Porting uCLinux to DSP without gcc toolchain

高岡 正, 蓼沼 伸二
株式会社 アックス*

2007-09-13

概要

我々は 32bit DSP に uCLinux を移植したので報告する。このアーキテクチャには gcc toolchain が存在しないので、移植にあたってはカーネルのソースコードを修正することで対応した。その過程で判明した問題点と対処法、カーネルの移植性などについて考察する。

1 はじめに

Linux システム [1] の移植には、GNU C Compiler Collection[2] (以降 gcc) や GNU binutils[3] が必要とされている。 カーネルをはじめ、ライブラリやアプリケーションが gcc 以外の開発環境を考慮していないためである。そのため gcc を持たない CPU アーキテクチャへ Linux システムの移植が試みられ、報告されたことはなかった。

このたび我々は、gcc を持たない CPU アーキテクチャへ uCLinux[4] を移植したので報告する。移植にあたって、カーネルやビルドシステムへの変更が少なくなるよう考慮した。移植したカーネルのバージョンは 2.6.17.14 である。

2 背景

画像処理やマルチメディア機器など、デジタル信号処理プロセッサ (以降 DSP) を使用した組込みシステムへの要求が高機能かつ複雑になってきている。特にマルチタスクやネットワーク通信、ファイル処理などが増えており、このような分野は Linux が得意とするところである。

一方で DSP も高性能化がはかられて、32bit アーキテクチャとなり、プログラムを C 言語で記述することが普通になってきた。したがって、DSP に Linux を移植できれば、高度な DSP システムの開発が容易になるのではないかと考えた[*1]。

2.1 ターゲット・システム

今回移植を行ったターゲットは、テキサス・インスツルメンツ社 (以降 TI 社) の DSP である `c64x+` アーキテクチャ [7] で、以下を特徴とする。

- 64 個の 32bit 汎用レジスタ
- ループ展開用の専用レジスタ群
- 6 個の整数 ALU と 2 個の乗算器
- 2 個のロード・ストア・ユニット
- 2 個のブランチ・ユニット
- 最大 8 命令同時実行の可変長 VLIW
- 最大動作周波数 1GHz で低消費電力
- Gigabit Ethernet などを内蔵した SoC

2.2 最初の試み

当初は `c64x+` アーキテクチャに、gcc および binutils の移植を行い、その後 Linux の移植を行う方針であったが、`c64x+` アーキテクチャの

- 命令の並列性が高い (最大 8 命令同時実行)
- ディレイ・スロットが深い (ロード時 4 クロック、ブランチ時 5 クロックなど)
- パイプラインのハードウェア・インターロックがない

* `http://www.axe-inc.co.jp/`

[*1] 株式会社ヒカリ [13] と共同開発。

```
int func (int *a, int *b, int c) {
    return *a + *b + (c < 0 ? -1 : 1);
}
TEXT Section .text (Little Endian), 0x20 bytes at 0x0
00000000             .text:
00000000             _func:                          ; A4=a B4=b A6=c B3=link
00000000       104d           LDW.D2T2   *B4[0],B4 ; fetch *b
00000002       004c ||        LDW.D1T1   *A4[0],A4 ; fetch *a simulteniously
00000004       41ef           BNOP.S2    B3,2      ; jump to caller, 2cycle nop
00000006       fda6           MVK.L1     -1,A3     ; A3=-1
00000008   001818da ||        CMPGT.L2X  0,A6,B0   ; B0=(c<0)
                                                    ; B4,A4 are loaded
0000000c       9240           ADD.L1X    A4,B4,A4  ; A4=*a+*b
0000000e       e9ee || [!B0]  MVK.S1     1,A3      ; A3=1 if B0=0
00000010   02106078           ADD.L1     A3,A4,A4  ; A4=return value
                                                    ; branch B3 occured
00000014   00000000           NOP
00000018   00000000           NOP                  ; fetch packet header
0000001c   e1600049           .fphead   n, l, W, BU, nobr, nosat, 0001011
```

図 1　c64x+ アーキテクチャ

などの特徴 (図 1) が、gcc や binutils などの移植を困難にすることがわかった。

また開発期間が短く、開発環境と Linux の移植を直列に行うことは難しいため、gcc を使用しないで Linux の移植を試みることを決断した。

3　問題点

移植前に予想できた問題点には以下のようなものがある。

3.1　コンパイラ

コパイラには TI 社の開発環境である Code Composer Studio (以下 TICCS)[8] を使用することになる。この C コンパイラが受け付ける文法は C89(ANSI X3.159-1989) 相当であり、C99(ISO/IEC 9899:1999)[9] 相当である gcc 向けに書かれたコードがそのままではコンパイルできないことがおこる。

また、TICCS のコンパイラは高度な最適化を行うのであるが、これが問題をおこすこととなった。さらに、使用されているオブジェクト形式は COFF 形式であって gcc で標準的な ELF 形式と異なり、これも問題となる。

