より良い開発環境を自らの経験から

日本電気株式会社 マイクロコンピュータ事業部システム部
沼田 賢治／立原 裕司

■ 序

ZIPC と NEC ツールが連携を始めてから 4 年の月日が経ちます。ZIPC Watchers Vol.3 にもなると、愛読者の方は「ツールの連携の話はもういいよ」と、少し食傷気味かと思いますが、はじめての方のために簡単に経緯をお話します。

それまで弊社で提供してきたツールは、作ってからバグを出す、言わば後攻めが中心でした。1995 年、携帯電話端末が脚光を浴び始め、インターネットを中心に、JPEG/MPEG、デジタル放送、セットトップボックス…時代はデジタル化に向かい、企業間での主導権争いや標準化が活発化していました。半導体デバイスの集積度が飛躍的に向上し、CPU パワーとメモリ容量は拡大し、コストパフォーマンスは向上しました。

それを背景に、ますます高機能化する一方ライフサイクルが短くなる製品開発に対して、ソフトウェアの資産化と品質が重視されるだろうということは明らかでした。コンパイラ性能の向上やリアルタイムOSは、資源活用や性能向上、リアルタイム設計の支援を行います。しかし、品質に対してはもっと根本的なアプローチが必要でした。

私たちが開発しているソフトウェアは組み込みシステムではなく、開発を支援するツールです。用途や性質は異りますが、不具合の元となる真の原因を突き詰めていくと、組み込みシステムのそれと同じだということに気が付きます。それは”仕様設計の曖昧さ”です。この解決策として、仕様記述の統一やレビューの徹底が行われています。それは確かに効果をもたらしますが、設計手法に標準がなかったり、設計とソースが連動していないという問題が残ります。

組み込みシステム開発においては、この仕様記述の曖昧さを排除するために状態遷移表を書く場合が多いということは、以前より注目していました。そして、ZIPC は目的にあったツール

でした。ZIPC 上で記述した仕様は、プログラマの共通の資産とすることができると映ったのです。

第一期連携で実現した機能は、以下の通りです。

- システム・シミュレータの入出力パネルと状態遷移表を結合して、ビジュアル・イメージで状態遷移表の動作検証を行えるようにした。
- 状態遷移表から生成するコードを NEC マイコン用に最適化し、L/E で問題となるコード効率を上げた。
- 状態遷移表とシステム・シミュレータ／インサーキット・エミュレータを結合し、状態遷移表からブレークをかける、状態遷移表レベルのトレースを取ることを可能にした。

仕様設計を十分に行うほど、下流のデバッグ工程で負担が軽減していく環境の実現に、 一歩近づきました。

■ ツール開発者は考える ～ 自分の言葉で伝えたい

以上が、私たちの先駆者が築いてきた環境です。さて、次の時代を担う私たち後継者は、その意義を理解・咀嚼し発展させなければなりません。学生時代、高品質のソフトウェアを開発するには設計段階の検証が重要だとソフトウェア工学で学びました。しかし理論的には理解できても、百戦錬磨の現場の設計者に自信を持って説明するには、抵抗がありました。幸い、「もしお前達の言う環境で開発して納期が間に合わなかったら、誰が責任を取るんだ！」とは言われませんでしたが、お客様に紹介するのには真の声でなければならないと考えました。少なくとも自分の体験したセットとデバイスならば数字を示すことは可能になります。本誌に寄稿されている松下電器産業株式会社オーディオ事業部・根岸様からの支援もいただきましたし、自分達で開発しているツールの使用実態に触れる良い機会でもありました。お客様に紹介するには、ZIPC とその統合環境を使うことがいちばんです。こうして、私たちは二つの取り組みを行いました。

■ ツール開発者の試み(1) [ビデオ CD チェンジャ]

