いて簡潔での着手で設計されて構築されました。強健さや拡張性を提供する、競合するCライブラリ実装は殆どありません。それは本質的に町で唯一の方針です。stdioの領域を注意深く参照し、データ、特に文字列を応用のプログラム領域内へ置くのにpragmaを利用してください。それが不正に参照されない限り、応用の大きさに寄与しないとの確証を持ってそれを使ってください。

ARM Cライブラリについては同じ事が言えません。先ず第一に2つの種類、libc.a(newlib)とlibc_s.a(newlib-nano)になります。全ての装飾を持つCライブラリを求める人々に対し、newlibがあなた向けです。全ての膨張した形式任意選択(特に良いのは浮動小数点模倣動作支援での巻き込みを避ける浮動小数点への弱いシンボル参照)なしでprintf()の実装を求める人々、そして広範囲なchar支援や過剰に複雑なヒープ管理を必要としない人々に対し、newlib-nanoがあなた向けです。

例えばprintf("Hello World!¥n")を実施する簡単な応用は80%縮小することができます!。

既定により、Atmel StudioのARMに基づくプロジェクトは参照のためにどのCライブラリも指定せず、殆どのプログラム作成者は惰性によってリンカ行で-lcを使います。次回に-lc_sを試みてください。

更にストリームI/Oまたはiov緩衝の使用を制限することが賢明です。正しいI/Oシステムがない時に、これらの関数は驚くほどの量の空間を占有し得ます。newlib-nanoはnewlib(または完全な組み込みRTOS)ほど強力に扱われないFreeRTOS(または何れかのRTOS)の環境で使われる時の再入可能性の問題もあります。故にlibc_s.a経由で緩衝されたI/Oを使わなければならない場合、単一スレッドからそうしてください。

3. 応用フラッシュ必要条件削減

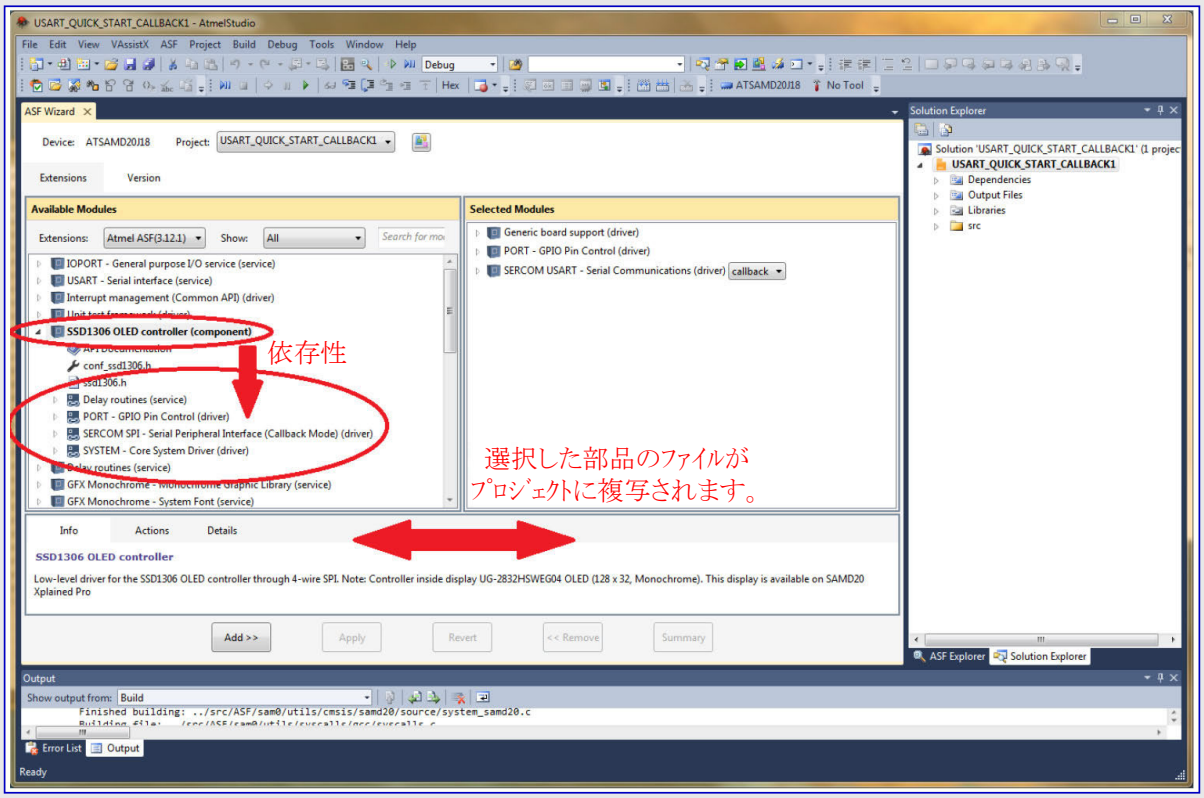
道具を調節することによる応用の大きさ最適化が充分でない場合、コードそれ自体に戻らなければなりません。text(プログラム空間)の量はAtmel Studio構築の完了に於ける出力ウィンドウで要約されます。簡単に言えば、何かが行われなければなりません。との機能を切るかをどう決めますか? 不必要なコードが構築形成物に蔓延っていないことを確信を持ってどう知りますか? 本章では応用のフラッシュ必要条件を減らすためにこれらの問題を考察して手法を提案します。