3.2　ソースコード

Linux カーネルや、uClibc[5]、busybox[6] などのソースコードは、gcc の持つ C 言語の拡張機能 [2] に強く依存した形で書かれている。

プリプロセッサのみ gcc を使用するとか、gcc の拡張機能を TICSS が理解できる形に前処理する方法も考えられるが、開発期間を考えて断念した。

今回はカーネルなどのソースコードの方を書き換える方法をとった。ただし、バージョン・アップにを追従できることを考慮し、書き換えは機械的かつ少なくなるように工夫した。

3.3　ビルドシステム

ここでのビルドシステムとは、Makefile の内容やコンフィギュレーションの仕組をさす。カーネルなどのビルドシステムは、Linux 上で開発されることを前提としており、さらに gcc や binutils 以外の開発環境の利用を想定していない。

一方で TICCS は Windows 上で動作する DOS アプリケーションである[*2]。そこで Cygwin 環境を導入してビルド環境を Linux に似せて、ビルドシステムへの変更が少なくなるように工夫した。

また、Makefile に記述されているコンパイラやリ

[*2] Linux で動作する TICCS もある。

ンカのオプションは gcc 向けで、TICCS 向けに読み替えが必要である。Makefile の記述を変更せず、途中にシェルスクリプトを介在させてオプションの読み替えなどを行い、ビルドシステムへの変更が少なくなるように工夫した。

以降では、移植中に遭遇した各問題点の詳細を述べ、我々がとった解決方法について解説する。

4 ターゲット固有の問題点

Linux システムを移植するターゲットのアーキテクチャや開発環境に由来する問題には以下のものがあった。

4.1 高度な最適化

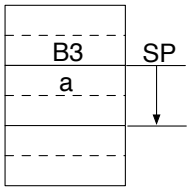
TICCS は高度な最適化を行う。ポインタ演算の結果が 32bit をオーバーフローしたことを検出するようなコードは、アセンブル結果のように完全に最適化されて取り除かれてしまう (図 2)。このようなコードが uClibc にあり (`libc/string/generic/strnlen.c`)、逆アセンブル結果を注意深く見ることによって発見した。

```
int optimized (char *p, unsigned n) {
    return (p + n < p) ? 1 : 0;
}
_optimized:
   BNOP.S2    B3,4   ; return, 4 nop
   MVK.L1     0,A4   ; A4=return value
```

図 2　高度な最適化

4.2 スタックのレイアウト

スタックは、ほとんどの CPU ではプレ・デクリメントで動作するが、`c64x+` ではポスト・デクリメントで動作する (図 3)。さらにスタックポインタには 8 バイトのアラインメントが要求されるため、スタック最上部の 4 バイトは未使用となる。したがってスタックポインタを初期化する値には注意が必要であった。




```
void callee (int *);
void caller (int a) {
    callee(&a);
}
_caller:                    ; A4=a
   STW.D2T2   B3,*SP--(8) ; save B3
   STW.D2T1   A4,*+SP(4)  ; save a
   ADD.L1X    4,SP,A4     ; load &a
|| CALLP.S2   _callee,B3  ; B3=link reg.
   LDW.D2T2   *++SP(8),B3 ; restore B3
   NOP        4           ; delay slot
   BNOP.S2    B3,5        ; return,5 nop
```

図 3　スタックのレイアウト

4.3 オブジェクト形式

TICCS が出力するオブジェクトは COFF 形式 [10] であるが、gcc が出力するオブジェクト形式は ELF[11] であることが多い。このオブジェクト形式の違いにより、以下のような問題が起こった。

4.3.1 `coff2flat`

uClinux ではユーザプログラムの実行形式に、サイズが小さい FLAT[12] 形式が使われることが多く、COFF 形式を実行するコードがない。

対処方法として、カーネルが COFF 形式を直接読み込めるようする方法と、コンパイル時に COFF 形式を FLAT 形式に変換し、カーネルは FLAT 形式を使用する方法が考えられる。

カーネルの移植と独立して開発が進められることから、実行形式として FLAT 形式を選択し、`coff2flat` コマンドを開発した。

4.3.2 32bit 即値

FLAT 形式は 32bit 即値に対してのみ再配置情報を持つ。32bit 即値が扱えない CPU の場合はコンパイラが 32bit 定数をデータとして生成し、PC 相対で読み出すことで対処されている。

TICCS には 32bit 即値を定数データとして生成せず、上位と下位の 16bit 即値を読み込む 2 つの命令が生成される。また 2 つの命令は最適化により必ずしも連続して実行されるわけではない (図 4)。

そこで FLAT 形式の再配置情報の種類に、下位 16bit、上位 16bit、上位 16bit の補足、の 3 つを追加した。上位 16bit の再配置計算では下位から桁上

```
extern int a[], b[];
int add (int x, int y) {
    return a[x] + b[y];
}
_add:
   MVKL.S2   _b,B5        ; lo(b)
|| MVKL.S1   _a,A3        ; lo(a)
   MVKH.S2   _b,B5        ; hi(b)
|| MVKH.S1   _a,A3        ; hi(a)
   BNOP.S2   B3,4         ; return, 4 nop
|| LDW.D2T2  *+B5[B4],B4 ; B4=b[y]
|| LDW.D1T1  *+A3[A4],A3 ; A3=a[x]
   ADD.L1X   B4,A3,A4     ; A4=return val
```

