最終報告書

概要

現在の mruby の処理系は CRuby 処理系と比較してメモリ消費量が少なくなるように実装されています。しかしながら、安価なマイコン環境ではメモリ (RAM) が少ない場合が多く、現行の mruby を活用することが難しいこともあります。
そこで本プロジェクトでは、mruby の ROM 活用を行い、mruby の機能を制限することなしに、RAM が小さい環境でも mruby が利用できることを目指します。

実現方法について

本プロジェクトはマイコン環境での ROM(Flash ROM) の活用に注目し、mruby の初期化時における RAM の消費量を減らす戦略を採用しました。

- a. Symbol オブジェクトの ROM 化
- b. メソッドテーブルの ROM 化
- c. 初期オブジェクト・クラスの ROM 化
- d. irep 構造体の ROM 化

a〜d については、通常は `mrb_open()` を実行した際に、`mrb_malloc()` 命令を使ってメモリを確保し、そのメモリ上に構造体を配置しています。これらの構造体を、いったん C のソースコードとしてファイルに生成し、このファイルを mruby と合わせてビルドすることで、ROM に配置するようにします。
a. については、山根が RubyKaigi 2018 で発表を行ったものを利用しています。
その時点では Symbol の候補としてソースコードをスキャンする形で行っていましたが、今回の作業では実際に nvgen 環境で生成した libmruby.a を用いて `mrb_open()` を実行し、そこで生成された Symbol を元に完全ハッシュ関数を生成するようにしています。
b. については、RubyKaigi 2018 の時点では ROM 化したメソッドテーブルへのポイ

ンタを RClass 構造体にメンバーを追加する形で実装していたため、構造体のサイズが増加してしまっていました。これに対し、本プロジェクトでは iv_tbl メンバーに入れるようにしたため、構造体のサイズには変更がない形になっています。
c.、d. は今回新しく取り組んだものになります。

ビルド環境と nvgen コマンド

本プロジェクトでは、ビルド環境を 3 種類用意しています。

- host 環境
- nvgen 環境
- presym 環境

host 環境では default.gem を使い、mrbc コマンド等を生成しています。
nvgen 環境は nvgen コマンドを生成しています。nvgen コマンドは PC(mac) 上で動作するコマンドで、`mrb_open()` を実行し、その際に生成された各構造体を走査し、その情報を元に C のソースコードを標準出力に生成します。生成されたソースコードには、メモリ上にあった構造体が C の定義文として出力されています。分量は使用する mrbgem によっても変わりますが、約 4000 行、383,000 バイトほどになります。同時に、presym.key というファイルを生成します。このファイルは gperf 用のファイルで、gperf を実行すると C のソースコードに変換されます。これが `mrb_open()` 時に作られる Symbol(pre defined symbol) に対応するものになります。
最後の presym 環境はターゲット環境で動作させる libmruby.a を生成するものです。この libmruby.a を、nvgen により生成された C ファイル、presym.c、さらにランタイム用ライブラリを合わせてビルドすると、最終的な実行ファイルが生成される、という流れになります。

実現における課題とその対応策

RAM と ROM の使い分け

現状 ROM に配置しているものは以下の通りです。

- 関数テーブル (初期化時のもの)
- mruby VM スタック

- irep
- range_edges
- mruby オブジェクト (RBasic 構造体)
- 初期 mrb_state 構造体

このうち、最後の初期 mrb_state 構造体は、実行時に ROM から RAM にコピーして利用しています。
一方、RAM 上のグローバル変数として配置しているものは以下になります。

- htable 構造体
- 関数テーブル (実行時に追加されるもの)
- iv_tbl 構造体
- mrb_context 構造体

Flash ROM や mruby が管理していない RAM への配置

構造体を mrb_malloc 等を使わずに Flash ROM や RAM に置いた場合、そのまま `mrb_free()` や `mrb_realloc()` を実行するとエラーで実行が止まります。
これを解決するため、メモリ管理ライブラリの TLSF(https://github.com/mattconte/tlsf) を導入しています。これを使い、`mrb_malloc()` 等で管理されたメモリかどうかを判別し、処理を変えています。

