Chicken can fly if they have rocket engine.

プログラミング

HAL/S

O'EMMY

Mizuki tohru

プログラミング HAL/S

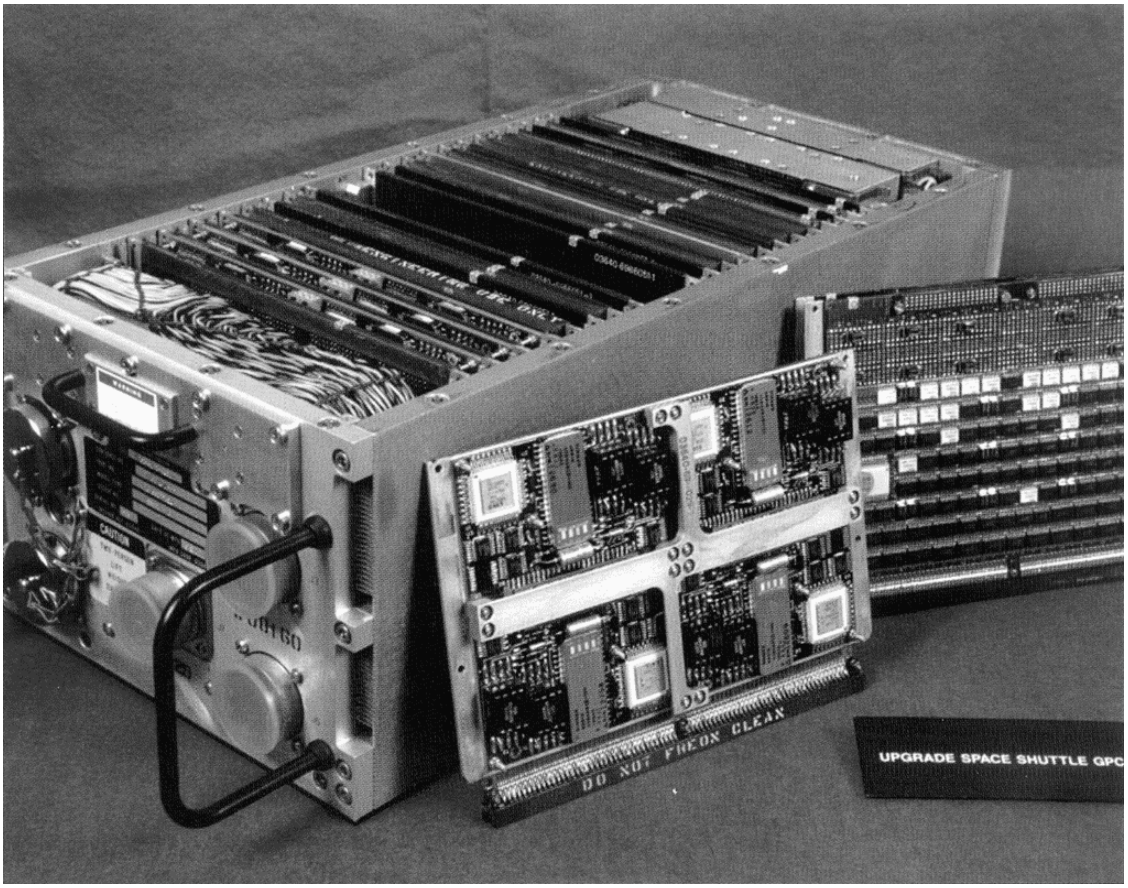
序：

　プログラミング言語に優劣は無い、と言う人がいる。
　それは今日の、素晴らしい言語が綺羅星のように並び立つ現状に慣れた、恵まれた立場からの発言である。残念ながらプログラミング言語に優劣は存在する。あまりの酷さに頭をかきむしり、製作者の正気を疑う、そういう言語は存在する。所詮、人が作るものなのだから。
　プログラミング言語は決して、チューリング完全を獲得次第速やかにユーザフレンドリーになったり、記述の簡潔さを自律的に獲得したりはしない。それらは創造者が与えるものなのだ。
　本書では、残念ながら優れた言語とならなかった例として、ひとつのプログラミング言語を紹介する。

目次

1：歴史的背景

図 1：AP-101



プログラミング言語 HAL/S は、米スペースシャトルの主制御計算機 AP-101 のプログラム記述に用いられている言語である。元々は人工衛星や宇宙機全般用途のプログラム記述言語となることを目指していたのだが、一部地上系で使われたのを除けば他には採用されることが無かった。実質シャトル専用の言語である。

シャトルの主コンピュータ AP-101 はアップデートを経て、運用停止までシャトルの全寿命の間使用され続けた。制御ソフトウェアもまた同じ期間使用され続け、そしてずっと立派に役割を果たしてきた。

シャトルはカプセルと違い、再突入において静的に不安定である。つまり制御しないと死ぬ。コンピュータが死ねば、ソフトウェアが阿呆をやらかせば、シャトルはほぼ即座に再突入に失敗する。そうなると待ち構えるのは OV-102 コロンビアの運命だ。

宇宙機としての振る舞いを制御しているのもこのソフトウェアである。シャトルが国際宇宙ステーションにドッキングできるのはこの制御のお陰である。主エンジン SSME の制御は別の機械が行っているが、それ以外、基本的なシャトルの制御はこのソフトウェアに委ねられている。

では、そのソフトウェアを記述している言語もまた素晴らしいものなのだろうか。

いや、残念ながら、極めて残念ながら、それは悲惨極まりない言語なのだ。

HAL/S 言語は Intermetrics 社によって開発、サポートされてきた。この企業は、アポロ誘導コンピュータ用プログラム記述言語 MAC(MIT Algebraic Compiler)の開発メンバーを中心として生まれた。

MAC 言語はチャールズ・スターク・ドレーパー研究所で開発された。この研究所は元々は MIT の付属研究機関、計装技術研究所が独立したものである。MIT リンカーンラボなどでは同名のプロジェクト MAC、悪名高き MULTICS OS の開発プロジェクトも同時に動いていたが、こちらの MAC とは関連は無いようである。

MAC は有人月計画で使われたアポロ誘導コンピュータのソフトウェア開発に用いられる予定の言語だった。だった、というのは開発に失敗したからだ。MAC は IBM650 で開発され、IBM704、Honeywell800 にも移植された。しかしアポロ誘導コンピュータ用のコンパイラの開発に失敗したらしい。

ではどう MAC 言語を使ったかと言うと、手で紙に、MAC 言語で書かれたコードをプログラミング部隊がハンドアセンブルしたのである。要するにプログラムの仕様定義をプログラミング言語もどきで形式化したに過ぎない。

1962 年から暫く、アポロ誘導コンピュータのソフトウェア開発初期には YUL と呼ばれる独自のアセンブラ言語が用いられていた。現在エミュレータ等で見ることがあるコードはこのコードである。しかしこれはあまりに珍妙な構文で、見るからに使いにくい代物だったので、最終的には DAP というアセンブラ言語に置き換えられたようである。YUL はパズルのようなアドレス指示の後ろに命令がつくという代物だったが、DAP は定数とラベルが使える、普通のアセンブラだった。

AGC のプログラムでは、インタプリタと呼ばれるプログラムが動いていた。但しこれは言語の体を成している訳では無い。これはプログラム中のサブルーチン指示をデコードするライブラリの一種で、これにより ROM サイズの縮小を狙っていた。

計装技術研究所では、当初 8 人がフルタイムのプログラミングに当たっていた。しかし、最終的にはそれは 300 人にまで膨れ上がることとなる。それはひとえに、AGC でのプログラム開発の困難さを物語っていた。

HAL/S という言語の名前の前半はこの MAC 言語の主開発者、Halcomp の名前に由来するとされているが、当時公開された映画"2001:a space odyssey"の中に出てくるコンピュータ、HAL9000 に由来する部分も当然ある筈である。

HAL/S の名前の後半、スラッシュの次の S は、シャトルを意味するとされている。元々の言語仕様から、シャトルへの最適化が行われた後でも汎用言語化するつもりはあったようだが、例えばガリレオ探査機にも HAL/S は採用される筈であったが、コンパイラの開発に失敗し、最後にはアセンブラ言語の HAL/S 風マクロというレベルにまで後退している。ガリレオ探査機で使用されたマイコン CDP1802 は、ただでさえコンパイラの製作の難しいマイコンであり、開発失敗は驚くに当たらないが、これは HAL/S 言語の実績蓄積に対する大きなダメージとなった。

HAL/S は当初予定では 11 機種ものコンピュータに移植される筈だった。HAL/S 言語は元々、コンパイル時に一度、中間言語 HALMAT 形式で出力してから、ターゲットハードウェア用のバイナリを生成する、多機種対応を目指した形式となっていた。例えば NASA の宇宙用標準計算機 NSSC 対応にも当初チャレンジしていたようだが、いつの間にか実装予定表から姿を消した。データジェネラルの 16bit 汎用機 Eclips で動くバージョンはなんとか AP-101 用のコードを吐くことが出来たが、追跡地上局に採用されていた汎用機 modcomp シリーズではそういう訳にはいかなかった。惑星探査機搭載用高信頼性計算機 ATAC-16M 用のものは、ハードウェア開発のキャンセルで CDP1802 用コンパ

イラの開発にスライドしたのだが、前述のとおり開発に失敗している。こうして宇宙機用のコンピュータソフトウェア記述言語のスタンダードを目指す夢は破れたのである。

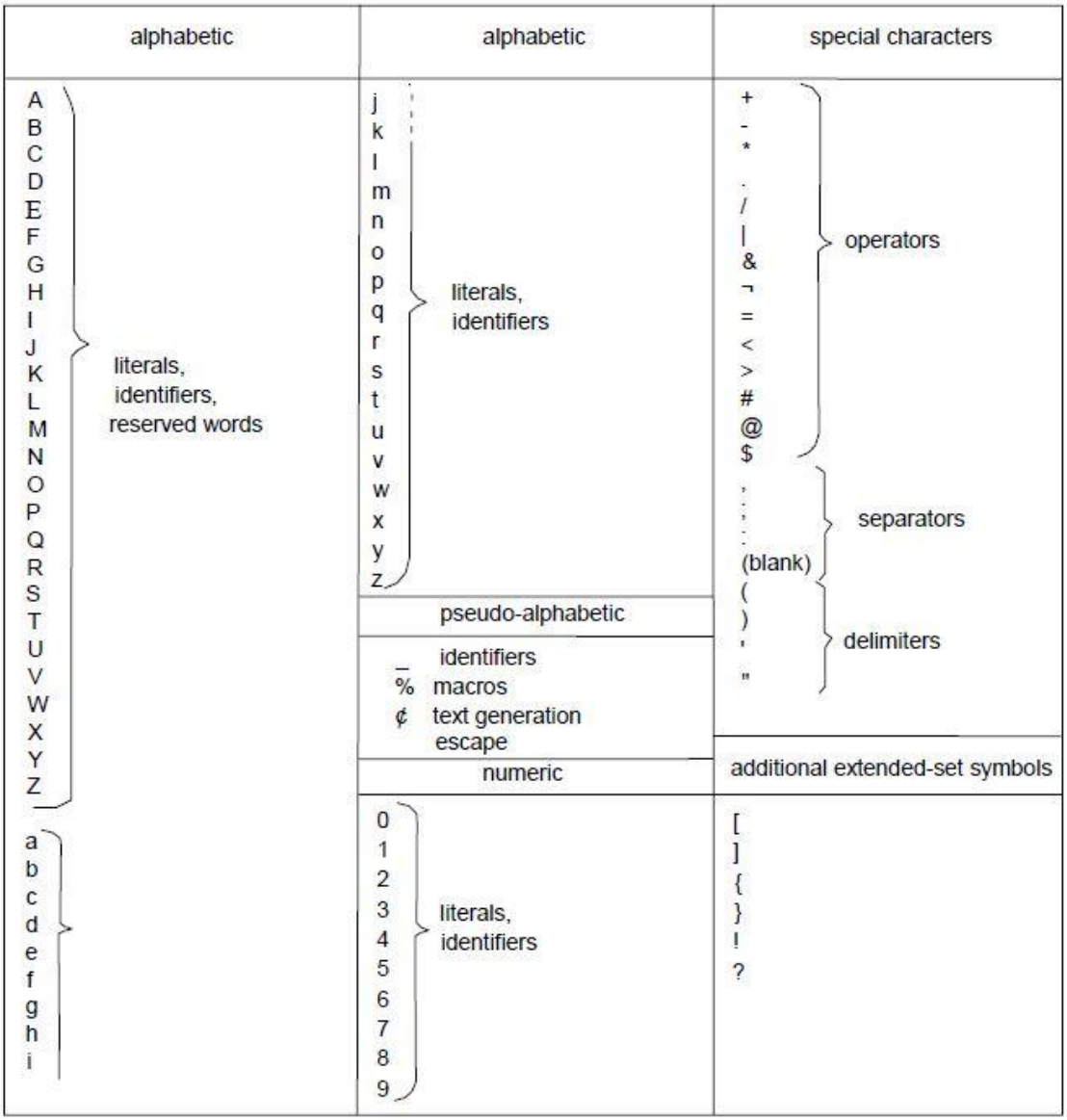
もしこの言語が優れていたなら、シャトルへの採用という素晴らしい実績に後押しされて様々なプロジェクトに採用されていただろう。もしかすると、組み込みシステムの標準記述言語にすらなったかもしれない。