図 4　32bit 即値

がりの可能性があるが、前述のように対応する下位の値を探すのが困難である。そこで対応する下位 16bit の値を上位 16bit の補足として再配置情報に追加することで対処した。

4.3.3　.cinit セクション

TICCS が生成するオブジェクトコードには、初期値つきデータを格納するいわゆる .data セクションがない。すべてのデータ領域は初期値ゼロを持つ .far セクションに割り当てられ、.cinit セクションに初期値のテーブルが生成される (図 5)。

```
int val = 0x12345678;
char str[] = "ab";
  .sect   ".cinit"
  .field  4,32           ; sizeof(val)
  .field  _val,32        ; &val
  .field  0x12345678,32 ; 32bit value
  .field  3,32           ; sizeof(str)
  .field  _str,32        ; &str[0]
  .field  97,8           ; 'a'
  .field  98,8           ; 'b'
  .field  0,8            ; '\0'
_val:  .usect  ".far",4,4
_str:  .usect  ".far",3,8
```

図 5　初期値付きデータ

FLAT 形式に .cinit セクションはないので、すべての .far セクションを .data セクションと見なし、.cinit セクションから初期値を埋め込んで対処した。プログラムによっては初期値ゼロの .data セクションが大きくなるが、圧縮 FLAT 形式を実装してサイズの増大を押さえた。

4.3.4　weak alias

ELF 形式には、リンク時に同じシンボルがあった場合、弱い方を無視して強い方をリンクに使用する、weak alias と呼ぶ仕組みがある。

しかし TICCS のオブジェクト形式は COFF であり、weak alias の機能はない。そこで weak alias を単純に同じ値を持つ別のシンボルとして実装した。しかし、このままリンクを行うと二重定義シンボルのエラーになる。これには以下のように個別に対応を行った。

カーネルの構成の違いにより実装されないシステムコールがある。この未実装のシステムコールを weak alias で実現している (kerne/sys_ni.c の cond_syscall)。これには CONFIG_ マクロによる条件コンパイルで二重定義を防いで対処した。

また uClibc では、pthread ライブラリを使用した場合、キャンセル・ポイントの処理のために置き換える必要のあるシステムコールが weak alias で実装されている。これには、weak alias によるシンボルを含まないライブラリを別途用意し、リンク時に使い分けることで対処した。

5　コードの問題点

カーネルやライブラリなどのコードのうち、gcc の拡張機能に依存している部分は、コンパイルを行って発生するエラーで検出した。以下に遭遇した主な問題とその解決方法を解説する。

5.1　long 型

カーネルのコードには C 言語の型として long がそのままの形で記述されている。コードの意味をみる限り、32bit か 64bit のどちらかと仮定している。これは int 型が 16bit, long 型が 32bit であった時代の名残りではないかと思われる。

ところが c64x+ アーキテクチャでは long の精度は 40bit、sizeof(long)=8 であって、コードが仮定している前提と整合性がとれない。

以下に記述するように、コードを書き換えて対処を行ったが、本来は意味に応じて typedef した型、あるいは uint32_t などのようにサイズを明示した

型が使われるべきかと考える。

5.1.1 ビットマップ

メモリ管理やファイルシステムなどで各種ビットマップが使われているが、この部分のコードには long がそのまま使用されており、BITS_PER_LONG 定数が 32 か 64 かで条件コンパイルされるようになっている (linux/bitmap.h など)。別途 bitmap_t という型を typedef し、関連すると思われる long の記述をすべて書き換えることで対処した。

5.1.2 ハッシュ関数

ハッシュ関数の値も long と宣言されている。また後述のようにハッシュ計算用の定数には'UL' 修飾子がういており、40bit 定数になってしまう (linux/hash.h など)。ハッシュ関数の型を明示的な符号なし 32bit 整数 __u32 に変更した。

5.1.3 ポインタ整数

システムコールの戻り値など、ポインタが返されるが整数型としたい場合、long あるいは unsigned long 型と書かれている (mm/mmap.c の sys_brk() など多数)。C99 ではこの目的に、intptr_t あるいは uintptr_t が定義されているので、該当する型にすべて書き換えた。

5.1.4 割り込みフラグ

spin_lock_irqsave() マクロなどで、割り込みフラグの保存変数が明示的に long 型と宣言されている。割り込みフラグを表す型はアーキテクチャに依存するので、irqflags_t なる型を typedef し、該当する部分をすべて書き換えた。

5.1.5 jiffies

カーネル内部の時間変数である jiffies が unsigned long 型と宣言されている (linux/jiffies.h)。これは jiffy_t 型を typedef し、すべて書き換えた。