3.1. ASFウィザード部品除去

ASFはソフトウェア部品の集合として構成されます。これらの部品はドライバ、サービス、または第三者ミドルウェアで有り得ます。これらの部品に関連するソース コードの全てはAtmel Studioのインストール フォルダに保持されます。この樹状系図での組み込みは部品の各々のメタデータを記述するのに拡張可能マークアップ言語(XML:Extensible Markup Language)を使う部品記述子ファイルです。メタデータは部品を構成するファイルの内容目録、そして依存する他のどのASF部品も含みます。これらの部品相互依存性の追跡はASFに基づく新プロジェクトへ必要なファイルの全てを正確に複写することをAtmel StudioのASFウィザードに許します。与えられたどのプロジェクトも選択した構成設定に関連するファイルのASF樹状系図の部分集合を単に複写されます。

このXMLファイルの集合はASF内容のデータベースとして考えられます。部品依存性を追跡するためのXML学習は時間の浪費かもしれませんが、感謝することにAtmel Studioはこの処理を直接的に行います。依存性の図を閲覧する最良の方法はAtmel StudioのASFウィザードで作業することです。図3-1.を参照してください。

図3-1. Atmel Studio ASFウィザードでのASF部品閲覧



各部品の依存性が補助部品として列挙されることに注意してください。この場合、OLED制御器はSPIに基づき、従って依存性の1つはSERCOM SPIﾄﾞﾗｲﾊﾞです。1つは部品階層樹状系図を単に展開することによってそれらの相互依存性を閲覧することができます。部品単位でﾌﾟﾛｼﾞｪｸﾄを構築してその構築の大きさの経緯を保つことにより、その機能の相対的な重みを勘定して評価することができます。これはどの機能が合致し、どの構成設定が応用に最適かを決める最初の段階を示します。

ﾂｰﾙﾁｪｰﾝのそれとASFｳｨｻﾞｰﾄﾞの機能を混同しないことが重要です。ASFｳｨｻﾞｰﾄﾞはｺｰﾄﾞそれ自体またはｼﾝﾎﾞﾙによる相互依存性を調べません。それは各部品の内容目録をAtmel Studioのﾌﾟﾛｼﾞｪｸﾄ ﾌｫﾙﾀﾞ内へ忠実に複写し、その構築環境を準備します。構築時、ｺﾝﾊﾟｲﾗとﾛｰﾀﾞは(先に検討されたように構成設定された時に)どんな未使用機能をも最適化し、さもなければｺｰﾄﾞを最適化するでしょう。