【開発概要】

私は以前、ZIPC を使用したデモセットの開発を行った事がありました。過去の展示会等で、CATS 殿のブースで使用された CD チェンジャ(松下電器産業株式会社オーディオ事業部殿製品)SL-MC410 です。このセットは、元の製品をデモ向けに再開発したものですので、ZIPC 化したのは一部だけでした。

本稿で発表する製品開発は、デモセットではありません。世の中に出回る製品です。バグなんてあってはならない品質の高い物が要求されます。初めてなので多少の不安もありましたが、ZIPC を使って「高品質な商品開発」にチャレンジする事にしました。

◇ 開発ターゲット

- ターゲット … 松下電器産業株式会社製ビデオ CD チェンジャ SL-VM535
- マイコン … 78K/0
- 担当モジュール … メカニズム制御部

◇ 開発器材

ハードウェア

- IE … IE-78000-R-A
- EM ボード … IE-780208-R-EM
- その他 … Realtime RAM Tracer

ソフトウェア

- コンパイラ … CC78K0
- アセンブラ … RA78K0
- デバッガ … ID78K0
- タスクデバッガ … MD78K0
- OS … MX78K0
- CASE … ZIPC V4.0w
- アイデアプロセッサ … idea-L

【開発工程】

1. 設計

すべてに ZIPC を適用するのはリスクが大きいとの判断で、モジュールの一つが選択され、その部分を私が担当しました。動作を説明した文章とフローチャート、及びタイミングチャートが記された要求仕様書が手渡され、設計が始まりました。

最初は1枚の状態遷移表で作成するつもりでしたが、表が複雑になりそうなので階層化を試みました。しかし表を作り込んでいくと、階層化よりも別タスクとする方が良いと判断し、最終的には図 1 の形になりました。

今回の開発では、MX78K0 (*1) という OS を使用しました。この OS のタスクは「WAIT 状態を持たない」という特徴を持っているため、時間制御が必要な処理を実現するためには複数

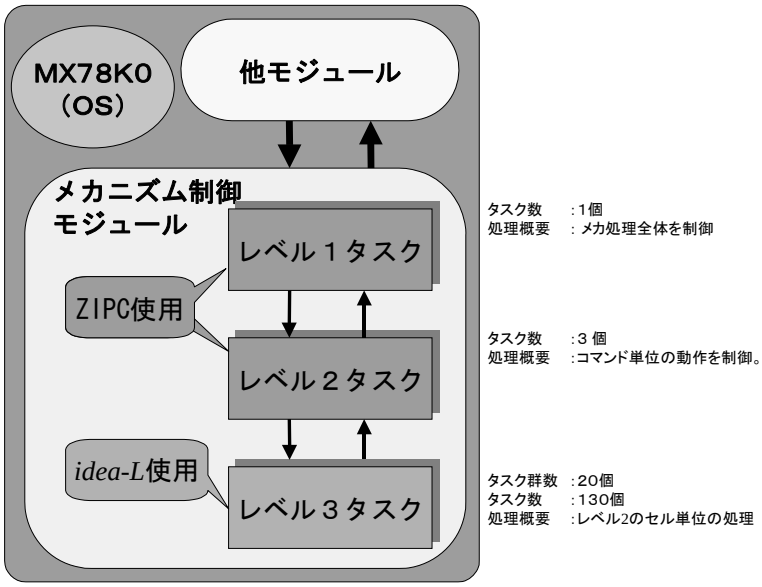

図 1 設計内容

のタスクを組み合わせる必要があります。そのため、レベル 2 のタスクで時間制御が必要な処理は、レベル 3 のタスクとして設計を行いました。

レベル 1／レベル 2 のタスクの設計には ZIPC を用いましたが、レベル 3 タスクの設計には idea-L (*2) を用いました。レベル 1・レベル 2 のタスクの設計が完了した時点で、状態遷移表レビューを行いました。

動作の流れを表の上で示しながら説明する事が出来たので、レビューの進行は容易でした。他モジュール設計者とインタフェース仕様を固める時に、コマンド内容と送受信のタイミングに関して、意志の疎通が図りやすかったと思います。特にエラー処理の確認においては、仕様書の内容