また time_t や struct timespec などで使われる long も、符号なし 32bit 型の __u32 にすべて書き換えた (linux/time.h など)。

5.1.6 L 接尾子

ビットマップや時間を表す定数に、long 型だと明示するために'L' や'UL' などの接尾子が付けられている。定数の値や使用される式によっては、40bit 演算のコードが出力されることがあった。定数の定義から'L' を取り除くことで対処した。

5.1.7 printk()

アドレス値など long 型を表示する printk() の書式文字列には修飾文字'l' が使われる。すべての書式文字列を書き換えるのは非現実的なので、printk() の実装の方で'l' 修飾文字を無視することで対処した (lib/vsprintf.c)。

5.2 三項演算子

カーネルでは、gcc で拡張された三項演算子が使われている (fs/buffer.c, net/core/stream.c, net/ipv4/route.c など)。図 6 のように等価な if 文に書き換えて対処した。

```
invalidatepage =
 page->mapping->a_ops->invalidatepage ?:
 block_invalidatepage;
err = sock_error(sk) > : -EPIPE;
return est_mtu ? : new_mtu;

if (!(invalidatepage =
  page->mapping->a_ops->invalidatepage))
  invalidatepage = block_invalidatepage;
if (!(err = sock_error(sk)) err=-EPIPE;
if (!est_mtu) return est_mtu
     else return new_mtu;
```

図 6　拡張された三項演算子

5.3 空の構造体

```
#ifdef __GNUC__
typedef struct { } raw_rwlock_t;
#define __RAW_RW_LOCK_UNLOCKED { }
#else
typedef struct {
  int dummy; /* dummy member */
} raw_rwlock_t;
#define __RAW_RW_LOCK_UNLOCKED { 1 }
#endif
```

図 7　空の構造体

カーネルにはメンバを持たない空の構造体が使われている (linux/spinlock_types_up.h)。gcc 以外のコンパイラでは文法エラーとなる。図 7 のようにダミーのメンバを追加して対処した。

5.4 void ポインタ演算

カーネルでは、型を特定できないポインタに void *型を多用している。一方でそのポインタにアドレス演算が必要になることも多い。gcc は void *型に対する演算が拡張されており、char *型と見なして演算が行われる (net/core/iovec.c など)。

図 8 ように等価な演算になるようにキャスト演算子を適宜追加することで対処した。

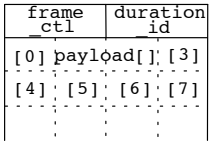
```
struct iovec {
  void __user *iov_base;
  __kernel_size_t iov_len;
} *iov;
int __user *base;
int copy, offset;
base = iov->iov_base + offset;
iov->iov_base += copy;

base = (int *)
    ((char *)iov->iov_base + offset);
iov->iov_base =
    (char *)iov->iov_base + copy;
```

図 8　void ポインタ演算

5.5 長さ 0 の配列

カーネルでは、固定長のヘッダ部分とその直後に続く可変長の配列というデータ構造を扱うことが多い。gcc は配列の長さの解釈が拡張されており、図 9 のように可変長部分を長さ 0 の配列として宣言でき、また簡潔に参照できる (net/ieee80211.h)。



これを payload[1] と宣言とする方法もあるが、構造体の sizeof() が変化するうえに影響の検出が困難である。図 9 のように構造体の直後を指定した型で参照する __endof() マクロを定義し、可変長配列への参照を機械的に書き換えることで対処した。

5.6 packed 属性

gcc は図 10 のように、構造体に packed 属性を指定し、メンバのアラインメント要求を無視して詰め込むという型宣言ができる (include/linux/mtd/cfi.h)。

```
struct ieee80211_hdr {
  __le16 frame_ctrl;
  __le16 duration_id;
  u8 payload[0];
} __attribute__ ((packed));
struct ieee80211_hdr *packet;
packet->payload[4] = 0;

#define __endof(ptr, type) \
      (((type)  *)((ptr) + 1))
__endof(packet, u8)[4] = 0;
```

図 9　長さ 0 の配列

```
struct cfi_ident {
  uint8_t  qry[3];
  uint16_t P_ID;
  uint16_t P_ADR;
  ...
} __attribute__((packed));
```

図 10　packed 属性

TICCS には packed 属性も対応する #pragma もなく、機械的な移植は困難である。個別の対処が必要であった。

drivers/mtd/chips/chipreg.c では、フラッシュ ROM の CFI 情報を読み出す部分で問題が発生したが、内部データ構造に変換する部分のみの変更で対処できた。

fs/cifs/cifspdu.h では、packed 構造体のメンバが広範囲で参照されており、対処が難しい。ネットワーク通信が少数の関数に集約されており、構造体と packed なネットワークパケットとの間で相互変換を行うことで対処可能かと考える。