関数ポインタから関数名の取得

上記 b. のメソッドテーブルを生成するには、アドレスを元に C の関数への関数ポインタを取得する必要があります。これについては、nm コマンドを使い名前解決することによって実装しています。そのため、nvgen 環境のビルドには nm コマンドが必要となります。

static 関数の非 static 化

現状、mruby で使用している C の関数は、MRB_API でないものは static になっているものが少なからずあります。これからの関数は、アドレスが分かっていてもコンパイル時に構造体からリンクすることができません。
そのため、`MRB_STATIC` というマクロを作り、static にするか否かを指定できるようにしています。これを用いて、static ではない関数として、構造体からリンクできる

ようにしています。

mruby 2.0 対応

中間報告でも言及した mruby 2.0 への変更点の対応ですが、以下の 2 点については対応しました。

- Class クラス (RClass) の iv の segment list 化
- Hash クラス (RHash) の htable 化

ただし、iv の segment list については、htable と同様に各 segment のサイズを指定できるように変更しています。これを使い、初期の segment は ROM 化した際の要素数と同じサイズにしています。

なお、現在ベースにしているのは 2019/01/20 時点のもの（git のハッシュ値は `b83e4df4`）で、その後に行われている Symbol の実装の変更についてはフォローしていません。

動作検証

STM32 (ARM) を使用したボード、Nucleo F401RE での動作確認を行いました。

- MCU: STM32F401RET6 (ARM Cortex-M4)
- RAM: 96KB
- Flash ROM: 512KB

RubyKaigi 2018 では Symbol オブジェクトを ROM 化することにより、当該ボードでも起動することを確認しました。

しかしながら、こちらを mruby2.0 対応するにはデバッグ用コードを大量に仕込み過ぎていて、マージに時間がかかるため、今回比較は断念しました。

後述のメモリアロケータとして使用した TLSF による動的な RAM 使用量と、map ファイルより算出した静的な RAM・ROM 使用量は以下の通りです。

- 適用前：mrb_init_mrblib 実行時に RAM を使い切っていたため最終使用量は計測できず
- 適用後：
  - 起動時に使用した TLSF のメモリ使用量：16,332 bytes
  - TLSF 以外にグローバル変数として RAM に配置されていた使用量 (map ファ

イルより算出): 14,196 bytes
  – 参考: ROM 使用量 (map ファイルより算出) 276,368 bytes

上記より、mruby で使用している RAM は 30KB 程度で済む、という結果になりました。
もっとも、この数値には STM32 のライブラリ等で消費している部分は含まないため、ターゲットボードに必要となるメモリ使用量はこれよりも大きい値になります。

今後の課題

RBasic 内の flags への対応

RBasic 構造体内のメンバーのうち、flags については生成後に変更されることがあります。

- Object の freeze
- Class に対する Module の include

これらに対応するには、flags 部分のみを RAM に配置する等の工夫が必要になります。

他のマイコンで動作検証

現状では ARM Cortex-M4 のみで動作検証を行っています。
他のマイコン環境でも基本的には同等かと思いますが、実機を使って動作検証を行う予定です。

mruby 2.0 最新版への対応

上述の通り、2019/01/20 時点の mruby から fork しているため、それ以降の修正については別途取り込む必要があります。

GC の不具合修正

現状 GC の実行中に落ちることがあります。これは ROM 上のオブジェクト等については GC 時に処理を行わないようにする処理が抜けているところがあるためと思われます。これについては GC に ROM 上のデータを表す新しい色を追加することで解決

しようとしています。

mruby 本体の変更点についてまとめる

今回の対応は mruby 本体への修正が必要となります。

- `MRB_STATIC` を使った static 関数の非 static 化
- src/hash.c に対する Hash の segment への prefix の付与 (`segment` → `ht_segment`)
- src/symbol.c に対する perfect hash 対応（`#ifdef`〜`#endif` で場合分け）
- src/variable.c の `segment` に `uint16_t size` を追加し、サイズを個別に変更できるように修正

このうち本体に取り込んでもらえそうなものについては差分を pull request として提案する予定です。

mrbgem 化

現在はプロジェクト内のディレクトリに mruby ディレクトリを作り、nvgen コマンドの生成等を行っています。
導入を容易にするため、プロジェクト自体を mrbgem として実装することを検討します。

BOXING への対応

現状では BOXING は `MRB_NAN_BOXING`、`MRB_WORD_BOXING` ともに対応していません。
BOXING を考慮すると、どうしても初期の構造体を構築する際に課題が出てくる可能性が高いためです。
BOXING を有効にするにはポインタの処理を含めて対応する必要があります。