(*1) MX78K0 → NEC 製 78K0 用 OS（μITORN サブセット仕様）
- ・タスクが WAIT 状態を持たない。
- ・プリエンプション（タスクの強制中断）機能を持たない。
- ・高速かつコンパクト。

(*2) idea-L → NEC 製アイデアプロセッサ機能付きエディタ
- ・記述内容を階層化して整理する事により、効率的なプログラミングが可能。
- ・複数のテキストや図、表を 1 つのファイル（.idl）にまとめる事が可能。

と自分の解釈のズレを無くす事ができました。エラーが発生した場合の動作内容と他モジュールに通知するエラーの種類を状態遷移表で説明する事により、理解していなかったこと、不十分であることが発見できたのです。

また設計内容も説明する事により、不具合に気が付きました。

「オープン動作中ににクローズ動作の指示を送ったらどうなりますか？」

「オープン動作が完了してから、クローズ動作を開始します。」

「それは駄目です！オープン動作を中断してすぐにクローズ動作を行って下さい！」

この問題が実機デバッグまでわからなかったら、メカを壊していたかもしれません。レビュー段階で問題を発見できたのは非常に良かったです。また、メカの特性変更やサーボ動作の変更で、担当モジュールの変更や修正をよく行いましたが、その内容も容易に状態遷移表に反映出来ました。

2. シミュレーション

ドキュメントを一通り書き終えたので、動作内容を確認するためにシミュレーションを行いました。表レベルでの正常動作の確認を行いましたが、それ以外の部分は省略してしまいました。

3. コーディング

ZIPC を使用した部分は、ボタン一つでコードのほとんどを生成できるので非常に楽でした。強いて言えばジェネレータの設定に手間取った程度ですが、一度設定を行えば後は変更する必要がないので、大した問題ではありません。レベル 3 タスク（ZIPC 未使用部）の作り込みに時間がかかったように思えましたが、レベル 1／レベル 2 は設計段階で時間をかけているので、実は同じ程度だったようです。

4. デバッグ

実際にターゲット上でプログラムを動かし始めてみると、多少の不具合が発生しました。不具合の解析時には表のセル単位で動作を確認したので、ソース・レベルで追うよりはるかに原因の特定が行い易いと感じました。不具合を発見した状態までもう一度セル単位で再実行させると、ほとんどの原因が直前に実行したセル、またはその 2、3 ステップ前のセル内容に関係していました。

また今回のデバッグには、統合デ

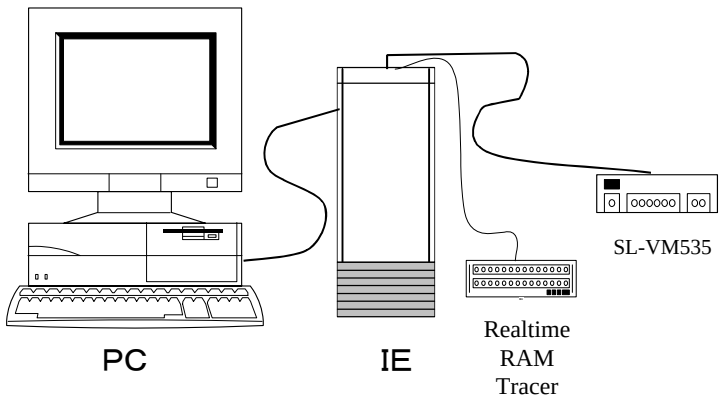

図 2 デバッグ環境

バッガ ID78K0 (*3) とタスクデバッガ MD78K0 (*4) も使用しました。レベル 1、2 のタスクのデバッグには ZIPC が、レベル 3 のタスクのデバッグには MD78K0 が非常に役に立ちました。