この点を超えるｺｰﾄﾞ量削減はｺｰﾄﾞ自体への変更が必要です。ﾌﾟﾛｼﾞｪｸﾄがASF部品の複製である限り、そこでのｺｰﾄﾞ変更は公開された部品から分離されます。1つは実験してASFｳｨｻﾞｰﾄﾞの使用によって元の公開版へ戻ることができます。けれども、何れかの変更が行なわれている場合、ASFｳｨｻﾞｰﾄﾞの方法によって元のASF部品で局所のﾌﾟﾛｼﾞｪｸﾄ変更の上書きを避けることに注意してください。

3.2. 依存性追跡と機能調整

ASFがﾌﾟﾛｼﾞｪｸﾄに関連するｿｰｽ ｺｰﾄﾞの全てを提供すると言う事実にも関わらず、どんな機能で、ﾂｰﾙﾁｪｰﾝ構築時間最適化を記し、実際に最終ﾊﾞｲﾅﾘで終わるかは必ずしも明白だとは限りません。結果として生じる単位部の各々の大きさの感覚を得ることも有用です。

最終ﾊﾞｲﾅﾘの内容学習を手助けするARM/GNU gccの一部である2つの道具があります。1つ目の道具は実際には毎回の構築中に作成されるﾌｧｲﾙです。Atmel Studioの出力ﾌｧｲﾙの1つは最終ﾘﾝｸ段階中に以下のﾌﾗｸﾞの方法による工程表です。

```
-Wl,-Map="USART_QUICK_START_CALLBACK1.map"
```

これは工程表ﾌｧｲﾙに含まれる詳細を習熟するためにかなりの時間的価値があります。工程表の内容の完全なﾁｭｰﾄﾘｱﾙは、その情報がldに関してGNU手引書内でより良く記載されているため、この応用記述の範囲外です。この表がﾒﾓﾘ配置、大きさ、そして構築形成物に関連する詳細の実質的全てを要約するとだけ言っておきます。それは最適化機械のための工程表です。

工程表ﾌｧｲﾙが我々に何かの情報を提供する一方で、2つ目の道具である、GNU nmは我々がどうそこに至ったかを推測する手助けをします!。このﾂｰﾙﾁｪｰﾝ ﾕｰﾃｨﾘﾃｨ ﾙｰﾁﾝはﾌｧｲﾙの相互依存性そしてﾌｧｲﾙの任意組み合わせさえをも調査するのに使うことができます。nmに対して多数の有用なｺﾏﾝﾄﾞ行ﾌﾗｸﾞがありますが、我々の目的に対してここが最も重要です。

```
% man nm
       -g, --extern-only
              Display only external symbols.
              (外部ｼﾝﾎﾞﾙだけを表示)
       -n, -v, --numeric-sort
              Sort symbols numerically by their addresses, rather than alphabetically by their names.
              (それらの名前によるｱﾙﾌｧﾍﾞｯﾄ順よりもむしろそれらのｱﾄﾞﾚｽによる数値順でｼﾝﾎﾞﾙを並び替え)
       -u, --undefined-only
              Display only undefined symbols (those external to each object file).
              (未定義(各ｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙへのそれらの外部)ｼﾝﾎﾞﾙだけを表示)
       --defined-only
              Display only defined symbols for each object file.
              (各ｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙに対する定義済みｼﾝﾎﾞﾙだけを表示)
```

ｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙ内で未定義ｼﾝﾎﾞﾙを調査することにより、何が予期せぬ機能の持ち込みを或る単位部に起こさせるかを素早く確認することができます。一旦予期せぬｼﾝﾎﾞﾙ参照が分離されると、関連するｿｰｽ ｺｰﾄﾞへの素早い逆参照が理由の説明を助けるでしょう。この技法は増分ﾘﾝｸや部分ﾘﾝｸでそれを結合する場合に特に強力です。

