Final Report

A STUDY OF PROGRAMMING LANGUAGE EFFECTIVENESS

Bruce W. Arden
Principal Investigator

James Hamilton

ORA Project 03222

under contract with:

U. S. ARMY SAFEGUARD SYSTEMS COMMAND
CONTRACT NO. DAHC60-70-C-0036
CONTRACTS OFFICE SSC-C
HUNTSVILLE, ALABAMA

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

May 1970
A. INTRODUCTION

The goal of this study is to compare the effectiveness of three programming languages, in terms of object code efficiency. There has been very little effort to evaluate the ease of coding in the three languages, although number of source lines is some measure of this.

The study consists of three separate problems, each coded in three languages: Assembler, FORTRAN IV, and PL/1. Each problem was tested for several different cases (data sets), in order to test each of the paths in the logic of the program.

The object machine is the IBM 360 Model 67, running under the Michigan Terminal System (MTS). The language processors used were:

1. MTS G Assembler.
2. FORTRAN H Compiler.
3. PL/1 F Compiler, version .

No statistics are given on the relation efficiency of these processors.
B. PROBLEM SPECIFICATION AND CONSTRAINTS

The problems were given by complete flowcharts, which were abstracted from the Safeguard ABM system. The data structures were also specified. Very little variation was allowed in program structure; the flowcharts were followed quite closely. Greater variation was allowed in the implementation of the data structures, and in an effort to evaluate the effects of data structure constraints, two of the problems were coded in two versions—one a strict interpretation of the data structure specification, and the other, a fairly free interpretation.

A brief verbal description of the three problems follows:

1. GET TASK—A task scheduler for a multiprocessor system. This essentially implements a pert-like network of tasks and routines, in which each task is conditionally enabled by each of its predecessors, and absolutely enabled when it has been conditionally enabled by all of its predecessors. The task is then started and passed the address of its data set. This is intended to be a list processing problem, but it has bit manipulation aspects also.

2. EXLIGEN—The execution list generator. This is part of a collection of tasks which ultimately determines the sequency of pulses to be transmitted by the radar. It processes a radar template and pulse pattern table, and produces an execution list of radar "events," ordered by priority and timing constraints, etc. This is intended to be a bit manipulation problem, but bit manipulation is not inherently part of the problem, and the free interpretation of the data structure requires no bit manipulation at all. Instead it involves a fairly complex flow of control.

3. FILTER—A Kalman filter which is a part of the radar tracking processing. This is a very simple program using little more than floating point arithmetic. It is intended as a numerical problem.
C. DRIVER PROGRAMS

For each problem a driver program was written to test the programs and to collecting timing statistics. There were separate driver programs for each problem, but, in general, they all went through the following steps:

1. Read input specifying:
   a. Whether or not to provide trace output verifying the correctness of the program.
   b. The number of iterations for which each case is to be run, thus specifying also the number of cases.

2. Set up the data structure, and execute each case for the indicated number of iterations, printing the average time for each case.

The driver for problems 1 and 2 was written in assembly language, and was identical for the three subject languages, except that the driver for the PL/1 version had to be modified to interface properly with the PL/1 library. The driver for problem 3 was written in PL/1.

The timer used was essentially the 360-interval timer, which has an increment of 13 μsec. The MTS supervisor attempts to simulate this timer for each task, in such a way that it indicates only the CPU time used by that task, but inevitably some of the supervisor overhead for other MTS tasks will be included in this time supply because, at the time of an interrupt, the interrupted task state must be preserved before the timer is sampled.

Part of the time, measured by the timer, actually includes the time required to execute the driver program and timer processing routines. In an attempt to exclude this component, a run was made for each problem, in which the subject program was replaced by a null program (one which just returns). The timing statistics resulting from these runs were subtracted from those measured with the subject program. This yields the "adjusted" time listed in the table of results.
D. RESULTS