デバッグ時に気が付いた事なのですが、不具合件数のほとんどが ZIPC 未使用部分でした。これは設計に時間をかけた成果であると思います。ZIPC 使用部の不具合の内容に関しても、表の再設計が必要になるような致命的なものはありませんでした。

不具合の中にはリトライ回数の間違いなど、動かせばすぐにわかる物もあり、改めてシミュレーションをも行っておくべきだったと思う事がありました。

5. 仕様変更

担当部分だけ先行してデバッグを完了したのですが、まだ作業は終わりませんでした。他のモジュールのデバッグ中に新たな問題が発見され、仕様変更を行う事になりました。

変更内容は、「状態の追加」と「事象の追加」です。表単位で処理が独立していた事と処理手順が把握しやすかったため、影響範囲の確認が簡

(*3) ID78K0 → NEC 製 78K0 用統合デバッガ
- 78K0 をターゲットにした統合デバッガ

(*4) MD78K0 → NEC 製 MX78K0 用タスクデバッガ
- ID78K0 と組み合わせて使用するタスクデバッガ
- タスクのキューイング状況の確認と、システムコールの発行が可能

単に行えました。そのため数回にわたった変更依頼にも、素早く対処する事ができました。

【開発を終えて】

ZIPC を適用した効果

・レビューなどで動作内容の説明が行いやすい。
・設計時に手間がかかるが、その分コーディングとデバッグ期間が短縮されて、結果的には開発期間全体が短縮される！
・仕様変更時の影響確認が容易であるため、変更による副作用がない！！

ZIPC に望む事

・ファイルの種類が多すぎるので減らして欲しい。STM、RAM、DEF、G??…などの作成したファイルの管理が大変でした。最近は idea-L を用いて、1つの .idl ファイルにまとめています。

【所感】

約半年に渡った開発作業は、特に大きな問題の発生もなく、無事に終了しました。 今回の共同開発者の方々には非常に勉強させて頂き、実際のセット開発現場での問題を体験できました。この経験を生かして、今後の ZIPC と NEC ツールの連携に役立てて行きたいと思います。

【データ】

◇ 開発期間（担当部分のみ）
- 設計 … 3ヶ月
- コーディング … 1ヶ月
- デバッグ … 1ヶ月

◇ 設計書（ZIPC 使用部）
- STM 4 枚（7x8、8x23、4x9、3x33）

◇ ROM サイズ

全体	60KB
メカ部（全体）	8.5KB
メカ部（ZIPC 使用部）	4.5KB
メカ部（ZIPC 未使用部）	4.0KB

■ ツール開発者の試み(2) ［ＩＣカード］

【IC カードとの出会い】

最近、ガソリンスタンドで IC カードを手渡されました。CM やニュースでも話題になっています。IC カードが徐々に普及してきていることを実感します。

ある日、社内でカード用 OS や開発環境を整備する目的でプロジェクトが発足しました。カード規格である JICSAP（Japan Ic Card System APplication council）は、動作概要が

状態遷移表で記されていました。プロジェクト・リーダーから「この状態遷移表を ZIPC に適用したら、開発効率はあがるか」と質問されました。使われているマイコンは78K0Sシリーズ、8ビットの中でも特に小さい、機能限定版です。ほとんどのアプリケーションがアセンブラで組まれていますので、コードサイズというボトルネックをクリアするのは難しそうです。しかし最近は、78K0S ユーザからも ZIPC 対応の要望をいただいています。品質重視の思想はローエンドでも同じです。ベンチマークの必要性がありました。

【開発への準備】

今回、以下の開発環境を使用することになりました。

ハードウェア

・インサーキットエミュレータ IE789814-NS

・シリアル接続の IC カードリーダ

ソフトウェア

・コンパイラ … CC78K0S

・アセンブラ … RA78K0S

・デバッガ … ID78K0S-NS

・CASE … ZIPC V5.0

構成は、以下（図 3）の通りです。

【状態遷移表を作ってみる】

まずは IC カードの通信処理の把握から始まり、状態遷移表を記述していきます。JICSAP の状態遷移表は、無理矢理すべての処理を一枚の紙に押し込んでいるようで、理解にかなり苦労しました。記述が表形式になっ