アラインメントの問題が予想される場合には、構造体をテンプレートとせず、バイト配列とアクセス関数による設計とすべきではないかと考える。

5.7 構造体の初期化

構造体の初期化で、図 11 のようにメンバ名を指定した代入が使用されている (net/sunrpc/pmap_clnt.c など)。構造体の宣言と同じなるように、メンバの初期値を並べ替えて対処した。

gcc ではこの初期化が局所変数でも使え、さらに

実行文の値でも初期化できる。明示的に初期化されていないメンバは値ゼロになることに注意が必要で、図 11 のように書き換えて対処した。

linux/wait.h ファイルの DEFINE_WAIT_INIT では current の参照が関数呼び出しを伴う実行文であるために問題が発生した。宣言と初期化を分離することで対処した。

```
struct rpc_message {
  struct rpc_proc_info *rpc_proc;
  void                 *rpc_argp;
  void                 *rpc_resp;
  struct rpc_cred      *rpc_cred;
};
struct rpc_message msg = {
#ifdef __GNUC__
  .rpc_argp = path,
  .rpc_resp = &result;
#else
  NULL,       /* rpc_proc */
  path,       /* rpc_argp */
  &result,    /* rpc_resp */
  NULL,       /* rpc_cred */
#endif
};

struct rpc_message msg;
memset(&msg, 0, sizeof(msg));
msg.rpc_argp = path;
msg.rpc_resp = &result;
```

図 11　構造体の初期化

5.8　可変長引数のマクロ

可変長引数のマクロ関数が図 12 のようにデバッグ用などで使われている (include/linux/kernel.h)。実引数の個数に合わせて複数のマクロ関数を定義し、書き換えて対処した。

```
#define pr_debug(fmt, arg...)    \
  printk(KERN_DEBUG fmt, #arg)

#define pr_debug(fmt)            \
  printk(KERN_DEBUG fmt)
...
#define pr_debug3(fmt, a1,a2,a3) \
  printk(KERN_DEBUG fmt, a1,a2,a3)
```

図 12　可変長引数のマクロ関数

5.9　式文とマクロ

gcc には任意のブロックを式とする文式という拡張がある。ほとんどは inline 関数に書き換えて対処できるが、図 13 のようにマクロの引数を左辺値に使用するものがある (asm/uaccess.h)。またマクロ引数の参照が複数回あるので、実引数に副作用を伴う式があると問題が発生する。個別に判断を行って書き換えた。

```
#define get_user(x, ptr)                \
({ typeof(ptr) _p = (ptr);              \
   int __e;                             \
   __e = access_ok(__p, sizeof(*__p)); \
   if ((__e) != 0)                      \
     (x) = *_p;                         \
   __e; })

#define get_user(type, x, ptr)          \
  access_ok((ptr), sizeof(type)) ?      \
   ((x) = *((type *)(ptr)), 0) :        \
   (-EFAULT)
```

図 13　式文とマクロ

```
unsigned long uaddr;
int curval;
ret = get_user(curval, (int *)uaddr);

char *optval;
int val;
if (get_user(val, (int *)optval))
  return -EFAULT;
```

図 14　なんのためか

ところが get_user() の参照は ptr の型で決まるため、図 14 のようにキャストを施している例が見受けられる (kernel/futex.c, net/core/socket.c など)。アーキテクチャによっては参照するデータのサイズを判断して最適化している (asm-arm/uaccess.h など)。get_user32() などのようにサイズ毎に inline 関数を用意すれば文式もマクロ関数も不要になる。

5.10　typeof とマクロ

gcc で追加された typeof 演算子を使うと、図 15 のような型独立なマクロが書ける (linux/list.

```
#ifdef __GNUC__
#define list_for_each_entry(pos, head, member)                    \
  for (pos = list_entry((head)->next, typeof(*pos), member);      \
       prefetch(pos->member.next), &pos->member != (head);        \
       pos = list_entry(pos->member.next, typeof(*pos), member))
#else
#define list_for_each_entry(type, pos, head, member)              \
  for (pos = list_entry((head)->next, type, member);              \
       prefetch(pos->member.next), &pos->member != (head);        \
       pos = list_entry(pos->member.next, type, member))
#endif
```

図 15　typeof とマクロ

h)。引数に型名を追加したマクロに変更し、参照部分をすべて書き換えて対処した。

5.11　alloca

gcc にはスタック上に動的に可変長のメモリを確保する alloca 関数が、コンパイラの組み込み関数として拡張されている。uClibc や busybox には alloca 関数を使用するコードが存在している。

TICCS には alloca 関数はなく、またコンパイラのサポートなしでは実装できないので、コードの意味をよく検討しながら malloc と free の組に置き換えて対処した[*3]。

5.12　出力セクションの指定

カーネルの初期化に関するコードやデータは、通常の .text や .data セクションではなく、異なるセクション .init.text や .init.data に配置される。これらのセクションは初期化終了後には解放されて再利用されるため、通常のセクションとは異なった位置にリンクするためである。

またカーネルのサブシステムの初期化関数へのポインタは、複数の .initcall⟨level⟩.init セクションに集めてテーブルとされる。このテーブルをたどって初期化関数を呼び出すことで、カーネルの構成が変わっても、初期化が確実に行われるよう工夫されている。