<table>
<thead>
<tr>
<th>GET TASK (strict data structure)</th>
<th>ASSEMBLER</th>
<th>FORTRAN</th>
<th>PL/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines$^a$</td>
<td>79</td>
<td>50</td>
<td>48</td>
</tr>
<tr>
<td>Object instructions</td>
<td>75</td>
<td>251</td>
<td>327</td>
</tr>
<tr>
<td>Object size$^j$</td>
<td>632</td>
<td>1254</td>
<td>1786$^b$</td>
</tr>
<tr>
<td>Time - Case 1$^c$</td>
<td>2.37</td>
<td>4.51</td>
<td>8.41</td>
</tr>
<tr>
<td>Time - Case 2</td>
<td>2.35</td>
<td>4.10</td>
<td>7.04</td>
</tr>
<tr>
<td>Adjusted time Case 1$^d$</td>
<td>0.70</td>
<td>2.84</td>
<td>6.74</td>
</tr>
<tr>
<td>No. of subroutines$^e$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explicit</td>
<td>0</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Implicit</td>
<td>0</td>
<td>0</td>
<td>7$^f$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GET TASK (free data structure)</th>
<th>ASSEMBLER</th>
<th>FORTRAN</th>
<th>PL/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>67</td>
<td>39</td>
<td>48</td>
</tr>
<tr>
<td>Object instructions</td>
<td>65</td>
<td>170</td>
<td>292</td>
</tr>
<tr>
<td>Object size</td>
<td>332</td>
<td>1036</td>
<td>1650</td>
</tr>
<tr>
<td>Time - Case 1</td>
<td>2.16</td>
<td>3.42</td>
<td>4.81</td>
</tr>
<tr>
<td>Time - Case 2</td>
<td>2.15</td>
<td>3.45</td>
<td>4.86</td>
</tr>
<tr>
<td>Adjusted time Case 1$^d$</td>
<td>0.49</td>
<td>1.78</td>
<td>3.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EXLIGEN (strict data structure)$^i$</th>
<th>ASSEMBLER</th>
<th>FORTRAN</th>
<th>PL/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>259</td>
<td>122</td>
<td>133</td>
</tr>
<tr>
<td>Object instructions</td>
<td>228</td>
<td>477</td>
<td>1100</td>
</tr>
<tr>
<td>Object size</td>
<td>1220</td>
<td>2618</td>
<td>4918</td>
</tr>
<tr>
<td>Time - Case 1</td>
<td>1.83</td>
<td>4.98</td>
<td>19.2</td>
</tr>
<tr>
<td>Time - Case 2</td>
<td>1.75</td>
<td>4.90</td>
<td>18.46</td>
</tr>
<tr>
<td>External references</td>
<td>0</td>
<td>1</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EXLIGEN (free data structure)$^i$</th>
<th>ASSEMBLER</th>
<th>FORTRAN</th>
<th>PL/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source lines</td>
<td>228</td>
<td>126</td>
<td>130</td>
</tr>
<tr>
<td>Object instructions</td>
<td>200</td>
<td>405</td>
<td>783</td>
</tr>
<tr>
<td>Object size</td>
<td>1104</td>
<td>2228</td>
<td>4295</td>
</tr>
<tr>
<td>External data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1036</td>
<td>1214</td>
<td>952</td>
</tr>
<tr>
<td>Time - Case 1</td>
<td>1.42</td>
<td>2.86</td>
<td>7.03</td>
</tr>
<tr>
<td>Time - Case 2</td>
<td>1.40</td>
<td>2.89</td>
<td>6.84</td>
</tr>
<tr>
<td>External references</td>
<td>0</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>
FILTER

<table>
<thead>
<tr>
<th>Source lines&lt;sup&gt;a&lt;/sup&gt;</th>
<th>ASSEMBLER</th>
<th>FORTRAN</th>
<th>PL/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>27</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Object instructions</td>
<td>89</td>
<td>134</td>
<td>351</td>
</tr>
<tr>
<td>Object size</td>
<td>364</td>
<td>774</td>
<td>1786&lt;sup&gt;b&lt;/sup&gt;</td>
</tr>
<tr>
<td>Time</td>
<td>1.86</td>
<td>2.51</td>
<td>3.44</td>
</tr>
<tr>
<td>Adjusted&lt;sup&gt;e&lt;/sup&gt;</td>
<td>0.74</td>
<td>1.39</td>
<td>2.32</td>
</tr>
<tr>
<td>External references</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