しかし、そうならなかった理由は単純である。この言語は開発中に時代遅れとなり、継ぎ接ぎだらけとなったが、採用しないわけにはいかなくなったという、巨大プロジェクトにありがちな駄作なのだ。

正直なところを言おう。シャトルがこの言語を使いながらコンピュータシステムで致命的な不具合を出さなかったというのは、ひとつの奇跡だ。

2：変数と文字

図 2：HAL/S 文字セット



HAL/S は、素晴らしいとはちょっと言い難い仕様を色々と持っている。最悪とまでは言わないが、基本的に駄目な仕様で溢れている。

例えば HAL/S では、数値及び変数間の一文字空白は、乗算を表す。

```
DECLARE RV INTEGER;
RV = 123 456;
C "RV = 56088"
```

2 行目の、123 と 456 の間にある空白がそれである。ちなみに 1 行目は変数宣言、3 行目はコメントだ。

確かに、数学での算術記述において、乗算演算子は通常省略されるが、それをこのような形で真似できるようにした HAL/S でも、他のプログラミング言語と同じく、字句解釈のために乗算演算子に何らかの文字を割り当てる必要があった。結果、恐ろしい事に、空白が乗算演算子と化した訳だ。他の言語では乗算記号に使われるアスタリスク*は HAL/S ではベクタ型の乗算にしか使うことができない。アスタリスク二連続**は階乗なのに、である。

FORTRAN の後継を目指す FORTRESS 言語でも、空白を乗算演算子と解釈するそうだが、近代の言語だからそれなりに構文の文脈を読んで解釈する。しかし HAL/S は 1970 年代のコンパイラである。当然そのような柔軟さは期待できない。

HAL/S の文字コードは ASCII の基本文字セットをベースとした EBCDIC の一種であると HAL/S のマニュアルは主張するが、これは大嘘である。否定演算子¬が有ることからわかるとおり、HAL/S のソースコードは EBCDIC を使っている。当然だが ASCII にはそんなものは無い。恐らく HAL/S は独自の EBCDIC コードページを定義している。

EBCDIC にはエクスクラメーションマーク!も同時に定義されているから、他の言語っぽく否定演算子を!に置換するわけにはいかない。また NASA ゴダードの NSSDC の EBCDIC コードページで解釈すると、たぶん¬はサーカムフレックス^に置き換わってしまう。ちょっとプリンタを変えただけで大惨事だ。

HAL/S の使う字形は EBCDIC のものだけ(おそらくプリンタの制約だろう)だから、コード変換して ASCII プリンタに出力すると、否定演算子は消えうせてしまう。現在 HAL/S シミュレータを書こうとすると、HAL/S 文字コードと UNICODE を相互変換するターミナルが必要になるだろう。

図 3：シャトル文字コード

First HEX Digit

Second HEX Digit	0	1	2	3	4	5	6	7
0	NULL	α	SPACE	⊙	ϒ	P	ψ	p
1	]	β	!	1	A	Q	a	q
2	[	ρ	~	2	B	R	b	r
3	SELF TEST	ω	#	3	C	S	c	s
4	•	ε	√	4	D	T	d	t
5	••	Ω	%	5	E	U	e	u
6	∇	_	&	6	F	V	f	v
7	•	—	‘	7	G	W	g	w
8	BACKSPACE	|	(	8	H	X	h	x
9	÷	◊	)	9	I	Y	i	y
A	⛉	□	*	:	J	Z	j	z
B	▷	•	+	;	K	Σ	k	σ
C	◁	↑	,	<	L	θ	l	|
D	CARRIAGE RETURN	↓	-	=	M	‾‾‾‾‾‾‾‾ OVERSCORE	m	UNDERSCORE
E	⛁	→	.	>	N	π	n	λ
F	⛂	←	/	?	O	ϕ	o	Δ

Table 8-2 DEU Character Set

問題はこれからだ。シャトルの AP-101 特有の機械依存コードは実は ASCII ベースである。0x20 に空白文字が、0x30 に 0 が、0x41 に大文字の A がある訳だが、更にシャトル型の絵文字とかπ、λ、Σやら SELF TEST の文字列(!)やらが空いたコード空間に割り当てられているのだ。HAL/S は自己の使命のために、それら独自文字を使いこなさなければならない。独自文字を扱うために、HAL/S は string 型の表現のうち、非 ASCII 文字にエスケープシーケンスを用いたコードを割り当てている。つまり AP-101 にしかない字形と、HAL/S ソースコード上にしかない字形がそれぞれ存在していて、共通領域は狭く、どちらにせよ独自字形は使わざるを得ない。AP-101 でソースが読めないのは別に良い。別に軌道上でソースを見る必要がある訳では無いのだから。しかし地上では、ソースコードを一瞥しただけでは AP-101 でどういう出力がされるのか判らないというのは深刻である。HAL/S の AP-101 出力用の 7 ビット文字セットのうち、およそ半分がエスケープシーケンス使用コードである。

たぶん、シャトル用の字形を打ち出すプリンタを作って、開発環境でも AP-101 の文字コードを使っていれば、開発に関わった全員が幸せになれたのではと思う。しかし、事実は違う。

HAL/S 言語はシャトルのコンピュータ開発以前から開発されてきた 70 年代の代物なのだが、これらドキュメントのうち古いものは 1978 年製作であると対して、最新のものは、なんと 2005 年製作である。シャトルのプログラムをメンテする羽目になることを恐れてドキュメント化したのであろうか。内容、特にソースコードの記述スタイルは新旧であからさまに違う。しかし恐らく、実際のコードは古いドキュメントベースである筈だ。そもそも現在のコードの大半が書かれたのは 1978 年以前なのだから。

古い言語なので、全ての予約語はアルファベット大文字である。ドキュメントのサンプルコードではコードの全てが大文字になっている。変数、ラベル、そしてコメントまで全てだ。HAL/S は建前では小文字が使えることになっている。しかし実装では恐らく全てが大文字だろう。行終端には C と同じようにセミコロンを置く。

ブロック内で使用する変数は全てブロック内の先頭位置で定義しなければならない。変数の型や構文には PL/I 言語の影響が垣間見られるが、1978 年のドキュメントでは主に Fortran からの移行に主眼が置かれ、全て大文字、時として字下げ無しのプログラミングスタイルで記述している。

HAL/S は数値のテキスト表記として 10 進の整数及び小数、2 進、8 進、16 進表記をサポートしている。少数は固定小数点と IBM Hex 浮動小数点の双方の表記をサポートする。ただ、接頭辞や接尾辞を自動的に付けてくれるような親切な言語では無いことは承知すべきだ。

HAL/S でのコメント文は、PL/I 言語形式、つまり COBOL 形式と C 言語形式の二種類が使用できる。

先のサンプルコードの 3 行目は COBOL 形式のコメント文である。行頭でひと文字、その行の属性を宣言するという COBOL ライクな記法は、COBOL と ALGOL のいいとこ取りという、当時の IBM の野心的だが間違った言語設計によってもたらされたものだ。HAL/S には明確に PL/I 言語の影響が見られる。

PL/I はもう 1 つのコメント形式を持っている。C 言語でお馴染みのものだ。

```
/*THIS IS A COMMENT*/
```

そもそも C 言語のコメント形式は PL/I のコメント形式から来ている。HAL/S はどこまでも PL/I の追従者なのだ。HAL/S の仕様は PL/I の初期の仕様を反映した、ある種の化石であると考えられる。

HAL/S のコーディングは最終的には IBM パンチカードに落とし込まれた。IBM パンチカードでは 1 枚に付き一行 80 文字のテキスト記述を許す。HAL/S には実質一行 80 文字のコード長制限が存在し、字下げは全く推奨されなかっただろう。変数等のキーワード名も長くするのは推奨されなかったものと推測できる。

3：演算

HAL/S の数式記述法は中置記法なのだが、少し独特である。

```
HOGE = a b+1;
```

これは (a×b)+1 の結果を変数 HOGE に代入しているのだが、目を引くのは乗算の演算子に空白が使われている点だろう。数学の記法では確かに乗算演算子は省略できる。が、普通のプログラミング言語ではこんなことはしない。空白を単語の区切りに使うからだ。

単語の区切りと乗算演算子を区別できないと、例えば以下のような危険がある。

```
DO FOR I=0 T0 100 BY 1;
```

上記例は T の後ろにあるのはオーではなく数字のゼロである。もし T0,T1,T2 などの変数を不用意に定義していた場合、I は 0×T0×100 つまり 0 となりループ終了値 100 は定義されない。

HAL/S のコンパイラは、恐らく再帰下降構文解析を採用している。あまり想像力に富んだ字句解釈は期待できない。HAL/S の変数は事前の宣言が必要で、ローカル変数が使えるため、変数名で危険な事態になることは少ないかもしれない。それでも、短い変数名は危険であることには変わりない。

演算子の優先順位は以下のようになっている。

最高
- 階乗演算子 **
- 乗算演算子（空白）
- ベクタ乗算演算子 *
- ベクタ和算演算子 .
- 除算演算子 /
- 加算演算子 + および 減算演算子 -
- 否定演算子 ¬