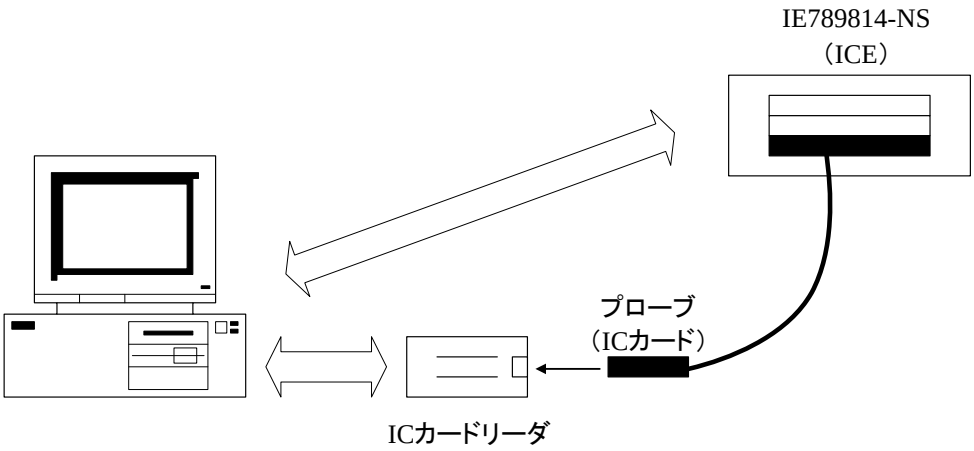

図 3 開発環境

ているから簡単に移植できるというわけではないようです。不明個所はカード OS の担当者に聞きながらの作業です。一通り完成したものを印刷すると、40%縮小してA4用紙6枚になりました。それらの紙を切り貼りして見渡してみました。印象は「とても大きい。」の一言でした。(セルの数は、252 個です)

この状態遷移表を用いて、レビューを行いました。動作の説明にも苦労しました。この時は、プロジェクト・リーダーもZIPCの効果を疑いはじめていました。ピンチです。通信処理には向いていると聞いていたのに、使い物にならないというレッテルが貼られそうでした。ZIPCによる開発経験のあるメンバとレビューを続けました。最初は、JICSAP 記述形式に拘り過ぎていたようですが、徐々に改善されました。大きさも A4 の紙 1 枚サイズに収まりました。(セルの数は、84 個まで縮小されました。)

【シミュレーションそしてジェネレーション】

ここで OK が出て、状態遷移表シミュレーションを行うことにしました。準備は単純です。

1. シミュレーション、ジェネレーションを行うための設定ファイルを作成する。
2. シミュレータで実行可能にするため、内部的なコンパイル、修正を繰り返す。

初めてZIPCに触れる人間にとっては覚えることが多いような気がしましたが、一度覚えてしまえば、変更や修正はさほど手間はかかりません。

実際にシミュレーションを行った印象ですが、プログラムの動作を把握しやすいと思いました。ソースをステップで追うより、はるかに全体像が見えます。しかしディスプレイ上では十分なスペースが取れないために、せっかくの効果は半減しています。事象による動作と遷移後の状態の表示、変数の表示に必要な領域が足らないのです。このあたりは、何か工夫が欲しいと感じました。

シミュレーションで動作確認と修正が完了したので、ソースの自動生成を行いました。必要なファイルの作成は、ほとんどシミュレーション時に完了しています。簡単な設定と STM 名を入力して"生成開始"のボタンを押すと・・・10 秒ほどで完了しました。