AVERAGE RATIOS

<table>
<thead>
<tr>
<th>Source lines</th>
<th>ASSEMBLER</th>
<th>FORTRAN</th>
<th>PL/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>2.2</td>
<td>4.3</td>
<td></td>
</tr>
<tr>
<td>Object instructions</td>
<td>2.2</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>Object size</td>
<td>2.7</td>
<td>7.4</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:

a. Excluding comments.

b. Includes static storage csect and program csect.

c. All times are in milliseconds.

d. Adjusted time equals total time minus fixed time due to driver and timer routines (1.67 msec).

e. Does not include calls to RTEQ and EOT, the two driver entries.

f. Actually, the number of external references. Not all of these are actually called by the program. (In fact none are called by FILTER.) All must be loaded, however, and the PL/1 library required turns out to be about 40K bytes, only a small amount of which is actually used.

g. Fixed time for FILTER driver - 1.12 msec.

h. For each of the problems that part of the data structure which was specified by the problem was external to the procedure itself. In problems I and III this quantity was essentially constant, even in the review GET TASK.

i. The fixed time for the Problem II driver was negligible (0.016 msec), so no adjusted time is given.

j. Object size is bytes, decimal.
E. CONCLUSIONS

FORTRAN II appears to produce roughly twice as much object code as the Assembler, for about half as many source lines, and PL/1 produces about four times as much, for, again, about half the source code. The ratios in execution time are even worse. In an attempt to provide general explanations for these differences, the following five categories are proposed:

1. Problem specification constraints.

2. Language restrictions.

3. Language generality.

4. Inefficient compilation—difficulty of compiling for 360; inherent and compiler induced object code inefficiencies.

5. Programmer bias and unfamiliarity with language features.

In more detail:

1. When designing systems, one takes language abilities into account. Both program and data structure are heavily dependent upon language considerations. The programming language dictates, to a large extent, the style of programming. A LISP program, for example, will typically be a collection of recursive "functionals," PL/1 programs may have complex data structures, and several levels of DO group nesting, procedures, and block structure, and FORTRAN offers only a primitive DO loop. But the program structure for these problems was specified by flowcharts taken from assembly language programs. This structure did not allow the use of PL/1 block structure, and in many cases did not even allow simple DO loops in places where DO loops would be useful. This can only add to the differences due to the other four possibilities.

Data structure constraints are probably even more responsible for differences of this type. FORTRAN, for example, deals only with arrays and simple variables, and it is unfair to ask more of it than this. In an attempt to determine the effects of data structure variation, the problem I and problem II programs were revised, to allow a completely free data structure. All three languages appear to benefit about equally in terms of object instructions, and object size, but in execution time, the higher level languages benefit somewhat more than the assembler versions. In the problems at hand, the only important data structure variation involved removing the necessity for some bit manipulation. Therefore, the effects of this variation on the three languages depends on the ability of the language to handle
bit manipulation. PL/1 has this ability in the language, but its implementation tends to be extremely inefficient. FORTRAN does not have it at all, so it was necessary to write small assembler language programs to perform the bit string functions. This tended to benefit the FORTRAN versions, since the functions written were more specialized than the very general bit string routines which implement the PL/1 functions.

Thus one concludes that problem specification constraints tend to exaggerate the differences between languages. This should not be interpreted, however, as meaning that the problems themselves are inherently difficult to code in higher level languages, but only that the specifications of this study prevented the most efficient use of such languages.

2. Language restrictions. One might think that a perfect compiler could compile a program equal in efficiency to an assembly language program. But language restrictions, and, to some extent, language generality, prevent this. Most languages restrict the parameters of certain underlying functions in such a way that additional statements are required to preprocess data into the required form. Examples are the restriction of FORTRAN DO-loop parameters to simple integer variables or constants, and the restrictions of pointers, used as bases or in pointer qualification in PL/1, to be simple, nonbased, pointer variables. These restrictions result in extra processing which is irrelevant to the task at hand.

3. Language generality. One answer to the problem of language restrictions is to make the functions more general. PL/1, for example, has removed the restriction on the DO loop. But this requires either that the resulting object code be more complex to handle the more general inputs, or that the compiler recognize the simple cases when they occur. Unfortunately, the former case usually occurs, because it is very difficult, and sometimes impossible for the compiler to recognize possible simplifications.