これらの機能は、図 16 のように gcc の拡張機能である __attribute__ で出力セクションを指定することで実装されている (linux/init.h)。カーネルへの引数の解釈を行う __setup(str,fn) にも同様の仕組みが使われている。

TICCS では出力セクションの変更に #pragma を使用する。ところが #pragma はプリプロセッサで処理されるので、マクロ展開で使用してもうまく働かない。C99 にはこの目的で _Pragma() が規格化されているが TICCS にはない[*4]。

そこでマクロでは @pragma にいったん変換し、プリプロセッサ処理だけを行ったソースファイルを再度検査して、出現する @pragama を #pragma に置き換えた後、再度コンパイルを行うという方法をとった (図 17)。

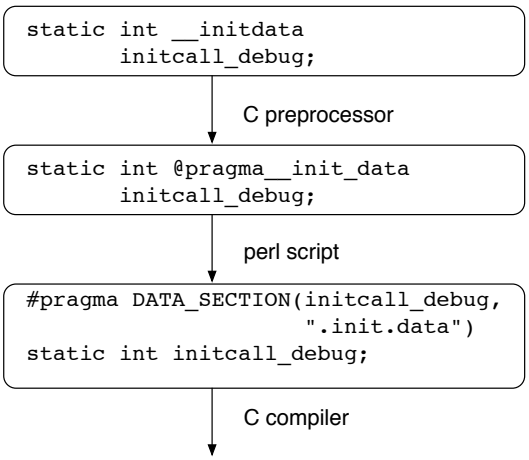


図 17　pragma の処理

[*3] malloc を使った alloca の実装もある。

[*4] 最近の TICCS には実装されている。

```
#ifdef __GNUC__
#define __define_initcall(level,fn)                                    \
    static initcall_t __initcall_##fn __attribute_used__               \
    __attribute__((__section__(".initcall" level ".init"))) = fn
#elif defined(__TICCS__)
#define __define_initcall(level,fn)                                    \
    @pragma DATA_SECTION(__initcall_##fn, ".initcall" level ".init") \
    const initcall_t __initcall_##fn = fn
#endif
#define core_initcall(fn)    __define_initcall("1", fn)
...
#define device_initcall(fn)  __define_initcall("6", fn)
```

図 16　出力セクションの指定

6　ビルドの問題点

カーネルや uClibc などの移植にあたっては、ソースコードは適宜変更するが、Makefile などのビルドシステムの変更は少なくなるように工夫した。

6.1　シンボリックリンク

Cygwin 環境下でコンパイルを行う場合、シンボリックリンクの扱いが問題になる。TICCS はシンボリックリンクも、Windows のショートカットも理解しない。`ln` コマンドに代えて `winln` スクリプトを作成し、引数を解釈して適宜ファイルをコピーしてしまうことで対処した。

6.2　ar コマンド

binutils の ar コマンドには、次のような機能があり、カーネルや uCLibc のビルドシステムで使用されている。それぞれ ar スクリプトで機能を実装して対処した。

- `ar rcs obj.o` とすると、空のアーカイブファイル `obj.o` が生成されるが、ld コマンドはこれを空のオブジェクトファイルとして扱う。`/dev/null` をソースファイルとしてコンパイラを起動し、空のオブジェクトファイルを生成することで対処した。
- `ar dN 2 lib.a obj...` とすると、指定されたオブジェクトファイル中に同じ名前のものが複数ある場合、アーカイブからそのオブジェクトファイルを 1 つを残して削除する。

6.3　gcc コマンド

gcc の代わりに TICCS を使えるようにするために、何らかの変換プログラムを作成し、ビルドシステムへの影響を少なくする工夫が必要である。gcc のコンパイラ・ドライバとプリプロセッサを移植し、TICCS を中間パスとして使用する方法も考えられるが、開発期間を考えてシェルスクリプトで実装することとした。gcc スクリプトでの主な処理は以下の通りである。

- オプションの読み替え。gcc のオプションを読み替えて TICCS のオプションに変換する。
- `-Mp,-MD` によるヘッダファイルの依存関係の出力。TICCS も同様の機能を持つが、出力ファイルの形式が異なる。スクリプト中で出力結果を書き換えた。
- `#pragma` の処理。前述のようにソースプログラム中の `@pragma` を書き換えて、再度コンパイラを起動して処理する。
- 標準入力のコンパイル。標準入力を一時ファイルに書き込むことで対処した。

6.4　ld コマンド

binutils の ld コマンドは、リンクスクリプトで指定されたセクションの併合を行うとき、コマンドラインで指定されたオブジェクトファイルと同じ順番で出力ファイルを作成するという特徴がある (図 18)。

ところが TICCS のリンカは順序を保存しな

い。通常のセクションでは問題にならないが、.initcall.init セクションは初期化関数の順序が重要であり、コマンドラインでのオブジェクトファイルの並び順に初期化されることが必要であった。