最低

階乗演算子は以下のように使う。

```
V**100
```

この記述は V の 100 乗を表す。階乗演算子のみが右結合で、他はすべて左結合で評価される。

上記、見てのとおり、乗算と除算で優先順位に差がある。また、*はベクタ型、マトリックス型専用演算子で、整数などのスカラ型には使えないことがわかる。

実はほかにも演算子は色々ある。例えば論理演算子だ。優先順位は以下のようになっている。

最高
　結合演算子 || CAT
　論理積演算子 & AND
　論理和演算子 | OR
最低

& と AND、| と OR、|| と CAT、どちらを書いても良い。|| は実は文字列結合に使う演算子で、使い方は想像通りだ。

```
DECLARE G CHARACTER(4) INITIAL('HOGE');
X = G||"hoge"
C "X = HOGEhoge"
```

こういう用途以外で、CAT 演算子は使われる事は無い。
論理演算子だと、あと当然論理否定¬が出てこなければいけないが、優先順位は定義されていない。あと、論理演算子の優先順位は、算術演算子の下になる。また、大小比較演算子>や代入演算子=はみな同じ優先度で、算術演算子と論理演算子の間の優先度に位置する、らしい。
HAL/S にはインクリメント、デクリメントや、剰余や排他的論理和などといった甘えた演算子は存在しない。しかし慈悲深い HAL/S は組み込み関数というかたちで剰余などの演算を提供している。
問題は優先順位だ。ドキュメントをまともに取ると、論理演算する前に代入が行われてしまう。明らかにおかしいから、そこは間違いだろう。論理否定¬の優先度は恐らく論理演算子の最上位だ。組み込み関数に優先度は当然無い。括弧()内は優先されるから、まともな結果が欲しければ括弧を多用したほうが良いだろう。
HAL/S の演算子優先順位は PL/I のそれにかなり近いが、空白が乗算だったり、除算演算子の優先順位が乗算のそれと違っていたり、微妙に食い違っているのが面白い。もしかするとこれも初期の PL/I の仕様の反映、化石なのかも知れない。

HAL/S はマルチライン記法を、数式表現でサポートする。
一般的な言語では、マルチライン記法というと、例えば Ruby のヒアドキュメントのような、文字列に使用は限られる。これが HAL/S では、数列にのみ使用される。一例を以下に示す。

```
E                  100
M   DELTAV = G + V
S              E
```

100 は V の上添字、つまり V の 100 乗である。E は G の下添字、つまり対数 G の logE を示す。
行頭のひと文字 M が主の行で、その上に行頭文字 E の宣言で上添字ラインを、M 行の下に行頭文字 S の宣言で下添字ラインを宣言できるのだ。
これは文字の位置が狂うと大変な事になるし、コンパイラは一体どう作ったものか、まあそもそもこんな言語仕様にしようという方が間違っているのだが、こういう仕様も、COBOL の血のなせる業と理解すると、なんとなく判る気もする。
マルチライン記法での優先度は不明である。

HAL/S はマトリックス演算をサポートしており、そのために MATRIX 型と VECTOR 型という専用型がサポートされている。データ型については後述する。

VECTOR 型とスカラー型値、VECTOR 型と VECTOR 型の演算も可能である。制約は他の言語と大体同じで、例えばVECTOR型とVECTOR型の足し算は双方の要素数が一致していなければならない。MATRIX型も同様に演算可能である。

4：式と構文、構造

HAL/S のプログラムは、次のような開始と終端の宣言を必要とする。

```
"プログラム名": PROGRAM;
    /*ここにプログラムが記述される*/
CLOSE "プログラム名";
```

開始宣言では、プログラム名の後ろにコロンが必要で、そこからブロック開始宣言 PROGRAM が付く。終了宣言は CLOSE で、この後ろにもプログラム名が必要だが、こちらにはコロンは必要無い。そもそも OPEN していないのに CLOSE するのに気持ち悪さを感じる人もいるだろう。なお、ブロックの終わりの CLOSE の後ろのブロック名、この場合はプログラム名を省略することはできない。何故省略できないのか、多分設計者の思想なのだろう。意図は測りかねるが。

グローバル変数などの宣言は、プログラム開始宣言以降、最初の実行可能命令までの間に記述しなければならない。

HAL/S の制御構造であるブロックは以下のような 4 種のスタイルを取る。

```
"ブロック名": PROGRAM;
    内容;
CLOSE "ブロック名";
```

もしくは

```
"ブロック名": COMPOOL;
    内容;
CLOSE "ブロック名";
```

もしくは

```
"ブロック名": FUNCTION;
    内容;
CLOSE "ブロック名";
```

もしくは

```
"ブロック名": PROCEDURE;
    内容;
CLOSE "ブロック名";
```

ブロック PROGRAM は引数を取らず戻り値も無い。これがプログラム全体の基底、C 言語で言うところの main()関数となる。このブロックは基底に 1 つだけ宣言され、ネストすることはできない。

ブロック COMPOOL は引数を取らず戻り値も無い。このブロックはネストすることはできない。

ブロック FUNCTION は戻り値がある。このブロックは CLOSE する前に RETURN 文で変数の値をひとつ返すことができる。

ブロック PROCEDURE は引数も戻り値も備える。引数はキーワード PROCEDURE の後ろに括弧でくくって記述できる。引数は複数指定可能である。

C 言語なら 1 種で済むところを使い分けるのは、ただ単に言語設計者の錯乱によるものと思われる。多分言語設計者は使い分けを強制したかったのだろうが、この仕様は無用の複雑さ、設計変更の困難さ、そして言語使用者へ余分な負担を強制するだけだっただろう。

ブロックのスタイルにはもう一つ、TASK というものがあるが、これはスタイル PROGRAM のリアルタイム版である。このブロックについては後述する。

以下は PROCEDURE ブロックの例である。

```
HOGE: PROCEDURE(X,Y);
    RETURN X * Y;
CLOSE HOGE;

A = HOGE(B,C);
```

PROCEDURE は CALL 文を使って呼び出すことも出来る。その時は、戻り値はキーワード ASSIGN を使って取得することになる。

```
CALL HOGE(B,C) ASSIGN(A);
```

このように、プログラムブロックの呼び方は引数や戻り値の有無、主呼び出しかそこに展開されるマクロ、テンプレートであるかによって使い分けなければならない。

HAL/S は一応、構造化言語である。FUNCTION や PROCEDURE はネストすることが可能で、ローカル変数を持つことができる。但し、再帰は許されていない。

COMPOOL は特別な使い方が想定されている。外部のプログラムに共通変数のアドレスを示すための宣言として、C のヘッダファイルのように使うのだ。従って COMPOOL ブロックの中にはデータしか置くことが出来ない。COMPOOL はネストできないが、構造体でもベクタ型でも置くことができる。COMPOOL は実行されない。関数も置ければ素晴らしかったのに。

```
DATA_POOL: COMPOOL;
    DECLARE J INTEGER INITIAL(100);
CLOSE DATA_POOL;
```

こういう宣言が、どこかにあれば、他のプログラムがこれを参照するために、

```
DATA_POOL: EXTERNAL COMPOOL;
    DECLARE J INTEGER INITIAL(100);
CLOSE DATA_POOL;
```

とキーワード EXTERNAL をつけて、PROGRAM ブロックの外で宣言すれば良い。COMPOOL の中で宣言したデータは、メモリ空間に永続的にその位置を確保される。そのプログラムのどこからでも、COMPOOL で宣言した変数を参照することができる。

FUNCTION や PROCEDURE も、キーワード EXTERNAL を使って外部呼出しができる。ただ変数は永続的確保されず、その場で初期化される。

面白いことに、HAL/S には無名関数みたいなものも使う事ができる。

```
X = X + FUNCTION SCALAR;
        RETURN X;
        CLOSE;
        **2;
C "X = X+X**2"
```

FUNCTION 文の前に関数名が無いことに注目していただきたい。FUNCTION の後ろの SCALAR は戻り値の型である。上記はドキュメント内のサンプルそのままで、サンプルは一体何がしたいのかよく判らない。字下げもサンプルのままである。普通の関数のようにちゃんと CLOSE までのブロックに色々書けるのかも不明だ。変数に代入して便利に使うとか、引数に直接記述するとか、そういう機能は一切サポートしない。あくまでマクロの一種と割り切るべきである。

制御構文には何やら似たようなものが色々有る。

IF..THEN..ELSE は、ブロック宣言が無ければ、条件判断後の直後の行しか実行しない。ブロック宣言が無ければ、これは一行で記述すべき構文なのだ。

```
IF X THEN
    Y
ELSE
    Z;

A /*この位置は既に条件判断構文の外*/
```

この仕様はコンパイラを単純化するためのものだ。もし条件分岐の結果、直後一行しか実行されないのなら、条件分岐しなかった場合の飛び先は自動的にその次の行ということになる。多分、HAL/S コンパイラは飛び先を保持するリストなど持っていなかったのだろう。

しかし if 文の次の行にブロック宣言 DO..END を挟むと、そこは通常の C の if 文のようにシーケンシャルに実行してくれる。

```
IF X THEN DO;
    A;
    B;
    C;
END;

ELSE D;
```

つまりDO..ENDブロックの実体はメモリの離れたところに配置され、IF文の後ろにはブロックへの無条件ジャンプ文が挿入される訳だ。DO..ENDブロックの終端には、IF文を抜けた位置への無条件ジャンプが挿入されることになる。
こういった特長により、上のサンプルを見て判るとおり、制御構造は字下げで明瞭になるとは言い難い。

条件分岐にはもう一つ､DO CASE文というものが存在する｡CASEという単語から連想するような、C言語ライクなcase文とは違い、ただ単にCASEの後ろに記述された変数なり式なり定数なりの真偽値によって条件分岐をおこなうものである。IF文と同様にELSEも使用可能である。つまりIF文と大して変わらない。
あと、勿論GOTO文も用意されている。GOTO文を使わなくても構造化して記述できるとドキュメントで力説してあるが、説得力は無い。

構文ブロックから抜ける宣言には、RETURNとEXITの二種類がある。EXITは通常ループ構文と組で使用される。　DO..ENDブロックからはEXITで脱出できる。またREPEATでブロック先頭へ遷移する。REPEATにはラベルを指定してブロック以外へも遷移することができる。HAL/Sではこのような多彩な構文でスパゲティが製造できる。ループ構文でRETURNを使った場合、その他の構文ブロックでEXITを使った場合どうなるか、ドキュメントに記述は無かった。
ループ用に、DO FOR という構文も用意されている。

```
DO FOR I=0 TO 100 BY 1;
    A;
END;
```

FOR構文だけでは使い物にならないので、DO..ENDブロックと組み合わせてDO FOR構文と称している。構文は見たとおりである。FORの後ろに変数とその初期値、TOの後ろにしきい値、BYの後ろに変数の変化量となる。このループから条件によって抜けたいとき、例えばこう書く。

```
DO FOR I=0 TO 100 BY 1;
    IF I>A THEN EXIT;
END;
```

前述のEXITを使用するのである。

WHILE構文も存在する。WHILEの後ろの式が評価され、真なら後ろが実行される。これも現実的な動作をさせたいなら、DO..ENDブロックを使わなければならない。

```
DO WHILE X<10;
    X = X + 1;
END;
```

UNTIL構文も存在する。UNTILの後ろが評価され、真になるまで後ろが実行される。WHILEと真偽反対なだけである。

```
DO UNTIL X>10;
    X = X + 1;
END;
```

　このように、煩雑な構文を使い分けなければならないのは、勿論実装を意識せよという意味なのだが、公開されているドキュメントでは、その辺りを窺い知ることはできない。

5：データ型

変数宣言は、変数名の前に予約語 DECLARE を付けておこなわれる。変数名の後ろに型指定、更に初期化したいなら初期値指定が付く。型指定、初期値指定は省くこともできる。型指定を省くと型は INTEGER に、初期値指定を省くと値は不定となる。