Many modern languages have added many powerful functions, and much more varied data types, in an attempt to make the languages more generally useful. It is only fair in evaluating the language, to use these facilities wherever they are useful, even though it might be more efficient to do things differently. This has been done in PL/1, where such features are abundant. It would have been possible to translate the FORTRAN programs directly to PL/1, resulting in more efficient PL/1 programs, but using none of the power of PL/1; but it was felt that this would not be a true evaluation of PL/1.

It turns out that the most natural way to generalize language primitives is to add levels of "indirection" in data references. In PL/1 this means that the more complex data structures are referenced via dope vectors. This turns out to be unnecessary in many cases, but it is done just the same. This is perhaps the most common form of inefficiency due to language generality.
4. Still, it is abundantly clear that the available compilers do not recognize many of the possible simplifications, even where it would be simple to do so. FORTRAN H, for example, does not implement multiplication and division by powers of two as shifts, and PL/1 puts out many instructions which might as well be No-ops. Sloppy compilation is still the greatest contributor to the differences observed here.

It has been suggested that the IBM 360, and machines like it are difficult to compile for, either because of the base-displacement addressing, which requires checking of addressability, or because the 360 instruction set is not well matched to the underlying procedure oriented language primitives. There seems to be evidence for this in the fact that FORTRAN compilers for other machines seem to produce code which rivals assembly language efficiency. However, a study of the object code from these problems does not show any clear instances of language-machine mismatch.

It is true, beyond a doubt, however, that the 360 has many instructions which do not correspond to any language primitives. Examples include translate and test, and branch on count. These instructions are very useful to the assembly language programmer, but useless to FORTRAN or PL/1. It generally requires several FORTRAN statements to perform the equivalent function. This is not so much an incompatibility, as the inability of the compiler to use the full 360 instruction set.

5. It is only fair to point out that the programmer is biased toward assembly language programming, and is certainly most conversant in it. The programmer is quite familiar with FORTRAN, however, and it is felt that the FORTRAN programs are reasonably effective for this reason. PL/1, however, is a different matter. While he is reasonably familiar with the facilities of that rather complex language, he is by no means an expert. To cite a specific example, one of the biggest inefficiencies in the PL/1 object code is its handling of bit strings with implicit calls on external functions, which requires creation of dope-vectors, etc. It turns out that if the bit strings are declared ALIGNED, in-line code is generated. This reduced the execution time in problem I from 11.99 msec to 8.41 msec. It is not known whether there are any more changes of this nature which would so significantly increase the efficiency of the PL/1 programs. One might argue, however, that if intimacy with the details of a language are required in order to use it effectively, then that makes the language that much less desirable.
APPENDIX

FLOWCHARTS
**GET TASK DRIVER - Page 2**

Entry RTEX2

This Subroutine "Executes" a Routine

No

Trace = 'T'

Yes

Insert Routine Name and Data Set Name Into Trace Message

Print Trace Message

Record Time

Return

Entry EDT

This Subroutine Records "End of Task"

Trace = 'T'

Yes

Print 'End of Task'

Record Time

Return

11
AEB TABLE

Displacement of Bit

CEB Word for Task
A (Block List Ent. (1st RTN))

TASK LIBRARY TABLE (TLT)

BLOCK LIST
(Routine Printer Blocks)

A(PCAT Ent.) A(BALT Ent.)
- BLINC A(Succ. List)
- Time Reqts. # Successors

SET SUCCESS LIST

A(TLT Ent.) for Succ.
CEB Mask

PROGRAM COPY ADDRESS TABLE (PCAT)

A(Code) Size

NOTE: "ENT" = "Entry"
A(x) = Address of x

DATA SET ADDRESS TABLE (DSAT)

A(Data Set)

BASE ADDRESS LOCATOR TABLE (BALT)

A(DSAT Ent.)