TICCS のリンカでも、リンクスクリプトにワイルドカード指定 * を使わず、ファイル名を明示的に指示すれば、リンクスクリプトでの指定順で出力される。ld スクリプトでは、コマンドラインで指定されたオブジェクトファイルを調査し、ファイル名を明示したリンクスクリプトを動的に作成することで対処した。

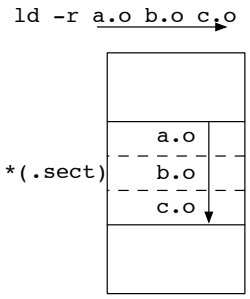


図 18　リンクされる順番

7　まとめ

移植対象のアーキテクチャの特徴による問題点、使用できる開発環境による問題点を表 1 にまとめた。

表 1　ターゲット固有の問題点

問題点	原因、改善案
高度な最適化	コンパイラの最適化の行き過ぎによりコードが消える。
スタックレイアウト	スタックの成長方向、アラインメントなどに関する暗黙の仮定。
オブジェクト形式	ELF 形式が使えず、COFF 形式の場合もある。COFF 形式に weak alias 機能はない。 FLAT 形式を使用するかカーネルが COFF 形式を実行できるように変更する。 アーキテクチャによっては再配置情報などに工夫が必要。

Linux カーネルや uClibc などのコーディング方法による問題点、gcc の拡張機能への依存による問題点を表 2 にまとめた。

表 2　コードの問題点

問題点	原因と改善案
long 型	移植性がないコードが原因。整数型の bit 数に関する仮定。使用目的に応じて型名を導入する。
三項演算子	gcc 拡張。if 文で記述する。
空の構造体	gcc 拡張。
void ポインタ演算	gcc 拡張。明示的なキャスト演算を使用する。
長さ 0 の配列	gcc 拡張。明示的にアドレス計算する。
packed 属性	gcc 拡張。構造体をテンプレートに使用しない。
構造体の初期化	C99 機能。実行文による初期化は、明示的に代入文で行う。
可変長引数のマクロ	C99 機能。
式文とマクロ	gcc 拡張。inline 関数を使用したコードに書き換える。
typeof とマクロ	gcc 拡張。明示的に型名をマクロの引数に与える。異なった名前の inline 関数に分ける。
alloca	gcc 拡張。malloc と free などに置き換える。
出力セクションの指定	gcc 拡張。_Pragma() を使用する。特に Linux カーネルの初期化方法に注意が必要。

ビルドシステムのオペレーティングシステムに関する仮定による問題点、また binutils コマンド群の動作への依存による問題点を表 3 にまとめた。

表 3　ビルドの問題点

問題点	原因と改善案
シンボリックリンク	Cygwin 環境と Windows 環境での仕様の違い。パス名形式や改行コードの違いなどもある。
ar コマンド	スクリプトを介在させてオプションなどを読み替える。
gcc コマンド	スクリプトを介在させてオプションなどを読み替える。
ld コマンド	オブジェクトのリンク順序が保存されることを仮定している。リンカスクリプトを動的生成して対処。

8　DSP で Linux

以上のようにして移植した Linux システムの起動時のメッセージを図 20 に示す。動作しているサブシステムには、IPv4, NFS, MTD, jffs2 などがあり、一通りの開発ができるようになっている。

図 19 は工業用カメラを接続できるターゲットシステム[*5]で、動作周波数 1GHz の DSP、Gigabit Ethernet、NAND Flash のファイルシステムを持つ。DSP で Linux であるとともに、高性能な組込みマイコンシステムと考えることもできる。

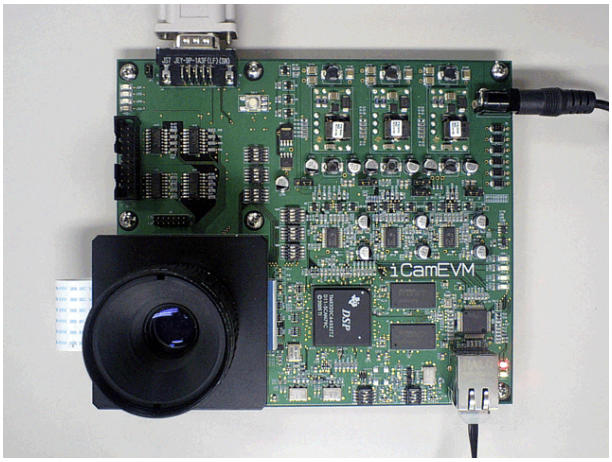


図 19　ターゲットボード

9　おわりに

ソースコードを機械的に書き換え、スクリプト言語を活用することにより、gcc を持たないアーキテクチャへの Linux の移植を、短期間で成功させることができた。またこの変更を通して、Linux カーネル・コードが持つ移植性の問題が明らかになった。

変更した量はパッチファイルの行数にして、カーネルが約 19 万行、uClibc と busybox がそれぞれ約 1 万行強であるが、ほとんどは機械的な変更である。作成したスクリプトファイルは全部で 2 千行ほどであった。