HAL/S は配列をサポートする。HAL/S の配列には、高速でサイズ実行時可変だが 2 次元配列しか作れず厳しいサイズの制約があるがマトリックス演算に直接使用できる配列型と、1 次元当たり 32768 個までの要素を持つ多次元配列を実現する配列コンテナ型、計 2 種が存在する。

まず配列型から説明する。

```
DECLARE I INTEGER INITIAL(100);
```

I という整数変数を、初期値 100 で宣言している。

```
DECLARE J INTEGER DOUBLE;
DECLARE H INTEGER DOUBLE INITIAL(4.5E-3);
```

倍精度整数まで宣言可能である。更に、わざわざ単精度であると宣言することもできる。

```
DECLARE K INTEGER SINGLE;
```

浮動小数点値型は SCALAR とよばれる型となる。これも倍精度宣言 DOUBLE、単精度宣言 SINGLE キーワードの付与ができる。以降紹介する型でもこれは同様である。

```
DECLARE L SCALAR DOUBLE;
```

HAL/S にはキャラクタ型が存在する。

```
DECLARE Q CHARACTER(80);
DECLARE G CHARACTER(200) INITIAL('HOGE');
```

キャラクタ型は配列要素に文字を 1 つづつ格納した配列である。配列サイズは 1 から 255 までの値でなければならないが、実際には配列サイズゼロの場合もあり得る。ゼロ終端などの必要は無く、つまり PASCAL 文字列と同じである。

ブーリアン型も存在する。

```
DECLARE R BOOLEAN;
DECLARE F BOOLEAN INITIAL(FALSE);
```

ブーリアン型は TRUE と FALSE 以外の値を取らないから、当然 DOUBLE 宣言も SINGLE 宣言も存在しない。

イベント型という変数も存在する。

```
DECLARE EV1 EVENT;
DECLARE EV2 EVENT LATCHED;
```

これは特殊なので、リアルタイム機能の記述時に詳説する。

```
DECLARE I INTEGER STATIC;
```

STATIC 指定されたデータは宣言されたブロックが終了してもメモリにそのデータを保持し、永続的になる。逆は AUTOMATIC 指定で、指定が省略されている場合はこれが相当する。
FUNCTION ブロックは PREENMT 指定で再入可能となり、外部呼出し可能となるが、その場合 STATIC 指定されたデータ宣言があってはならない。
単精度、倍精度相互の変換には、BIT 型宣言 BIT()を用いる。

```
DECLARE X INTEGER SINGLE INITIAL(100);
DECLARE Y INTEGER DOUBLE;
    Y = BIT(X);
C " Y= 00000000 00000000 00000000 01100100"
```

また、特定のビット位置を切り出すことも出来る。

```
    Y = BIT 20 TO 30 (X);
C " Y= 0000011001"
```

ビット位置指定は、左端を 1 として数える。変数の精度が変わると、当然左端の示す二進桁の意味も変わるので、当然ながら非常な注意が必要である。
BIT()はほかにもなんと、文字列表記を数に変換してくれたりする。文字列がどういう基数表現なのか、指定するために@が使われる。

```
DECLARE X INTEGER SINGLE;
X = CHARACTER@HEX('64');
C "X = 100"
```

逆にはキャラクタ型宣言 CHARACTER()を用いる。

```
DECLARE BUF CHARACTER(10);
DECLARE X INTEGER SINGLE INITIAL(100);
BUF = CHARACTER@HEX(X);
C "64"
```

CHARACTER()は BIT()と同じく、文字列の特定位置を切り出す機能も持っている。

コンパクトな整数配列型 MATRIX の宣言では、行列の要素数も同時に宣言する。行列は二次元固定である。要素数は 1 から 64 までの整数でなければならない。もし行列要素数の宣言が無かった場合、その行列のサイズは 3×3 が仮定される。

```
DECLARE M MATRIX(10,10) DOUBLE;
```

行列の要素数はプログラム実行中に可変できる。

```
DECLARE N MATRIX(I,J);
```

コンパクトな浮動小数点配列型､VECTOR 型の要素数は 1 から 64 までの整数でなければならない。もし要素数の宣言が省略された場合、要素数は 3 が仮定される。行列の要素数はプログラム実行中に可変できる。

```
DECLARE P VECTOR(K);
```

HAL/S には配列コンテナ宣言が用意されている。これはどのようなデータ型でも利用可能である。

```
DECLARE S ARRAY(1000) VECTOR(X);
DECLARE T ARRAY(10000) CHARACTER(200);
```

配列要素数は 1 から 32768 までの整数が宣言できる。このサイズは宣言したらそれで固定で、実行中にサイズは可変できない。要素数ゼロも宣言できない。要素数の制約は各次元ごとに独立で、他に制約されない。また次元数も制約されない。
変数等データ宣言は、まとめることで多少だが簡略化できる。

```
DECLARE S;
DECLARE I INTEGER DOUBLE;
DECLARE M3 MATRIX;
DECLARE M6 MATRIX(6,6); separate declarations
DECLARE B BOOLEAN;
DECLARE C ARRAY(5) CHARACTER(20);
DECLARE V ARRAY(3) VECTOR;
```

例えば上記の例は、以下のように簡略化可能である。

```
DECLARE S
I INTEGER DOUBLE,
M3 MATRIX, equivalent compound
M6 MATRIX(6,6), declaration
C ARRAY(5) CHARACTER(20),
V ARRAY(3) VECTOR;
```

セミコロンとコンマの位置に注意していただきたい。

つまりデータ宣言を実質一行にまとめると、キーワード DECLARE を最初以外省くことができるという訳だが、5 行目、行頭コメント宣言と紛らわしい事に注意が必要である。この機能は簡単に開発メンバーを地獄送りにできる便利機能と言えるだろう。

HAL/S は構造体をサポートしている。記述法は PL/I のものとほぼ同じである。

```
STRUCTURE HOGE;
    1.FUGA;
    1.MOGE;
        2.HOEHOE;
        2.NOENOE;
CLOSE HOGE;

HOGE.MOGE.HOEHOE = X;
```

要素行頭の番号が煩わしいが、そういうものと思って我慢していただきたい。実は要素はいくらでも階層を深くすることができる。番号は階層の深さを表しているのだ。まぁ要するに、番号でしか階層を表現できない仕様だという事である。

ある階層のデータ要素は、直前の一段浅い要素に属している。例で言うと、HOEHOE と NOENOE は、MOGE を節とした葉に当たる。つまり木構造データだ。

但し既に説明したが、HAL/S で再帰は使えない。木の全要素をなめるのに再帰は使えない事は強く留意する必要がある。もちろん動作中にこの木構造に要素を付け足したりする事ができる訳でもない。またこの構造体は入出力のフォーマットとも親和性が低い。

6：置き換え

HAL/S には他の言語のマクロに当たる構文が二種類用意されている。しかしこれら二種はまったく別種のものであり、更にはマクロでもない。そして後述のとおり使い分ける必要がある。
まず、REPLACE 文がある。

```
REPLACE name BY "内容"
```

この宣言の後では、REPLACE 文で定義したキーワード name を書くと、これがダブルクォートで囲まれた内容に置換される。REPLACE 文は複数の行にまたがることができない。つまりあまり複雑な内容を含むことはできない。しかし、REPLACE 文のダブルクォートで囲まれた内容には、他の REPLACE 文で定義されたキーワードを含むことができる。REPLACE 文は実行時評価である。
REPLACE 文は、サンプルコードによると VECTOR 型や MATRIX 型と共に、そのサイズを可変したりするのに使うらしい。

```
DECLARE V1 VECTOR(N);/*N は事前に宣言された何か*/
REPLACE N BY "4";
DECLARE V2 VECTOR(N);/*N は 4*/
```

何が嬉しいのかいまいちわからない。

組み込みの REPLACE 文と呼ぶべき、%マクロという機能もある。定義済みのキーワードの前に%を付けたもので、組み込みの関数や機能のように使えるというものである。キーワードはおよそ 100 個ほどが定義されている。一例を以下に示す。

```
  N1 = %NAME;
```

%という文字は､HAL/S においてはこの約 100 個の単語と組み合わせるためにだけ存在しており、ユーザが新しくマクロを定義するのに使えたりすることは決して無い。

もうひとつのユーザ定義マクロ類似機能はラベルである。ラベルは実際には無条件分岐である。

```
ONE: X = X +1;
```

ラベル ONE:は以後、使用するとラベル以後の内容に置換されたように動くが、実際には、ラベル定義地点へのジャンプである｡ラベルへのジャンプの後､内容を実行すると元位置に実行は戻る。

```
IF X = 0 THEN ONE: Y = 0;
```

ただ、ラベルはGO TO文のジャンプターゲットとしても動作する。

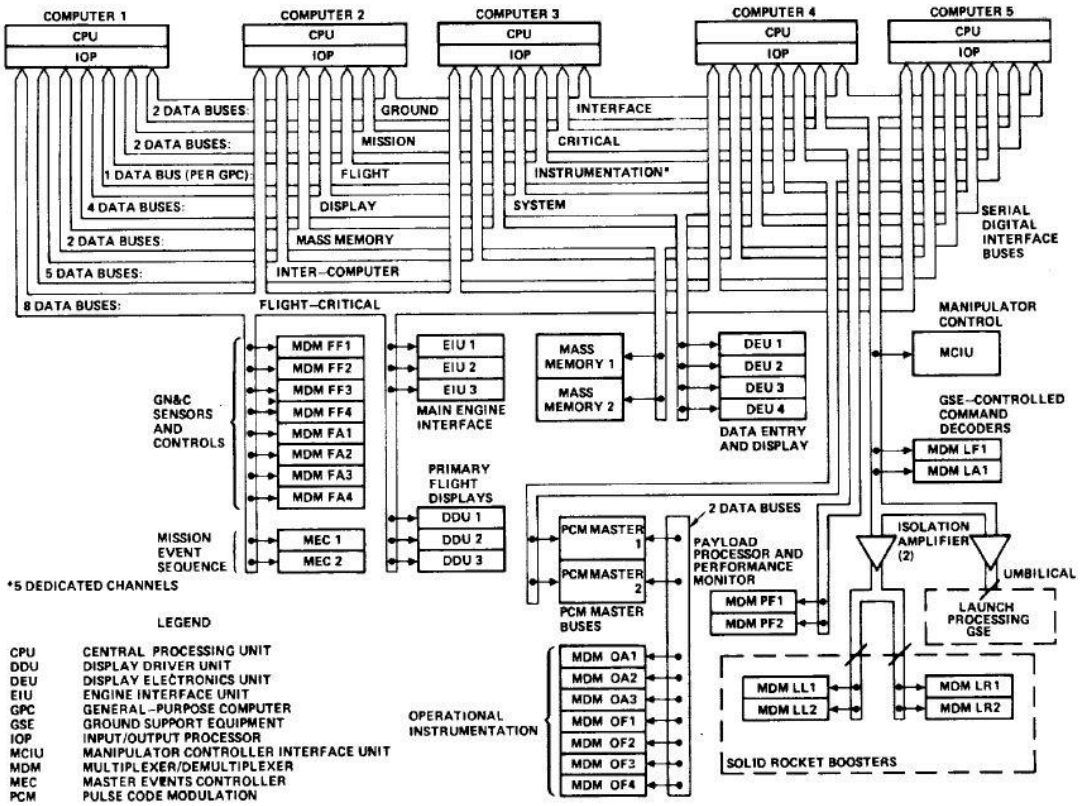
GO TO ラベル;