増分ﾘﾝｸはldに対する入力として順に供給することができる、1つの単一再配置可能ｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙへ複数の再配置可能ｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙの集合を結合する技法です。換言すれば、集合のｵﾌﾞｼﾞｪｸﾄ単位部間の全ての解決可能なｼﾝﾎﾞﾙ相互依存性はﾘﾝｸされ、集合の外側のｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙを参照する未定義ｼﾝﾎﾞﾙだけが残ります。ｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙの選ばれた集合は任意ですが、実際問題として、1つの分割は既知の機能の塊へｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙのこれらの集合(例えば複数ﾌｧｲﾙで構成されるASF部品)です。ｵﾌﾞｼﾞｪｸﾄ ﾌｧｲﾙの集合を増分ﾘﾝｸして残りの依存性見つけるのに必要な段階は次のとおりです。

```
../arm-none-eabi-ld.exe -o result.o -r one.o two.o three.o
../arm-none-eabi-nm.exe -u result.o
```

部分的にﾘﾝｸされたｵﾌﾞｼﾞｪｸﾄ単位部の様々な組み合わせでの一連の試みと工程表の注意深い調査は望まれない単位部が参照された場所への見識を提供するでしょう。一旦ｼﾝﾎﾞﾙ水準で確立されたなら、ｿｰｽ ｺｰﾄﾞ水準で依存性を調査するのにAtmel Studioの Visual Assist機能を利用してください。

前のように、この手法は望まれない単位部を捨てようとすることによって応用を漸次削除(分解)するのに使うことができます。或いは、この手法は漸次追加(増強)や機能が要る応用に使うことができます。どちらにしても、意外なものを見つけて、大きすぎる応用への根本原因であるｿｰｽ ｺｰﾄﾞ相互依存性を明らかにするかもしれません。次は何でしょう?

3.3. ｺｰﾄﾞ単純化

前項では、構築形成物で科学捜査分析を実行しました。我々は応用を単純化するのに可能なｺｰﾄﾞ変更での見識を持ちます。ここはいくつかの可能性です。

定数ﾃﾞｰﾀはRAMの保存のためにﾌﾗｯｼｭ ﾒﾓﾘに割り当てられます。或る場合、この方法で格納された定数ﾃﾞｰﾀが実際に多すぎることがあり得ます。まだ利用可能なRAMがある場合、揮発性記憶装置に定数ﾃﾞｰﾀを戻すようにｺｰﾄﾞを変更することを考慮してください。ｺﾝﾊﾟｲﾗへの情報の損失(そしてﾌﾟﾛｸﾞﾗﾑ作成者へﾊﾞｸﾞの警告!)があるため、単に"const"ｷｰﾜｰﾄﾞを削除することは推奨されません。定数ﾃﾞｰﾀの位置を管理する構造についてはﾂｰﾙﾁｪｰﾝの資料を参照してください。

多くのASFﾄﾞﾗｲﾊﾞは走行時に支援されて変更されるべき広範囲の動作種別範囲用に書かれています。ﾄﾞﾗｲﾊﾞの初期化は初期化構造体の方法によって要素別です。この手法には賛否両論があります。高度の実行時構成設定性、可搬性とｺｰﾄﾞ可読性の改善、しかし、この一般性はｺｰﾄﾞの大きさの犠牲になります。応用がﾄﾞﾗｲﾊﾞの既知の静的構成設定を念頭に置いている状況では、固定された機能を提供するﾄﾞﾗｲﾊﾞの初期化を単純化することは至極普通です。ASF SERCOM USART構成設定構造体と以下のｺｰﾄﾞ片でUSART初期化を考慮してください。

```
/* config構造体に既定構成を設定 */

config->data_order                = USART_DATAORDER_LSB;
config->transfer_mode             = USART_TRANSFER_ASYNCHRONOUSLY;
config->parity                    = USART_PARITY_NONE;
config->stopbits                  = USART_STOPBITS_1;
config->character_size            = USART_CHARACTER_SIZE_8BIT;
config->baudrate                  = 9600;
config->receiver_enable           = true;
config->transmitter_enable        = true;
config->clock_polarity_inverted = false;
config->use_external_clock        = false;
config->ext_clock_freq            = 0;
config->mux_setting               = USART_RX_1_TX_2_XCK_3;
config->run_in_standby            = false;
config->generator_source          = GCLK_GENERATOR_0;
config->pinmux_pad0               = PINMUX_DEFAULT;
config->pinmux_pad1               = PINMUX_DEFAULT;
config->pinmux_pad2               = PINMUX_DEFAULT;
config->pinmux_pad3               = PINMUX_DEFAULT;

/* 次にUSART初期化 */
usart_init(&usart_instance, SERCOM3, config);
```