【デバッグまでの道のりはまだ続く】

このセットは、既にアセンブラによ

る開発実績がありました。イベント解析部や処理はアセンブラ・ソースを流用するつもりでしたが、ここで大きな壁に突き当たりました。それは、通信のコアになるブロック送受信部のタイミング処理です。元のソースは、タイミングを命令で計測し最適化されていました。状態遷移表にタイミングを見積もる処理を入れるにはタイマーが必要ですが、デバイスの制限でタイマーを使用することができません。また状態遷移表ではブロックを C 言語の変数として扱っているため、オーバーヘッドも予想されます。処理の切り分けをどうするか悩みました。

結果としては、タイマーを使用せずにいけそうです。しかし送信部オーバヘッドは回避できず、コード効率に不安が残ります。それでも動作するプログラムを作ることが最優先なので、実機によるデバッグ作業を開始しましたが、デバッグはあまり時間がかかりませんでした。

最後にアセンブラとのコード効率を計ることになりました。計算結果は元のアセンブラ・ソースに比べて 2.2 倍と判明、これでは実用できるサイズには程遠いです。

【コードサイズと STM のダイエット】

ZIPC ジェネレーターは、ソースをどのように生成するか指定できる機能を持っています。早速コードサイズが縮小されるオプションを指定して再度生成しました。しかしサイズは小さくならず、思惑通りいきませんでした。このオプションは、無視や不可のセルが存在しないと効果がないそうです。

今回の目的の一つはコードサイズですので、急遽別な対策を検討することになりました。考えられたのは、〝直接生成されたコードを改造する〟ことと、〝状態遷移表を根本的に見直す〟ことです。コード改造では自動生成の意味がなくなるので、見直しすることになりました。

まず、通信プロトコルを状態遷移表で設計したことがある経験者に見てもらい、わずか数時間のミーティングにより変更が加わりました。その結果、なんと同じ処理でセルの数を 45 個まで縮小することができました。この時縮小できた喜びと同時に、経験も必要になることを改めて痛感しました。

状態遷移表はほとんど 1 からの作り直しに近い状態になり、再度ソースコードを出力しなけれなりません。大きな改造に思えたのですが、設定ファイルの変更がほとんど必要なかったのでなんと 1 日で自動生成まで移行

でき、自動化の効果を実感しました。

気になるコードサイズは、期待したほどではありませんでしたが、1.86 倍まで縮小できました。

さらに、自動生成コードのチューニングが可能か、ZIPC 開発者ともミーティングを行いました。コンパイラの特性を生かしたコードを生成することにより、更に効率を上げられる事が出来そうだということがわかりました。

【次へのステップ】

目標は、大きく分けて 2 つあります。

1 つめは、アセンブラからの移植で実用化できるレベルにまで、コード効率をあげることです。現在コードサイズの縮小化を図るよう、ZIPC ジェネレータにチューニングが施されています。ここは ZIPC に期待します。

2 つめは、シミュレーション環境を強化することです。現在、弊社のシステム・シミュレータには外部イベントを取り込むための I/O インターフェースが備わっています。また、この処理のビジュアル化のため、独自の外部部品パネルを持っています。これを ZIPC VIP と共通化することにより、状態遷移表のシミュレーションだけでなくコード・シミュレーションのデバッグ環境を構築する事が容易になります。自動テストも共有できるでしょう。

最後に全体を通しての感想ですが、初めて ZIPC を使った開発を行い、さまざまな経験をしました。そして新しい開発環境が身近になりつつあることを実感しました。今は良い事ばかりではありません。しかしこの経験を生かし、次へのステップをなんとかクリアしながら、今後の発展へつなげていきたい思います。

■ 発展

長々と開発経緯と感想ばかり書いてしまいましたが、今まで構築してきた事に間違いがなかったことがわかりました。また勉強もしました。すべてのシステムに状態遷移表が適用できるのではないということ、不向きなものもあることです。

現在完成している環境では、まだまだ不十分な面も多いです。今回の経験から、実現してみたい機能が膨らみます。今後を期待してください。

（ぬまた けんじ／たちはら ゆうじ）