GO TO文のラベルの後ろは:ではなく;が使われる。GO TO文でのジャンプでは、実行制御はジャンプ先に行ったきりになる。

ラベルのこの2つの機能は混在させると大惨事につながることは明白なので、GO TO文の使用を禁じるか、ラベルのマクロ的使用を禁じるか、どちらかを選ぶ必要があるだろう。

7：入出力

図 4：シャトル情報処理系



HAL/S は外部とのインタフェイスのために I/O 文を持っている。Unix システムが基本的に I/O をファイルというかたちに抽象化しているように、HAL/S は I/O を全て"プリンタみたいなもの"に抽象化している。例えば外部デバイスの特定のアドレスにデータを書き込むとき、HAL/S では行とカラムのかたちで抽象化された指定アドレスに、改行文字やタブ文字を出力し、データを文字として出力することになる。読み出しは改行やタブを発行したあと、プリンタヘッドの位置にある文字を吸い出すようなイメージで行われる。だから厳密にはプリンタそのものに抽象化されている訳ではない。

純正のプリンタ I/O は"ページ化"されている I/O と呼ばれ、それ以外の大多数の I/O は"ページ化されていない"I/O と呼ばれている。シャトルが機内にどれだけの数の IBM ゴルフボールプリンタを搭載しているかは知らないが、まるでシャトルの制御がプリンタの出力と同じ調子で出来ると言うかのようである。

ストレージへの出力もプリンタへの出力と同じように行い、入力もラインとカラムを指定してやれば、まるでプリントアウトを読むかのようにデータを読むことが出来る。つまりラインとカラムは、磁気ディスク記憶装置のトラックとセクターの関係に近い。それはまた、ファイルシステム等は持っていないという事でもある。

ちょっと擁護するとすれば、これは当時の宇宙機のテレメトリフォーマットである固定長フォーマットに適合しやすい入出力フォーマットであるとも言える訳で、送信機にプリントアウトをする調子で地上にデータを送れる訳だ。

```
WRITE(4) V1;
READ(5) V2;
```

上記は I/O 入出力命令である WRITE 文、READ 文の例である。WRITE の後ろの括弧の中の数字はチャンネルと呼ばれ、各ハードウェア入出力先を示す。チャンネルは 0 から 9 までの数字を指定できる。これは数字だけである。前述のように HAL/S にはコンパイル時に決定、置換されるようなマクロは存在しないので、ここには数字か変数しか指定できない。

更にその後ろにあるのは変数だ。WRITE 文ではこの変数の内容が書き込まれるし、READ 文では読み出した内容がこの変数に代入される。

```
READ(4) V1,V2,V3;
```

変数は複数指定できる。型の違うものも混ぜる事ができる。この命令は今チャンネル内の仮想プリンタヘッドがあると思われる位置(特に宣言が無い場合はライン 1、カラム 1 の位置)から、3 カラム文データを取得し、変数 V1,V2,V3 にそれぞれ入れるという動作をする。READ 文では変数は並びの最初から代入されるし、WRITE 文では並びの最初から出力される。

気づいた方もいるかと思うが、ラインとカラムは 1 が原点である。0 ではない。チャンネルは 0 指定が出来るのでいかにもちぐはぐだが、まぁそういうものが HAL/S だと思って頂きたい。

プリンタのヘッドにあたる、開始ライン位置、開始カラム位置は、各文の変数記述位置に書く事ができる。これは READ 文、WRITE 文の従属構文となっており、単独では記述しない。構文として考えると変だが、ターミナル端末の画面制御文字と同じように考える事ができる。

```
WRITE(6)TAB(-50),V1,COLUMN(5),V2,V3,TAB(2),SKIP(3),V4,LINE(2);
```

TAB 文は後ろのカッコ内の数字だけ現位置より右へカラム個数ジャンプする。数字が負数なら左へのジャンプだ。COLUM 文は指定した絶対カラム位置への直接ジャンプである。SKIP 文は指定した数だけラインを数の増えるほう、下方に相対ジャンプする。指定数が負数なら上方ジャンプである。ただ上方ジャンプは(例えばプリンタのように)対応していないチャンネルがあるかも知れない。LINE 文は、指定したライン位置への直接ジャンプである。これも SKIP 文と同じ制約を持つ場合がある。

ページ化された I/O チャンネルでは、PAGE 文を使う事ができる。要するに改ページ命令だ。

```
WRITE(6) V1,PAGE(1),V2;
```

チャンネルは使い勝手を考えると英単語で記述できる変数指定にしたいところだが、もし変数が 9 以上になったら……とか、変なチャンネルを指してしまったら……とか無意味なリスク要因を作る事になる。またチャンネルが 10 個までしか無いことも問題となる。プリンタを 10 台まで繋げると考えると景気が良いが、シャトルの外部サブシステム、コンポーネントが 10 個しか無い訳が無く、つまりデータアクセス用のサブシステムがチャンネルの向こうに必要になる。コンポーネントとコンピュータの間に中継機器が必要なのだ。

の辺りはハードウェア設計の問題であり、HAL/S の問題ではない。コンピュータ AP-101 の設計は、組み込みシステムと言うよりはシャトルに載った汎用機と形容したほうが良い。

ただ、チャンネルの向こうには複数のサブシステムがある訳で、HAL/S は結局それらサブシステムを区別してアクセスする必要がある。こういう場合、サブシステムを仮想的にモジュール化してアクセスを容易にする工夫などが言語に備わっていると嬉しかった筈だが、HAL/S にはそういったものは一切存在していない。

その代わりに、入出力フォーマットを整形する、IN 文が存在している。これも READ 文、WRITE 文の従属構文だ。

```
WRITE(6) (V1, V2, V3) IN 'I4';
```

IN 文の後ろシングルクォーテーションの中が、フォーマット指定だ。I4 は整数で 4 文字であることを示している。10 進 4 桁と言い換えても良い。実に EBCDIC 的な指定の仕方だ。printf みたいなものだと思ったほうがいいだろう。変数を囲む括弧は、その直後の IN 文の適用される範囲を示している。つまり、括弧が無ければ、IN 文の指定は V3 にしか適用されなかっただろう。但し、フォーマット指定はそこから、次に別の指定があるまで永続する。つまり、以下のように書いても同じである。

```
WRITE(6) V1 IN 'I4', V2, V3;
```

フォーマット指定は以下のようになっている。

I 整数
F 固定小数点実数
E 浮動小数点実数
U 実数もしくは文字
A 文字
X 空白
P 整数と実数

```
READ(5) V1 IN 'F8.2', V2 IN 'F10.3',SKIP(1),COLUMN(1),V3 IN 'A6';
```

固定小数点実数指定では、後ろの数字は固定小数点表記でどのようになるか、小数点以上の文字数、少数位置、少数点以下の文字数をそれぞれ示す。これも 10 進値である。

フォーマット指定には、算術的な表現を使って記述を色々と簡略化できるようになっている。たとえばフォーマット指定のアルファベット 1 文字の前に書いた数字は、フォーマット指定を数字回繰り返すのと同じ記述になる。'5I2' は、'I2','I2','I2','I2','I2' と同じである。しかし、前述の通りフォーマット指定は次の指定があるまで同じ指定を引き継ぐので、このような記述にはぜんぜん意味は無い。ただ、括弧指定ができるので、以下のようになると意味を持ってくる。

```
IN '2(I2/I4)'
```

スラッシュ/は、フォーマット指定を並べるときの区切りである。上記例は、

```
IN 'I2','I4','I2','I4'
```

となる。複雑なフォーマットを定義したいときに使うことになるだろう。多分、構造体と 1 対 1 で対応するよう、構造体記述と首っ引きでフォーマット定義を行う作業が発生する筈だ。

フォーマット記述の中には文字列も混ぜる事ができる。ダブルクォーテーションで囲った文字列がそのまま出力される。

```
WRITE(6) V1 IN ' "ANSWER=", I4';
#ex ANSWER=0001
```

余計な機能に思えるが、大量データの連続読み出しのために READALL 文というものが存在している。

```
READALL(3) V1,V2,V3;
```

V1,V2,V3 はそれぞれ、長さの決まった文字列もしくは配列でなければならない。READALL 文はこれら変数を満杯になるまで連続して読み出す。この機能は READ 文に最初から存在しているか、フォーマット記述を使うべき部分だと思うのだが、開発側はどうもそうは思わなかったようだ。そして WRITEALL 文は存在しない。

ストレージデバイスへのバイナリデータの読み書き用に、FILE 文というものが存在している。嬉しいことに FILE 文は読み書き双方に働く

```
FILE(4, ADDR) = V1;
```

上記は、チャンネル 4 のアドレス ADDR に、変数 V1 の内容を書き込むものである。

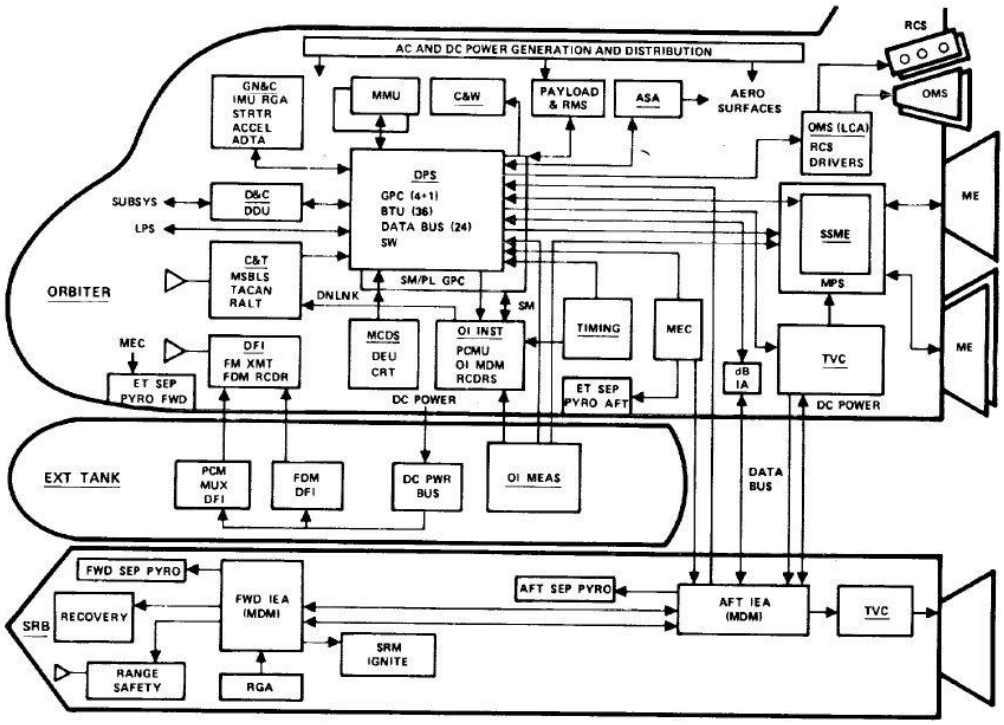
```
V2 = FILE(3, ADDR);
```

上記は、チャンネル 3 のアドレス ADDR から読み出した内容を、変数 V2 に書き込む動きをする。

アドレスはスカラー型であればどんな変数でも定数でも指定できる。これは実はライン、カラムという指定子とは関係が定義されていない。FILE 文はハードウェアの仕様に密接に関連している。アドレスはハードウェアの仕様依存である。FILE 文は恐らく後からストレージが追加された時に同時に付加されたものと思われる。