ここでのこの構成設定の詳細が重要でない一方で、考慮すべき大きさの問題があります。この構成設定構造体は明らかに高度な実行時構成設定性を許します。構成設定構造体は48ﾊﾞｲﾄだけを取り入れます。一般性を犠牲にして、初期化は特定の動作種別配線に対してusart_initを書き直すことによって単純化することができ、初期化構造体の対応する領域を削除することができます。呼び戻し動作でのD20 SETCOM USARTﾄﾞﾗｲﾊﾞの場合、1Kﾊﾞｲﾄの命令の簡単化を達成できます。

別の素直な簡単化はASF pinmux支援を適用することができます。ASFは選択したﾋﾟﾝ機能に対して完全にﾏｲｸﾛ ｺﾝﾄﾛｰﾗ ﾋﾟﾝの再設定可能な範囲を支援するように書かれています。これは例のように広範囲な独自構成設定の範囲へ容易に目標を変えることができます。最終製品設計に於いて、ﾋﾟﾝ割り当てが完全に指定されます。usart_initでのように、ﾋﾟﾝ機能の配線割り当てが可能で、著しい節約を実現してください。

3.4. 例

上で要約された技術を適用することで、ここはADCとUSARTを使うAtmel SAMD20での簡単なASF応用を小型化する時のいくつかの結果の例です。この例ではｽﾀｯｸの大きさを既定の0x2000から減らしませんでした。この課題では、ADCの呼び戻し支援が削除され、最適化ﾚﾍﾞﾙが増され、newlib-nanoが利用され、選択されたﾄﾞﾗｲﾊﾞのﾎﾟｰﾘﾝｸﾞ版、IARｺﾝﾊﾟｲﾗが選択され、そして結局、初期化ﾙｰﾁﾝが簡単化されました。

Application: ASF example with ADC and USART drivers
(応用：ADCとUSARTのﾄﾞﾗｲﾊﾞでのASF例)
Atmel Studio, Optimization -O1 no nanolib, callback support enabled (standard configuration):
(Atmel Studio、-O1最適化 nanolibなし、呼び戻し支援許可 (標準構成設定))
Program Memory Usage : 10000 bytes
Data Memory Usage : 9392 bytes

Atmel Studio, Optimization -Os, nanolib, polled ADC and USART driver:
(Atmel Studio、-Os最適化 nanolib、ﾎﾟｰﾘﾝｸﾞ形式のADCとUSARTのﾄﾞﾗｲﾊﾞ)
Program Memory Usage : 7532 bytes
Data Memory Usage : 8280 bytes

IAR, With max optimization for size, polled ADC and USART driver:
(IAR、大きさ最大最適化、ﾎﾟｰﾘﾝｸﾞ形式のADCとUSARTのﾄﾞﾗｲﾊﾞ)
readonly code memory : 4548 bytes
readonly data memory : 208 bytes
readwrite data memory : 8276 bytes

IAR, Cutting out all adc_init and usart_init code except pinmux code and clock setup
(IAR、pinmuxとｸﾛｯｸ構成設定を除くadc_initとusart_initのｺｰﾄﾞ全てを省略)
readonly code memory : 2788 bytes
readonly data memory : 208 bytes
readwrite data memory : 8276 bytes