TABLE STRUCTURE - GET TASK

Data Set

Actual Code
FILTER DRIVER

Read Trace, No. Iterations, No. of Steps, (AVAR RVAR for Each Step), Initial Data

Initialize Timer

ITCNT ← 1

Move Data Into Position

StepCNT ← 1

Call Filter (AVAR (StepCNT), RVAR (StepCNT))

Trace = 'T'

Yes

Print Results

StepCNT ← StepCnt + 1

No

StepCNT > No. Steps

ITCNT ← ITCNT + 1

No

ITCNT > No. of Iterations

Compute and Print Time

Exit
FILTER FLOWCHART

\[ R_{\text{Fact}} \leftarrow 1/(RVAR + 2R1) \]
\[ E_{\text{Fact}} \leftarrow 1/(Theta*AVAR + QB2) \]

\[ WR_i \leftarrow Q_i^* R_{\text{Fact}} \]
\[ WE_i \leftarrow QE_i^* E_{\text{Fact}} \]
\[ \text{for } i = 1, 2, 3 \]

\[ R_i \leftarrow R_i + WR_i^* \text{DELRNC} \]
\[ E_i \leftarrow E_i + WE_i^* \text{DELSNA} \]
\[ A_i \leftarrow A_i + WE_i^* \text{DELSNB} \]
\[ \text{for } i = 1, 2 \]

\[ A_R \leftarrow 1 - WR_1 \]
\[ A_E \leftarrow 1 - WE_1 \]

\[ 1A \]

\[ QE4 \leftarrow (QE4 - WE_2^* QE2)/Theta \]
\[ QE5 \leftarrow (QE5 - WE_2^* QE3)/Theta \]
\[ QE6 \leftarrow (QE6 - WE_3^* QE3)/Theta \]
\[ QR4 \leftarrow QR4 - WR2^* QR2 \]
\[ QR5 \leftarrow QR5 - WR2^* QR3 \]
\[ QR6 \leftarrow QR6 - WR3^* QR3 \]

\[ QR_i \leftarrow A_R^* QR_i \]
\[ QE_i \leftarrow A_E^* QE_i \]
\[ \text{for } i = 1, 2, 3 \]

END
EXECUTION LIST GENERATOR
FLOWCHART

Start

Is Template Passed In

Yes

Return

No

(LET 'TNUM' = Templ. #)

XPTR Address of the Particular Area in EXList to be used for this Template From Template (TNUM)

(There Are Four Areas Provided, One for Each 'Sub' Template)

Is the 'OUT' Bit Set in Template (TNUM)?

Yes

Return

No

IA

XLISTIM ← 0
XLISTEXT ← 0
StartX ← XPTR + 4
StopX ← XPTR + 4

Zero Out 1st 2 Words of All Work Areas

IC ← 0
EC ← 0

FISStart ← Start(TIC)
FISStop ← Stop(TIC)

Start ← FISStart + 200
Cl ← Template Link Portion of Template (TNUM)

1B

Cl EC PPP
Edit Out From
Template (Cl)

1D

Utype ← Usertype Field of PULPAT1(PPP)

3C

Yes

Utype > 9

Yes

2A

No
Is "Inactive" Bit of PULPAT (PPP) Off?

Yes

CHAN  
FCHAN  
UserID

Edit out From
PULPAT(PPP)

XMITT  > CCHAN

No

(Test to See if XMITT 
Time Falls Before Adjusted 
FStart)

CSTOP < FSTOP

-GATEW -GateWidth(Type)

XMITT < FSTART

Yes

No

(Test to See if XMITT Time + Gate Width Will Fall Beyond FSTOP)

CSTOP

Set Up A Work Area 
Corresponding to User Type 
(Let WA Denote the 
Work Area)

WA(0)  
WA(1)  
CHAN

WA(WA(0))  
UserID

WA(WA(0)) + 1  
CHAN

EC - 1

PPP + 1

1B

EC

≥ 0?

No

1B

EC

≥ 0?

No

Yes

FStart <- XMITT - 200

3A

2F

2D

Does XMITT Fail 
Before Unadjusted 
FStart?

No

3B

Yes

IC = 0?

No

Yes

FStart <- XMITT

(Are We in Format 
Interval Zero?)

IAT This Point We Have Established 
That:

1) XMITT < Start

2) XMITT > FStart

3) IC > 0

We Have "Moved Back" FStart, 
and Now We Must Check the Last 
Entry Placed in ExList to See if We 
Can "Get Away" With This Adjustment.
(CANNOT MAKE ADJUSTMENT)

If $\text{XList } + \text{XLINText} > \text{FISStart}$?