8：リアルタイム

図 5：シャトル制御系



HAL/S はリアルタイム言語である。

何故シャトルはこんな言語を使い続けたのか、ここに理由がある。リアルタイム言語とは、簡単に言えばリアルタイム OS の機能を持った言語である。実際には、リアルタイム言語の歴史はリアルタイム OS のそれに先行する。

世界最初の本格リアルタイム OS、DEC RSX-15 は 1970 年に生まれた。リアルタイム機能の基礎であるタスク間同期は、リアルタイム化 FORTRAN である OLERT のランデブ機能として最初に実装されたアイディアを使ったものである

リアルタイム機能とは最初それはハードウェア割り込みの事であった。非同期で低速な外部入力をコンピュータが受け取るために、軍事通信システムで生まれた割り込みは、すぐに産業用コンピュータでも一般的な仕様となった。マルチタスクは割り込みを処理するためのソフトウェア技術として生まれた。マルチタスクは一時期並列コンピューティングとすら呼ばれたことがある。

RSX-15 のマルチタスク機能は、最初"ソフトウェア"割り込みと呼ばれた。ハードウェア割り込みを処理するソフトウェア層の上に、抽象化された割り込みを定義可能にしたのがその実態である。しかしそれだけではマルチタスクは成立しない。非同期に動くタスクの間でデータをやり取りするには特別な仕組みが必要だった。RSX ではそれがランデブ機能だった。HAL/S もマルチタスク機能とタスク間通信のサポートを言語レベルで提供する。

HAL/S の各タスクは、ランタイム内のリアルタイム実行部(RTE)によってコントロールされる。実のところ PROGRAM ブロックもまたコントロール下にあり、スケジューリング定義の無い状態のままで RTE の実行待ち行列に突っ込まれ、実行されているのである。RTE は恐らく優先度付きラウンドロビンスケジューラである。

タスク定義はPROGRAMブロックと同じ構文で行われる。引数を取らず戻り値も無い。

```
"ブロック名": TASK;
    内容;
CLOSE "ブロック名";
```

タスクがどのように実行されるかはスケジューリング定義SCHEDULEで記述される。

```
SCHEDULE "ブロック名" PRIORITY("優先度");
```

優先度は0から255まで指定可能だ。数字の大きいほうが実行は優先的に行われる。

```
SCHEDULE "ブロック名" PRIORITY("優先度") DEPENDENT;
```

キーワードDEPENDENTは、このTASKプロセスを作ったプロセスに動作が依存することを示している。これは、タスク内からスケジューラ的な小型タスクを呼び出して使用するときなどに使う。

```
SCHEDULE "ブロック名" IN "時間間隔" PRIORITY("優先度");
```

キーワードINを用いて実行インターバルを設定できる。この時間単位は秒で、少数点数で指定ができる。

```
SCHEDULE "ブロック名" AT "指定時間" PRIORITY("優先度");
```

キーワードATを用いて、指定時間になったら実行するようにできる。指定時間はスカラー値の変数と小数点の直値を使う事になる。

SCHEDULE文はPROGRAMブロック内、もしくはタスク内にしか書けない。SCHEDULE文はタスクの呼び出しを兼ねている。

タスクはRETURN文を実行すると停止する。またブロックの終端であるCLOSEに到達しても停止する。また、タスクはTERMINATE文を使っても停止する事ができる。これら三種は同じ動作を示す。タスクは再びSCHEDULE文によらなければ動作しない。

タスクが内部から自分自身を停止する場合は、

```
TERMINATE;
```

他のタスクを停止する場合は、

```
TERMINATE タスク名;
```

というかたちの記述となる。停止タスクはカンマで区切って複数並べることもできる。

CANCEL 文は、タスクを停止し、RTE の実行待ち行列から弾き出す。これは主にエラー状態のときに用いる。

タスクは以下の 4 種類の状態を遷移する。

Active　：タスクは動作しています。
Wait　　：タスクは状態を保ったまま停止し、制御を他のタスクに渡します。
Ready　：実行待ちです。
Stall　　：タスクは不活性化しています。SCHEDULE によって Ready に遷移します。

CANCEL 文は、タスクを Stall 状態へと遷移させる。記法も動作も TERMINATE と同じである。タスク名は変数としてブーリアン型の値を持つ。タスクが Active だと true、Active でなければ false を返す。

タスク間の同期機能は、タスクによって分割された処理を再び統合するために必須の機能である。ここを上手く処理しなければタスク間で情報を正しく渡し損ねる場合がある。例えば情報を渡している最中に割り込み処理が入ったらどうなるか。情報の受け渡しは中断され、失敗するだろう。HAL/S も一応そのための仕組みを持っている。

WAIT 文は、タスクを一時停止して制御を他のタスクに渡す事ができる。

WAIT の後ろに数字もしくは変数で、停止時間間隔を指定できる。これは SCHEDULE の時間間隔指定と同じ、秒単位の指定である。キーワード UNTIL を付けると、指定の RTE システムタイマ絶対値まで停止し続けるという動作になる。キーワード FOR はイベント待ちである。

```
WAIT FOR DEPENDENT;
```

これは親プロセスに同期して停止するという意味である。

UPDATE 文は、SCHEDULE で指定された優先度を変更できる。タスク名を省略すると、自タスクの優先度を変更する。HAL/S のスケジューラは、低優先度タスクの自動優先度上昇のような機能を持たない。代わりに、手動で行うのである。

```
UPDATE PRIORITY タスク名 TO 16;
```

WAIT 文のキーワード FOR で出てきたイベントとは、DEPENDENT 以外にも様々に定義が可能である。これにはイベント型変数を用いる。

```
DECLARE EV1 EVENT;
```

イベントは要するにソフトウェア生成される割り込み、通常のリアルタイム OS で言うところのキューだと思えば良い。これが HAL/S のタスク間同期機構を担っている。

```
WAIT FOR EV1;
```

これは EV1 のイベント待ちである。

```
SIGNAL EV1;
```

これで一発、イベントが発生する。

イベントには、発生したら打ち消すまでセットされるものもある。
キーワードLATCHEDは、このイベント変数がオンオフできることを示している。

```
DECLARE EV2 EVENT LATCHED;
SET EV2;
RESET EV2;
```

SET文はLATCHEDでないイベント変数にも使える。その場合はRESETしてやる必要は無い。
LATCHEDなイベント変数は、最初からセットしてやることもできる。

```
DECLARE EV3 EVENT LATCHED INITIAL(TRUE);
```

イベント変数は配列にすることもできる。

```
DECLARE EVN ARRAY(3) EVENT;
SET EVN(1 TO 2);
```

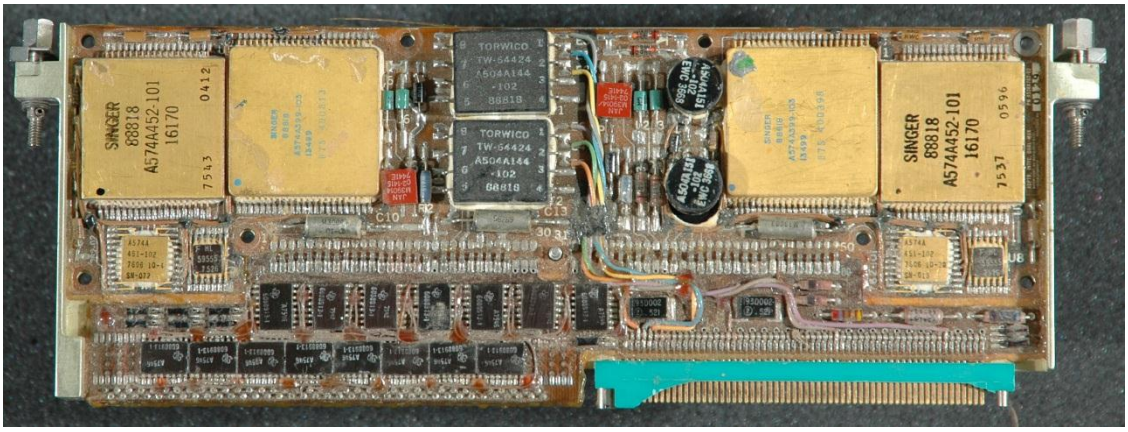
イベント変数は論理演算することもできる。

```
WAIT FOR(EV1&EV2);
```

HAL/Sにはタスク間同期機構がイベント変数しかなく、データの集合をアクセス衝突なく同期伝送できる仕組みを備えていない。HAL/Sのタスク間データのやりとりはCOMPOOLを使う訳だが、セマフォに相当する仕組みがHAL/Sには備わっていないため、データフローの設計は非常に気を使うものになっただろう。

9：ハードウェアと運用仕様

図 6：AP-101 CPU ボード



HAL/S のメインプラットフォームである AP-101 計算機は、1970 年代に開発された TTL トランジスタコンピュータである。およそ高さ 20 センチ、幅 26 センチ、奥行き 50 センチ、重さ 26 キログラム。メーカである IBM は自社の System/360 互換のアーキテクチャを AP-101 に与えた。

System/360 は 32 ビットアーキテクチャ機で、8 ビット単位でワード長を可変できる最初のコンピュータだった。うち System/360 モデル 30 をベースに IBM は 60 年代後半、耐放射線性コンポーネントを使った組込みフォールトトレラントコンピュータ、System/4 pi アーキテクチャの AP-1 計算機を開発した。AP-101 はこれを元にしている。

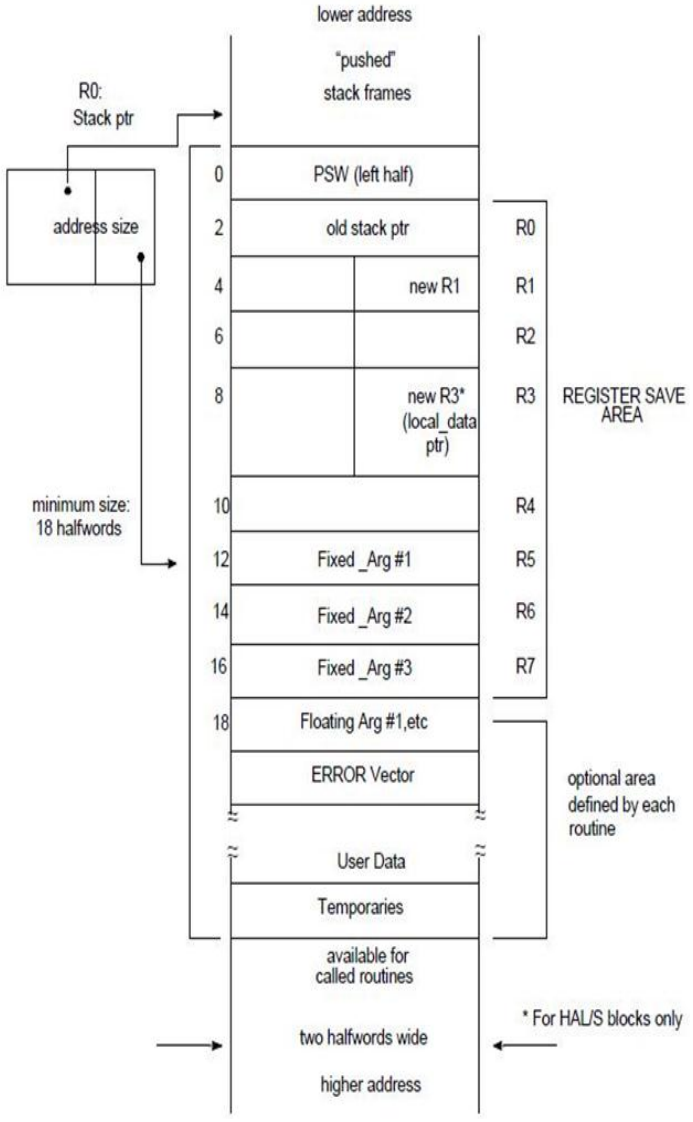


図 7：メモリアドレス 0 付近

演算能力は 0.4MIPS、メモリはコアメモリ、入出力チャンネルは 24 本用意されていた。OP コードも基本命令 154 種をサポートする。32 ビットの汎用レジスタ 16 個を持ち、立派な 32 ビットマシンなのだが、実際は 16 ビットマシンみたいなものだった。まずアドレス空間が 16 ビット分しかない。バス幅が 16 ビットなのだ。データアクセスの単位も 16 ビットが実質 1 ワードで、メモリ空間が 64 キロワードだから、メモリ容量は 128 キロバイトだ。少なくとも HAL/S にとって単精度は 16 ビットで、倍精度で 32 ビットだった。

