Uppsala University
Department of Computer Systems

FPSCALC Version 2.00 User's Manual

Harmen van der Velde

January 30, 1997
Technical Report nr. DoCS 97/89

Abstract

This document describes the FPSCALC program, a program that is used to perform schedulability tests on task sets for real-time operating systems implementing fixed priority scheduling. It was written in the light of research done on Real-Time CAN-bus driven environments.

1 Introduction

When performing real-time systems fixed priority scheduling [Tin95] we need to do a lot of mathematical calculations to determine if our task sets are schedulable. For instance, if we take a task set like this[1]:

Task Name	T	C	D	P
A	250	14	50	1
B	500	50	200	2
C	800	90	400	3
D	800	20	800	4
E	1000	50	1000	5
F	2000	10	2000	6
G	2000	10	2000	7
H	2000	30	2000	8

Table 1: An example task set

This example has priorities assigned according to deadline monotonic priority ordering (earliest deadline has the highest priority). We use the following formula to determine the response time of a task i:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

The results for the consecutive iterations n are denoted as R_i^n. The initial values are $R_i^0 = 0$.

For example, we can now calculate the value of R_C:

$$R_C^1 = C_C + 0 = 90$$

We iterate the solution a second time:

$$R_C^2 = C_C + \left\lceil \frac{90}{T_A} \right\rceil C_A + \left\lceil \frac{90}{T_B} \right\rceil C_B = 154$$

And again:

$$R_C^3 = C_C + \left\lceil \frac{154}{T_A} \right\rceil C_A + \left\lceil \frac{154}{T_B} \right\rceil C_B = 154$$

The iteration has now ended since $R_C^2 = R_C^3$, so we can say that $R_C = 154$. We can also now say that R_C is schedulable, for $R_C < D_C$. To make sure the whole task set is schedulable, we will have to repeat this calculation for every single task. It is obvious that this involves a lot of manual calculation, which is rather mechanical. In fact, it is so mechanical that this job can easily done by a computer. Therefore, the FPSCALC program has been developed.

[1]A higher priority is represented by a lower number.

2 Functional Description of the FPSCALC Program

The FPSCALC program takes input from a user specified file, which should be organised as follows:

- One or more task sets.
- Each task set's relevant variables and (optional) their initial values.
- Each task set's set of formulas to be used for calculation.
- Optionally, extra data concerning semaphores, used to calculate the blocking factors for each task set.

2.1 Description of the Input File Grammar

The input file can describe one or more *systems*. Each system is represented by a set of variables and the formulae that compute these variables.

The input file is a text file that should should conform to the following grammar given in appendix A.

System names, variable names, semaphore names and task names do not conflict with each other. However the names should be anything other than the reserved words, including the two reserved indices: **i** and **j**. The former is reserved to indicate the main index while the latter indicates the secondary index. When used in a formula, the main index specifies that the calculation should be performed for each task of the task set. The secondary index j is used inside a summation only, indexing tasks of the task set according to the used priority scheme. This will be explained in detail later on in this document.

The order in which the system blocks are placed is of no importance. Comments are preceded by the '!' character. When this character is found on a line, everything following it on this line will be ignored.

Within each system, the following blocks can be placed:

- A **declarations** block.
- A **semaphores** block.
- An **initialise** block.
- A **formulas** block.

How these blocks are declared, is described in the following sections.

2.1.1 The declarations Block

Within the **declarations** block, the following symbols can be declared:

- Variables
- Tasks

The variables can be any of four types:

- Scalar
- Indexed
- Priority
- Blocking

The scalar variables can hold only one value, where the indexed variables are arrays. These arrays are indexed by the task names.

The user can only declare one **priority** variable as well as only one **blocking** variable. Both variables are indexable.

The **priority**-type variable is reserved to indicate the priority of each task. These priorities will be used inside of summation expressions, to be able to distinguish between lower and higher priority task. Also, when calculating blocking factors, the priorities are required. The priorities can be designated manually in the desired order, but we can also set up a formula to calculate the priorities dynamically.