Yes -> 3B

$\text{TEMPERR} \leftarrow \text{AIPULPAT}(\text{PPP})$

Yes -> 2G

$\text{UType} > 9$?

No -> 3C

Is inactive Bit of PULPAT (PPP) Off?

Yes -> 4A

TestT

TINTVL

Edit Out From PULPAT(PPP), PULPAT(PPP + 1)

No -> 3D

Start $\leftarrow \text{FISStart} + 200$

No $\leftarrow \text{FISStart}$

Stop From Previous Interval (Now Stored in EXTTEMP)

Start $\leftarrow \text{FISStart} + 200$

Yes -> 3E

$\text{TINTVL} \leftarrow \text{WA(III)}$

$\text{WA(III) + 1} \leftarrow \text{TestT/60}$

No -> 3D

$\text{TestT} > \text{CStop}$

Yes -> 5A

$\text{TestT} > \text{FISStop}$?

Yes -> 4B

No $\leftarrow \text{FISStop}$

(Return to Either Test Entry Processing or Tactical Entry Processing)

(If the XMIT of the Last Entry + its GateWidth or Interval Extent) is Less Than the Adjusted FISStart, Adjustment was Legitimate)
4A

IC = 0

Yes → 3B

No → Temp ← F1Start - 200

TestT > Temp

No → 3B

Yes → F1Start ← Temp

3A

4B

IC = 3

Yes → 3B

No → Temp ← TestT + TINTVL + 15

StopT (IC + 1)

No → 3B

Yes → F1Stop ← Temp

3E
Calculate Address for 4 EXList Header Words in EXT temporarily.

\[ \text{XStartX}(IC) \leftarrow \text{StartX} \]
\[ \text{XStopX}(IC) \leftarrow \text{StopX} \]
\[ \text{XStartT}(IC) \leftarrow \text{FStart} \]
\[ \text{XStopT}(IC) \leftarrow \text{FStop} \]

(One Complete EXList Header Stored in EXT temporarily for Each Format Interval.)

\[ \text{StartX} \leftarrow \text{StopX} \]

\[ IC \leftarrow IC + 1 \]

\[ IC < 4 \]

Zero Out 1st 2 Words of WA's

\[ \text{FStart} \leftarrow \text{FStop} \]
\[ \text{Start} \leftarrow \text{FStart} + 200 \]

\[ \text{FStop} \leftarrow \text{StopT}(IC) \]

\[ EC > 0 \]

1D Yes

18 No

\[ CI = 0 \]

8A Yes
These entries made for Remaining Intervals. All of Which Have No Pulses.

XSTARTX(IC) ← STOPX
XSTOPX(IC) ← STOPX
XSTARTT(IC) ← STARTT(IC)
XSTOPT(IC) ← STOPT(IC)

IC ← IC + 1

IC < 4 ?

Yes

No

Set 'Old Bit' in Template (TNUM)

EXIT
EXECUTION LIST GENERATOR - DATA STRUCTURE

EXTEMP

<table>
<thead>
<tr>
<th>Area #</th>
<th>Template #</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSTARTX₀</td>
<td></td>
</tr>
<tr>
<td>XSTOPX₀</td>
<td></td>
</tr>
<tr>
<td>XSTARTT₀</td>
<td></td>
</tr>
<tr>
<td>XSTOPT₀</td>
<td></td>
</tr>
</tbody>
</table>

Template Assignment Word

One of these EXLIST Headers for each of 4 Format Intervals

Separate Space reserved for 4 Templates

TEMPLATE

<table>
<thead>
<tr>
<th>Old New</th>
<th>A (Exlist Area)</th>
<th>Template Link</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Template Link Word

TYPICAL PULPAT ENTRY

<table>
<thead>
<tr>
<th>STARTT</th>
<th>STOPT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EXLIST

<table>
<thead>
<tr>
<th>STARTX</th>
<th>STOPX</th>
<th>STARTT</th>
<th>STOPT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TYPICAL WORK AREA

<table>
<thead>
<tr>
<th># Entry Words</th>
<th>Channel Time of Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>USERID</td>
<td>CHAN</td>
</tr>
</tbody>
</table>

Test Entries Contain TINTVL in this Word

GATEW

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

(Gate width = 10 for all Tactical Pulses)