Some function optimizations remain, error handling could be disabled, but this is sufficient.
(いくつかの関数最適化が残り、異常処理は禁止されるかもしれませんが、これは充分です。)

4. 応用RAM必要条件削減

或る場合に於いて、応用は利用可能なﾌﾗｯｼｭ ﾒﾓﾘへ楽に合いますが、読み書きﾃﾞｰﾀを支援するのに充分なRAMがありません。ここはいくつかの取り組み方です。

4.1. ﾃﾞｰﾀ ｾｸﾞﾒﾝﾄと定数ﾃﾞｰﾀ

工程表、ld、それとこれらの道具が更にこの調査に適用されるよう、前項で網羅されたnmに関する資料を再吟味してください。(ｼﾝﾎﾞﾙ名順に相対するものとしての)数値順でのﾃﾞｰﾀ ｼﾝﾎﾞﾙの一覧があなたの応用で最も重いデータ構造体を決めるのを許すことに注意してください。nmのいくつかの版は更にあなたのためにこの数値計算を実行します。

このﾃﾞｰﾀのいくつかは読み出し専用に宣言することができますか?。もしそうで、ﾌﾗｯｼｭ内に未だ空きがあるなら、再びそれを割り当ててください。より重いﾃﾞｰﾀ構造体のいくつかはconstantやvolatileの部分に部分集合化できますか?。配列のいくつかは既定で大きすぎですか?。等々。

8ﾋﾞｯﾄから32ﾋﾞｯﾄの基本構造まで移植される時に、しばしばﾃﾞｰﾀ構造体は不必要に大きくなります。整数は実際に長さ32ﾋﾞｯﾄですか?。標準ANSI C形式(stdtype.h)の1つでそれらの大きさを縮小することができますか?。ﾌﾞｰﾙ値のﾋﾞｯﾄ包括や副8ﾋﾞｯﾄ領域を使うことができますか?。

これらの質問の全ては長期的な再使用、可搬性、可読性の考えに於いて評価されなければなりません。ｺｰﾄﾞの保守と維持がそのために被害を受けるなら、より小さなものが必ずしも望ましいとは限りません。

4.2. ｽﾀｯｸ割り当て調整

正しいｽﾀｯｸ量確立に挑戦することができます。厳密なｺｰﾄﾞ分析(または静的分析ﾂｰﾙ)なしで、控え目な複雑さの応用のﾙｰﾁﾝ呼び出しの最大の深ささえも確かめるのが容易ではありません。多くの呼び出しは実行時に於いて利用可能なﾃﾞｰﾀに基づいて条件付きで起こります。充分なｼﾐｭﾚｰｼｮﾝが必ずしも実用的だとは限りません。

共通的な1つの技法はｽﾀｯｸを既知のﾊﾞｲﾄ模様(例えば0xEE)で満たし、その後に実世界の或る妥当な期間実行後に、ｽﾀｯｸは高基準値を確立するために調査されます。高基準値はｽﾀｯｸのどの部分が無変化のままだったかを捜すことによってｽﾀｯｸ上で到達した最高(または最低)のﾒﾓﾘ位置を示します。この手法は確かに誤りやすくなります。妥当な期間はどの位の長いでしょうか?。ﾊﾟﾗﾒｰﾀとして構造体をﾙｰﾁﾝへ渡すことが可能で、この構造体が修正されない(または部分的に修正される)場合、ｽﾀｯｸ ﾎﾟｲﾝﾀが実際に到達した処よりも低いように見えるかもしれません。追加の安全係数で、この手法は妥当です。

別の手法は非同期割り込みでのｽﾀｯｸ ﾎﾟｲﾝﾀを周期的に採取することです。或る統計的に保守的な期間以上で、ｽﾀｯｸ ﾎﾟｲﾝﾀの高基準値を確立することができます。この手法での原則的な挑戦は真に乱採取を達成することで、この採取処理のために応用の正しい機能性を崩壊しないようにすることです。