スタックの最基部には 2 ワード 32 ビットの状態レジスタがある。その上にレジスタ退避エリア 16 ワードが用意され、この計 18 ワードがスタック基部に必ず確保される。HAL/S はここにレジスタのうち R0 から R7 までの 8 個しか退避しない。これは恐らく割り込み時のレジスタ退避エリアで、従って HAL/S は多重割り込みをサポートしないと思われる。スタ

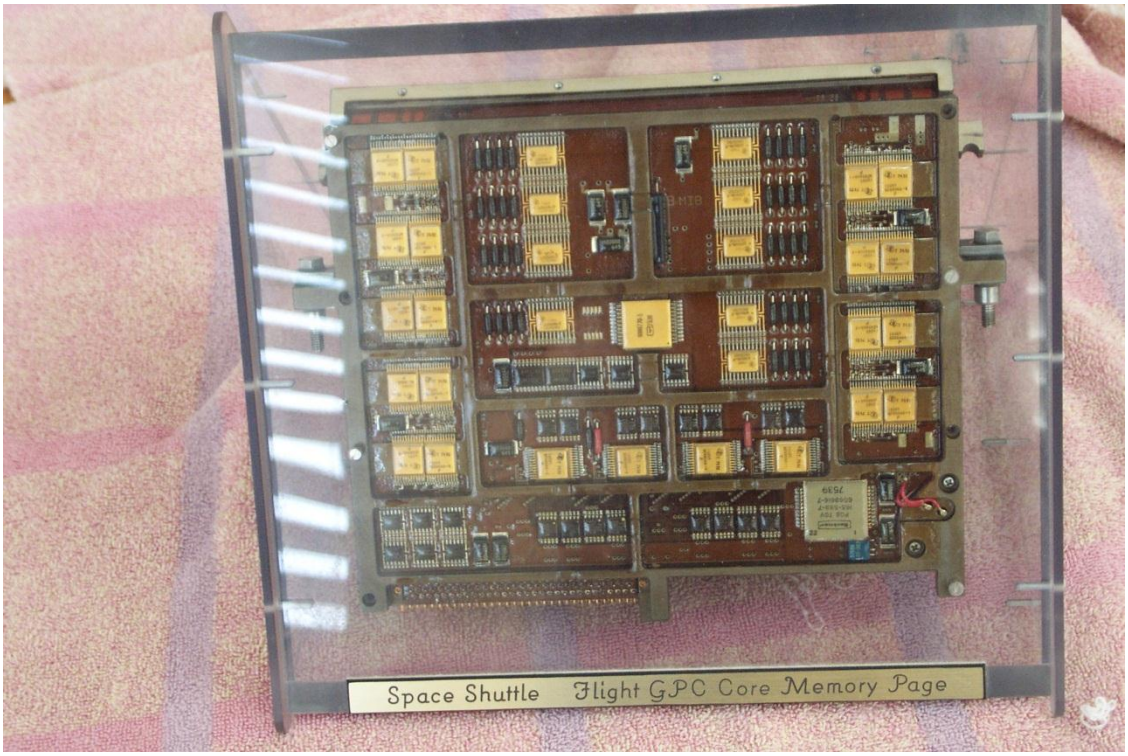
ックは基部からアドレスの下位へと成長し、グローバル変数などは 18 ワードのアドレス上位に確保される。スタックアドレスは R0 レジスタに格納される。R5,R6,R7 が整数演算用に予約されている。

シャトルでは同じ AP-101 を 5 台搭載して、うち 4 台の出力を多数決冗長で 1 台分の出力にして出力し、4 台に同じデータがそれぞれ入力される。残り 1 台は待機冗長で、多数決には加わらず、4 台のどれか 1 台が壊れたときに取り替えるためのいわば予備部品である。

I/O チャンネルの接続先は主に DPS と呼ばれる、インテリジェンスなパラレル-シリアル変換器である。DPS は 1 台で 16 個のパラレル入力をシリアライズし、16 個のパラレル出力をシリアル入力から作り出す事ができる。つまりアクセスは 1 ワード単位だ。DPS は 28 ビット長のコマンドによって動作を制御できる。パラレル出力制御は主にこれによる。

チャンネルには他にもプラウン管ディスプレイや、後にはデータストレージも接続する。

図 8：AP-101 用コアメモリ



シャトルに搭載されたのは開発初期から多少仕様変更された AP-101B だったが、1990 年代に新しい AP-101S に載せかえられている。AP-101S はメモリをコアメモリから半導体メモリに変更し、メモリ空間を 1 メガバイトまで拡張した。速度は 1MIPS まで向上し、サイズと重量は半分になった。しかしシャトルに用意されている搭載空間は変わらないから、以前の 1 台のモジュールケースの中に 2 台の AP-101S が入るという構成になっている。

DPS のパラレル入力は一度のアクセスで 16 ビット、1 ワード分を取得する。DPS に対して、HAL/S のカラムやライン、タブといった制御命令がどういう効果を持つのかは不明だ。また WRITE 文で例えば 1 ビットの ON/OFF をするようなビット操作はどうしたらいいのだろうか。恐らく 2 バイトの文字列型変数を BIT 文で操作したものを出力する事になるだろう。

AP-101 が 24 本のチャンネルを持つのに対して、HAL/S は 10 本しかサポートしない。恐らくドキュメントの間違いか、AP-101 用に別にドキュメントがあるのだろう。元々チャンネルが 10 本というのはシャトルに対して少な過ぎる。

HAL/S はテレタイプ端末やプリンタを繋ぐには向いているが、決してハードウェアの生 I/O を接続するのには向いていない。ハードウェア寄りの仕様に一切歩み寄らなかった HAL/S という言語は、宇宙機用汎用言語という掲げた目標と比べると、ちぐはぐというか、訳がわからないといった感想をどうしても持ってしまう。

HAL/S で記述されたプログラムは、基本的には IBM System/360 か 370 上で開発することになる。少なくとも EBCDIC プリンタが接続した機械が必要で、実際には開発は紙の上でだっただろう。プログラムは IBM パンチカードに 1 行 1 枚の勘定で打ち出され、コンパイラはそれらカードの束の上に置かれた JCL 言語の制御カードに従ってコンパイルする。

気になるのはマルチライン記述であるが、恐らく各ライン毎に別のカードを割り当てたものと思われる。デバッグでどこかの行を書き直すと、カードの束の 1 枚とか数枚を新しいものに入れ替える事になる。こんな開発でバージョン管理なんてものはありえないし、そういったものは一切サポートされていなかった。最初のスクリーンエディタは 1960 年代末には登場していたが、そんな軟弱なものは HAL/S の開発環境には存在しなかっただろう。開発環境のマニュアルに書かれているのはただ、JCL の書き方のみである。

それでもアセンブラで開発するよりかは遥かにましだった。実際は当初 NASA ではアセンブラでの開発が強く主張されていた。組み込み用プログラミング言語は当時信頼性とパフォーマンスの両面でまったく信用されていなかった。

NASA ジョンソン宇宙センターは 1970 年に、有人宇宙ミッションの搭載計算機向けに、誘導と制御の数式処理と、信頼性のあるリアルタイム制御を実現するのが目標として高水準言語の採用を検討した。既存の言語と提案された新言語 HAL/S を一年半にわたって比較し、HAL/S はそうして新有人打ち上げシステムスペースシャトルの搭載計算機向けソフトウェアの開発言語として、その採用が決まった。パフォーマンステストで、アセンブラ記述より 5 パーセント速度が劣るだけという性能をベンチマークで示し、ようやく HAL/S の採用が決まったのである。

シャトルフライトソフトウェアは、1973 年に IBM とロックウェル社の合同プロジェクトとして NASA と契約、開発を開始した。そもそもオービターのメインカスタマーであるロックウェルは IBM を下請けに使うつもりだったのだが、IBM が NASA と単独契約をしようとしていたのに慌てて割り込んだのだ。だから開発主体は結局 IBM である。更に開発初期にはチャールズ・スターク・ドレーパー研究所がコンサルタントに入った。チャールズ・スターク・ドレーパー研究所はアポロ誘導コンピュータの開発主体であり、1973 年まで HAL/S 開発の主体である。HAL/S の開発者たちはこの年に Intermetrics 社を設立、独立している。HAL/S の一応の完成も同年となっている。

開発されたソフトウェアは OPS と呼ばれる複数のモジュールの塊からなっている。OPS は更に機体状態によって遷移するモードから構成されていた。例えば OPS-3、エントリーは最初の軌道離脱モードから始まって、複数の制御モードを遷移して着陸モードに到達する。

常に動作している OPS は 3 種類、OPS-0 はデータサンプリングとユーザインタフェイスを担当している。これは FCOS、フライトコンピュータオペレーティングシステムとも呼ばれる。OPS-9 はデータストレージコントローラだ。サービスモジュール用の 2 つの OPS は用途に合わせて呼び出される。機体制御用の OPS は 6 種類、これはうちどれか 1 つが必ず動作している。6 種類の中には地上整備時のコンフィギュレーション用の OPS もある。

OPS の切り替えはメモリ空間の制約のせいでもあった。打ち上げ時の機体制御プログラムは帰還時には必要ない。切り離す事が出来れば、一度にメモリ空間に置くプログラムサイズを小さく出来る。各プログラムモジュールは外部装置からロードして切り替えればいい。プログラムは絶対の前

提条件としてAP-101の64キロワード、つまり128キロバイトのメモリ空間に収まらなくてはならなかった。FCOSはうち35キロバイトを常に占めている。
その上で、プログラムの動作にはRAMが必要である。HAL/Sがローカル変数の機能を持ち、常にRAMを占拠するグローバル変数を減らせた点はプロジェクトへの貢献だった。

ロックウェルは1971年の段階では、プログラムサイズを56キロバイト、だからコンピュータのメモリは余裕を持って64キロバイトもあれば十分だと見積もっていた。しかしプログラム規模は1978年には実行時イメージで140キロバイトを突破していた。RAMの余裕を考慮するとプログラムは80キロバイト以内に収めろというのがNASAの要求だった。IBMは、大掛かりな機能分割によって、最初の打ち上げ前にこの要求を満たすことが出来た。