[*5] 株式会社ヒカリおよび株式会社アックスより開発キットのリリースが予定されている。

参考文献

[1] *The Linux Kernel.* http://www.kernel.org/

[2] *Using the GNU Compiler Collection.* http://gcc.gnu.org/onlinedocs/gcc-4.2.0/gcc/C-Extensions.html

[3] *GNU Binutils.* http://www.gnu.org/software/binutils/

[4] *Embedded Linux/Microcontroller Project.* http://www.uclinux.org/

[5] *A C library for embedded Linux.* http://www.uclibc.org/

[6] *BusyBox.* http://www.busybox.net/

[7] *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide.* http://www.ti.com/litv/pdf/spru732c

[8] *TMS320C6000 Optimizing Compiler.* http://www.ti.com/litv/pdf/spru187n

[9] Samuel P. Harbison III, Guy L. Steele Jr., *C - A Reference Manual 5th ed.* (Prentice Hall, 2002)

[10] *TMS320C6000 Assembly Language Tools.* http://www.ti.com/litv/pdf/spru186p

[11] *Executable and Linking Format (ELF) Specification.* http://x86.ddj.com/ftp/manuals/tools/elf.pdf

[12] *uClinux - BFLT Binary Flat Format.* http://www.beyondlogic.org/uClinux/bflt.htm

[13] **株式会社ヒカリ** *iCamEVM.* http://www.hikari-net.co.jp/products/products.html

```
loading '/mnt/vmlinux.bin' (2762072 bytes) at 0xe0000000
.....................................done
booting from 0xe0000000
Linux version 2.6.17.14-uc1 (t2@goccha.axe-inc.co.jp) (TMS320C6x C/C++ Compiler v6.0.11) #1
 Fri Jul 6 09:54:39 JST 2007
uClinux/TMS320C6455
Hikari iCamEVM support by AXE, Inc.
L1P Cache 32KB, L1D Cache 32KB, L2 Cache 256KB
  e0000000-effffff cached
On node 0 totalpages: 65536
  Normal zone: 65536 pages, LIFO batch:15
Built 1 zonelists
Kernel command line: root=/dev/ram debug
PID hash table entries: 2048 (order: 11, 8192 bytes)
Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)
Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory available: 256920k/260192k RAM, (1946k kernel code, 678k data, 512k init)
Calibrating delay loop... 1998.84 BogoMIPS (lpj=9994240)
Mount-cache hash table entries: 512
NET: Registered protocol family 16
NET: Registered protocol family 2
IP route cache hash table entries: 2048 (order: 1, 8192 bytes)
TCP established hash table entries: 8192 (order: 4, 65536 bytes)
TCP bind hash table entries: 4096 (order: 3, 32768 bytes)
TCP: Hash tables configured (established 8192 bind 4096)
TCP reno registered
JFFS2 version 2.2. (NAND) (C) 2001-2003 Red Hat, Inc.
Registering unionfs 1.3
unionfs: debugging is not enabled
io scheduler noop registered (default)
bfibis5: major number 241
bfibis5: initialize: HPI, LUT, FPGA, Camera, done.
BlueFinder RS-232C driver initialized
ttyS0 at MMIO 0xa0050000 (irq = 4) is a bf232
ttyS0: interrupt rx=4 tx=5
RAMDISK driver initialized: 4 RAM disks of 8192K size 1024 blocksize
TMS320C6455 GMAC: MAC address is 00:08:c4:00:01:03
TMS320C6455 GMAC Linux version updated 4.0
TMS320C6455 GMAC: Installed 1 instances.
NAND device: Manufacturer ID: 0x20, Chip ID: 0xf1 (ST Micro NAND 128MiB 3,3V 8-bit)
Scanning device for bad blocks
Bad eraseblock 1 at 0x00020000
Creating 1 MTD partitions on "iCamEVM NAND":
0x00000000-0x08000000 : "NAND Flash"
boot flash device: 00100000 at b0000000
Boot Flash: Found 1 x16 devices at 0x0 in 8-bit bank
 Amd/Fujitsu Extended Query Table at 0x0040
Boot Flash: CFI does not contain boot bank location. Assuming top.
number of CFI chips: 1
cfi_cmdset_0002: Disabling erase-suspend-program due to code brokenness.
Creating 5 MTD partitions on "Boot Flash":
0x00000000-0x00004000 : "boot1"
0x00008000-0x00010000 : "boot2"
0x00010000-0x00100000 : "miniloader"
0x00004000-0x00006000 : "param1"
0x00006000-0x00008000 : "param2"
TCP bic registered
NET: Registered protocol family 1
NET: Registered protocol family 17
RAMDISK: cramfs filesystem found at block 0
RAMDISK: Loading 436KiB [1 disk] into ram disk... done.
VFS: Mounted root (cramfs filesystem) readonly.
Freeing unused kernel memory: 508k freed (0xe01ea000 - 0xe0268000)
...
```

図 20　ブートメッセージ