最後に、応用が失敗するまでｽﾀｯｸ量が減らされ、その後に少しだけ増される試行錯誤手法があります。これは推奨されません。ｽﾀｯｸ衝突は診断するのが非常に難しくなり得、そしてそれらは殆ど常に有り得る最悪の時(例えば製品が出荷された後)に起こります。更にそれは応用を最小変更(ﾊﾞｸﾞ修正)がある脆弱な構成設定のままにし、奇怪で予期しない方法で応用を不安定にするかもしれません。

4.3. 先取り的 対 協調的の計画部

実行のﾀｽｸ(またはｽﾚｯﾄﾞ)の先取り的計画部を提供するために、多くのﾏｲｸﾛ ｺﾝﾄﾛｰﾗに基づく応用は今やRTOSに頼ります。ASFに基づく応用で使われるためにFreeRTOSが統合されて利用可能です。RTOSは未だそれらの一時的な動きを調整すると同時に、互いにに分離して開発される独立した応用機能を許す優れた枠組みを提供します。実時間応用は純粋に事象駆動やﾎﾟｰﾘﾝｸﾞの応用よりももっと確定性、拡張性、可搬性になり得ます。

先取りの利用に対する代価はそれが走行を計画されていない時間中に保存されるべき必要がある各ｽﾚｯﾄﾞの状態の複製です。これはﾏｲｸﾛ ｺﾝﾄﾛｰﾗのﾚｼﾞｽﾀ値を含むだけでなく、ｽﾀｯｸ毎の割り当ても含みます。実際問題としてRAMのｽﾚｯﾄﾞ毎のｽﾀｯｸ割り当ては実際に直ぐ合計することができ、ﾏｲｸﾛ ｺﾝﾄﾛｰﾗ上で同時に実行できるｽﾚｯﾄﾞ数を制限し得ます。

前項で検討されたように、正しくｽﾀｯｸを矯正することが重要です。この場合、各ｽﾚｯﾄﾞに対して適切なｽﾀｯｸ量を決める必要があるため、我々の問題が倍増しました!。共通的な技法は最低優先度でｱｲﾄﾞﾙ ﾀｽｸを使い、その他全てのﾀｽｸのｽﾀｯｸの高基準値を採取してｽﾀｯｸがいつか枯渇される場合に報告することです。

この応用記述の文脈に於いて、FreeRTOS ASF応用の大きさ削減は焦点の追加領域を除いて、全部の同じ手法に従います。それぞれとあらゆるｽﾚｯﾄﾞに対する必要物を正しくすることによって開始してください。ｽﾚｯﾄﾞを削除できれば、それはかなりのRAM節約を実現するでしょう。正しい大きさのｽﾀｯｸを得るよう仕事に励んでください。全てのﾀｽｸが割り込みをされ得、故に全てのﾀｽｸが最も深い入れ子の割り込み呼び出しのために最低限充分な予備のｽﾀｯｸ空間を持たなければならないので、割り込み処理ﾙｰﾁﾝは追加のｻﾌﾞﾙｰﾁﾝ呼び出しを持たないことを確実にしてください。割り込みも入れ子になり得ることを思い出してください。

ﾏﾙﾁ ｽﾚｯﾄﾞ応用のﾒﾓﾘ使用量の可能な全ての削減をし尽した後、協調的計画の採用を考慮し得ます。協調的計画は優先にされたﾀｽｸ(例えばContikiでのprotothread)の待ち行列を維持し、ｶｰﾈﾙが別のｽﾚｯﾄﾞから呼ばれる時に優先原則でそれらを配送します。協調的計画は決して別のｽﾚｯﾄﾞを先取りせず、故にｽﾚｯﾄﾞの前後関係維持物(ﾚｼﾞｽﾀ一式と個別用ｽﾀｯｸ)の余分な間接負担を持ち回る必要がありません。全てのｽﾚｯﾄﾞは時を得た方法でそれらのﾀｽｸを終了するためにその他各々に依存し、そして次のﾀｽｸを配給するためにｶｰﾈﾙに制御を戻します。この計画部が協調的として言及されるのはこの意味でです。