FCOSはリアルタイムシステムである。リアルタイムシステムにはその起源から、タイムスライスとイベントドリブンという相容れぬ2つの方式が存在している。ロックウェルはタイムスライスでのシステム構築を主張し、IBMはイベントドリブンでの構築を主張した。
ロックウェルの最初のバージョンは40ミリ秒で一周するタイムスライス固定ループだった。HAL/Sのリアルタイム機能など綺麗さっぱり無視したこのバージョンは、40ミリ秒のメインループはそのままで、IBMによって960ミリ秒のユーザインタフェイス対応タスクが追加された。両者の方式の戦いは二年間続き、最後には双方を採用して手打ちとなった。つまりタイムスライスとイベントドリブンの二つの処理が、二つの開発主体の溝そのままに共存する設計となったのだ。
シャトルフライトソフトウェアは、巨大なドキュメントの山によって完全に動作を定められていた。ドキュメントはA,B,Cの三レベルに分けられた。レベルAは機能要求とインタフェイス仕様である。これはNASAが中心となって策定した。Bレベルは詳細設計で、これはIBMが策定した。Cレベルは実装とほぼ同等である。
4台のAP-101はI/Oアクセスのたびに専用接続バスを通して、3ビットのメッセージを送出し、4ミリ秒他のAP-101の反応を待つ。メッセージの内容は"データ正常"であったり"異常入力"だったりする。これらメッセージとデータ内容から、多重冗長バスコントローラは、4入力の中から確からしい信号を更に多数決して外部に出力する。
もし4ミリ秒のうちに応答しないAP-101があれば、それは異常動作を疑われることになる。また、疑わしいメッセージを受けたら、AP-101はデータ汚染を防ぐために一定時間データバスを閉じる。この処理は40ミリ秒のタイムスライス部の担当であった。このプロセスは4台の歩調が合っている事が前提の動作である。内部タイマが4台ともほとんど合っていなければならない。もし1台がずれていても、やがて周囲に同期することが期待されていた。
更に4台は6.25ヘルツの周期で、64ビットのデータを交換し合い、互いにチェックした。このデータはエンジン情報、空力舵、スラスター等の情報をコンパクトにまとめたもので、このデータを三回他と食い違って出力したAP-101は失敗と判定、表示される。これをどうするか、例えば再起動はクルーの仕事である。

問題は最初の電源投入時である。コンピュータたちは全く別々のタイミングで起動する。1981年4月、最初のシャトル打ち上げの予定時刻20分前、コンピュータに火を入れるはずが全然起動しなかった。4台のコンピュータが順番に互いにバスを閉じ合って、同期に失敗するデッドロックに陥ったのである。

症状は即座にデッドロックだと診断された。打ち上げは 2 日延ばされ、ソフトウェアはそのままに打ち上げられた。

後の解析によると、電源投入は 67 分の 1 の確率でデッドロックにより失敗する事が判明した。初期化コードの肥大化によって起動に要する時間が延びて、初期の設計に無いデッドロックタイミングが生じたと診断された。調査すると、デッドロックは三ヶ月前にも一度開発中に起きていた。

その後もバグは発覚し続けたが、ソフトウェアの修正により新しいリスクを抱え込むより、テスト済みであるプログラムをそのまま使い続ける方針が維持された。バグは発見され次第、それを避ける詳細な手順書が作成された。STS-7 ミッションの時には、それは既に 200 ページにも渡っていた。バグの実際の修正はチャレンジャー事故後に一括して行われた。

シャトルフライトソフトウェアの、ロックウェルによる最初の開発費見積もりは二千万ドルだったが、最終的に開発費用は二億ドルにまで膨れ上がった。コード規模は削減の努力にも関わらず、HAL/S のコードにしておよそ 400000 行、500000 ワードの規模にまで成長した。但しこれは機能分割後の総和であり、FCOS など共通で使用するコードが重複して含まれた値である。従ってテストの対象となったコードはその半分ほどになる。NASA はシャトルフライトソフトウェアに 1 行あたりおよそ 1000 ドル相当のコストをかけた事になる。

現在このプログラムは”バグフリー”であるという評価を得ている。このプログラムがこの世で最も徹底的にテストされたプログラムの一つである事は間違いない。

以上の記事内容は、主に以下の資料を参照して執筆した。

HAL/S Compiler System Specification
http://klabs.org/richcontent/software_content/hal_s/hal-s_compiler_system_specification.pdf
HAL/S Language Specification
http://klabs.org/richcontent/software_content/hal_s/hal-s_language_specification.pdf
HAL/S Programmer's Guide
http://klabs.org/richcontent/software_content/hal_s/hal-s_programmers_guide.pdf
HAL/S-FC User's Manual
http://klabs.org/richcontent/software_content/hal_s/hal-s-fc_users_manual.pdf
Programming in HAL/S
http://klabs.org/richcontent/software_content/hal_s/programming_in_hal-s.pdf

10：終わりに

400000 行の HAL/S ソースコードは、400000 枚の IBM パンチカードを意味する。HAL/S が分割コンパイルをサポートしていなかったらと考えると恐ろしい。プログラマ達にとっては、HAL/S のコードは地獄行きの契約書に等しかっただろう。

HAL/S には頑健なコードを書くための要素が多く欠落している。C の構造体や共用体がどれだけ素晴らしい機能であるか、万人が知るべきである。組み込みプログラマはレジスタに値をセットするとき、それに HAL/S を使わないで良い事をカーニハンとリッチーに感謝すべきだ。C のプリプロセッサマクロにすら感謝の祈りを忘れるべきではない。C のポインタの危険性を剥き身のナイフに例えるなら、HAL/S の COMPOOL はピンが最初から抜かれている手榴弾に例えられるべきだ。EXTERNAL 宣言された COMPOOL が、いちいち再度変数宣言する必要があった事に気づいただろうか。つまり間違ってオリジナルと違う宣言を書いてしまう可能性がある。C のヘッダファイルをインクルードする時には、パンチカードに同じ穴をコピペしていた HAL/S のプログラマ達の事を想って欲しい。

HAL/S はリアルタイム言語としても未成熟だった。リアルタイム OS に必要な機能は、割り込みへの応答速度やタイムスライスの精度では無い。情報こそが制御の中心であり、この安全がなければ制御は安全ではない。今でも時折想像力の欠落した組み込み技術者がやらかすヘマだが、セマフォを使わずタイムスライスで全てが解決するといった考え方は、安全から遠くかけ離れている。

HAL/S の開発者たち本人が、ソフトウェア開発に最初から関わっていれば、HAL/S の仕様の半分、つまり無駄は存在しなかっただろう。もしシャトルフライトソフトウェア開発以前に、練習になる開発計画があれば、HAL/S はもっと素晴らしい言語になっていただろう。残念ながら HAL/S は学び成長する準備期間抜きにシャトル計画に、NASA と IBM の官僚機構の歯車に飲み込まれ、仕様を凍結されたのだ。

シャトルフライトソフトウェアが HAL/S を使いながら危険な事故を起こさなかった理由の一つは、その規模だろう。しょせん 1970 年代の組込みシステムなのだ。規模はたかが知れている。機能分割はソフトウェアのモジュール化を推進した。更に金と人員さえ掛ければ、バグは潰すことが出来る。

400000 行の HAL/S ソースコードは、現代的な言語なら更に簡潔に記述できるだろうし、考えてみれば 128 キロバイトしかメモリ空間が無いのだ。FCOS など 35 キロバイトに過ぎない。AP-101 のコード効率が良かった筈が無い。シャトルフライトソフトウェアは、そこらの組み込みシステムよりずっと小規模なシステムなのである。

詳細なドキュメントと紙上の検討がどれほど信頼性に寄与したのかはわからない。徹底した試験とシミュレーションの寄与も、どのような試験を行ったか、その情報をも含めてわからない。ただ言えるのは、仕様は基本的に間違っておらず、試験も必要な範囲を網羅した筈だという事である。

これは双方とも、実は難しい条件である。技術的にはまったく難しい条件ではない。それは政治的、経済的な条件であり、結局それがソフトウェアプロジェクトの成就を左右するのだ。上級関係者の単なる理解不足が、政治的歯車の中でプロジェクトを超難問に仕立て上げる。HAL/S の仕様も、そういう政治的歯車の産物の一種である。ただ、シャトルフライトソフトウェアの開発が HAL/S という高級言語で行えたことは成功と言っても良かったのかも知れない。もしアセンブラで開発が行われていたら、プロジェクトは果たしてどうなっていただろうか。……いや、それでもうまく行っていたのかも知れない。開発期間やコストはまったく違うものになっていただろうが。

米国防省は軍用の高水準プログラミング言語を求めて1975年から1976年にワーキンググループを作ったことがある。候補言語のリストにはFORTRAN、COBOL、PL/I、ALGOL68、PASCAL、SIMULAといったメジャーな面子に交じってTACPL、JOVIAL、CORAL、SPL/I、ECL、EUCLIDといった消え去ったエキゾチックな名前も見える。HAL/Sも当然混じっていて高い評価を受けている。しかし当時既に存在していた筈のC言語はこのリストにはない。

このレポートで、HAL/Sは実績のあるリアルタイム組み込み言語で、言語仕様はクリーン、マニュアルもよく整理されてあるとべた褒めになっている。欠点としては再帰ができないことがまず第一に挙げられている。HAL/Sの設計年代である1960年代後半では再帰はトリッキーな技巧に過ぎなかったが、1970年代後半には大容量データアクセスに欠かすことのできない機能と認識されていたのだ。またHAL/Sはユーザ定義型が無く、データのカプセル化の手段がないと指摘している。マルチタスクとリアルタイム機能にもアップデートが必要だという指摘も忘れていない。

実績は実際にはまだ存在していなかった。そしてシャトルの初飛行は1981年である。

シャトルフライトソフトウェアの開発はコストの問題はあるにしても、安全にその使命を果たしたという意味では成功だった。しかしHAL/Sそのものは、やはり失敗した開発である。宇宙機向け汎用言語という当初の目的を果たせず、組み込みソフトウェアの世界でHAL/Sはほとんど影響を与えることができなかった。

多分、酷い言語、失敗したプログラミング言語を普段見ることが無いのは、そういう言語はそもそも誰にも使われないからだ。ただ、仕方なく使用を強制されるものだけが失敗した言語として目に見える形で残るのだ。

Intermetrics社はその後、Ada83及びAda95規格の策定に関わり、Adaコンパイラを出荷した。Ada言語にはHAL/Sの特徴の幾つか、例えばリアルタイム言語機能などが見られる。Intermetrics社は更にその後1997年にルッキンググラス社と合併、コンピュータゲーム開発企業になり、PC用フライトシミュレータやスニーキングアクションゲーム"シーフ"、RPG"ウルテイマアンダーワールド"、N64向け"コマンド&コンカー"等を開発した後、1999年に再び分離、売却され軍事サービス企業タイタンコーポレーションに買収された。タイタンは軍のために翻訳ソフトウェア、空中警戒管制システム、海戦ウォーゲーム等を開発している。また国土安全保障省との契約を介してアブグレイブの囚人虐待にも関わった。

HAL/Sは珍しい、死亡日時がはっきりしたプログラミング言語となる。STS-135向けのミッション用コード差分の最後のものが書かれた瞬間に、HAL/Sの死は確定する。命日は2011年7月21日、最後のシャトルOV-104アトランティス搭載のAP-101Sから火が落とされる瞬間だった。これより先、HAL/Sでコードが記述されることは二度と無いだろう。

2011年10月 水城徹