いくつかの公に利用可能な協調的計画部がある一方で、大部分の組み込み技術者は応用特有の実装を開発します。最も一般的な実装は時間が大事な関数が前面ﾀｽｸに配給され、ｱｲﾄﾞﾙ時に雑務が背面ﾀｽｸによって管理される前面/背面計画部です。ﾏﾙﾁﾀｽｸの観点からの基本と同時に、協調的計画部は応用の分割とお荷物さえない解体のRTOSの利点のいくつかを提供します。

5. 改訂履歴

文書改訂	日付	注釈
42370A	2014年9月	初版文書公開

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA TEL:(+1)(408) 441-0311 FAX: (+1)(408) 436-4200 | www.atmel.com

© 2014 Atmel Corporation. / 改訂:Atmel-42370A-Optimizing-ASF-Code-Size-to-Minimize-Flash-and-RAM-Usage_ApplicationNote_092014

Atmel®、Atmelロゴとそれらの組み合わせ、AVR®、Enabling Unlimited Possibilities®とその他は米国と他の国に於けるAtmel Corporationの登録商標または商標です。ARM®、ARM Connected®ロゴとその他はARM Ltdの登録商標または商標です。他の用語と製品名は一般的に他の商標です。

お断り: 本資料内の情報はAtmel製品と関連して提供されています。本資料またはAtmel製品の販売と関連して承諾される何れの知的所有権も禁反言あるいはその逆によって明示的または暗示的に承諾されるものではありません。**Atmelのｳｪﾌﾞｻｲﾄに位置する販売の条件とAtmelの定義での詳しい説明を除いて、商品性、特定目的に関する適合性、または適法性の暗黙保証に制限せず、Atmelはそれらを含むその製品に関連する暗示的、明示的または法令による如何なる保証も否認し、何ら責任がないと認識します。たとえAtmelがそのような損害賠償の可能性を進言されたとしても、本資料を使用できない、または使用以外で発生する(情報の損失、事業中断、または利益と損失に関する制限なしの損害賠償を含み)直接、間接、必然、偶然、特別、または付随して起こる如何なる損害賠償に対しても決してAtmelに責任がないでしょう。**Atmelは本資料の内容の正確さまたは完全性に関して断言または保証を行わず、予告なしでいつでも製品内容と仕様の変更を行う権利を保留します。Atmelはここに含まれた情報を更新することに対してどんな公約も行いません。特に別の方法で提供されなければ、Atmel製品は車載応用に対して適当ではなく、使用されるべきではありません。Atmel製品は延命または生命維持を意図した応用での部品としての使用に対して意図、認定、または保証されません。

安全重視、軍用、車載応用のお断り: Atmel製品はAtmelが提供する特別に書かれた承諾を除き、そのような製品の機能不全が著しく人に危害を加えたり死に至らしめることがかなり予期されるどんな応用(”安全重視応用”)に対しても設計されず、またそれらとの接続にも使用されません。安全重視応用は限定なしで、生命維持装置とｼｽﾃﾑ、核施設と武器ｼｽﾃﾑの操作用の装置やｼｽﾃﾑを含みます。Atmelによって軍用等級として特に明確に示される以外、Atmel製品は軍用や航空宇宙の応用や環境のために設計も意図もされていません。Atmelによって車載等級として特に明確に示される以外、Atmel製品は車載応用での使用のために設計も意図もされていません。

© *HERO* 2021.

本応用記述はAtmelのAT08569応用記述(Rev.42370A-09/2014)の翻訳日本語版です。日本語では不自然となる重複する形容表現は省略されている場合があります。日本語では難解となる表現は大幅に意訳されている部分もあります。必要に応じて一部加筆されています。頁割の変更により、原本より頁数が少なくなっています。

必要と思われる部分には( )内に英語表記や略称などを残す形で表記しています。

青字の部分はﾘﾝｸとなっています。一般的に赤字の0,1は論理0,1を表します。その他の赤字は重要な部分を表します。