The **blocking**-type variable represents the blocking factor of each task. If declared, the program will *automatically* calculate the blocking factor of each task on initialisation. The introduction of blocking factors requires the declaration of a **semaphores** block, which will be discussed later on. Of course, the user is free to *manually* include blocking factors. The blocking factor variable should then be declared as a common **indexed**-type variable, after which the user can initialise the blocking factors manually, or set up a formula to calculate them dynamically.

Together with the variables, the user can declare a set of task names. These names can later be used to index the different members of the indexable variables. The order in which these task names are declared, is of no importance. For instance, when sharing an indexed variable between two systems, no problems will occur when indexing task names have been declared in different orders in these two systems. The program will match each index name automatically. If the user neglects declaration of tasks, only the scalar variables can be used, since the indexed variables would lack indices - the task names.

2.1.2 The semaphores Block

Some real-time systems use semaphores for managing shared resources. These semaphores cause *blocking factors* [Tin95] that influence the schedulability of a

task set. The blocking factor of each task is the maximum delay it might experience because of another task that has locked a certain resource by acquiring the corresponding semaphore.
For example, a task set might have semaphores that are locked as displayed in table 2.

Semaphore	Locked by	Time locked
S2	A	3
S4	G	3
S1	C	9
S2	E	13
S3	E	4
S3	F	4
S4	B	1
S5	G	7
S5	H	7

Table 2: Example semaphore declaration

Using this table, we can calculate the blocking factors with the following formula:

$$B_i = \max_{\forall k \in lp(i) \forall s \in uses(k) | ceil(s) \geq pri(i)} (cs_{k,s})$$

As mentioned in the previous section, blocking factors can be manually specified by the user or automatically calculated by the tool. For automatic calculation, we declare the semaphores in our input file, using the **semaphore()** command. The first argument of the **semaphore()** command specifies the name of the semaphore, the second specifies the name of the task it is held by, and the third and final argument specifies the time it is held for.

In order to declare semaphores, the user must first declare both a **blocking**-type variable and a **priority**-type variable. These two variables are essential for calculation of the blocking factors. The program will then automatically perform calculation of the blocking factors.

Last but not least, we must alter our formula for the response time to include the blocking factors. For instance, we could add it to the formula we used in our first example. The scientific notation of it is:

$$R_i = B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

IMPORTANT NOTE!

When we use a formula to manipulate the task priorities dynamically, the blocking factors may change dynamically too. In case of automatic blocking factors calculation, the program will automatically update these factors after each iteration if such a formula is present.

2.1.3 The initialise Block

Before starting the calculation, all declared variables within a system can be given initial values. This is done in the **initialise** block. In case of indexed variables, each member of a variable can be given a value by indexing with the desired task index. The value can be a constant value, or an arithmetical expression.

Here are a few examples of initialisation:

```
ScalarVar = 5.0;
IndexVar[TaskIndexName] = -5/7 + 3.14159265;
AllMembersOne[i] = 1.0;
```

By indexing with the reserved index **i**, all members of an indexed variable will be initialised with the same value.

Variables that aren't given an initial value, will be set to zero.

2.1.4 The formulas Block

Formulas are denoted in standard scientific notation. The types of the variables left and right of the equal-sign should be the same, meaning that we cannot compute an indexed expression and store it in a scalar, for instance.

Apart from the standard arithmetical operations '+', '−', '∗' and '/', identifiers have been reserved for the following mathematical operations:

- **sigma(hp,** *expression***)** - summation over higher priority tasks
  For example, write **sigma(hp, R[i] / T[j])** to calculate:

  $$\sum_{\forall j \in hp(i)} \frac{R_i}{T_j}$$

  The expression for summation is given between the parentheses. The program will execute the summation for all tasks j with a higher priority as the current task i.

- **sigma(lp,** *expression***)** - summation over lower priority tasks
  For example, write **sigma(lp, R[i] / T[j])** to calculate:

  $$\sum_{\forall j \in lp(i)} \frac{R_i}{T_j}$$

  The expression for summation is given between the parentheses. The program will execute the summation for all tasks j with a lower priority as the current task i.

- **sigma(ep,** *expression***)** - summation over equal priority tasks
  For example, write **sigma(ep, R**[**i**] **/ T**[**j**]**)** to calculate:

  $$\sum_{\forall j \in ep(i)} \frac{R_i}{T_j}$$

  The expression for summation is given between the parentheses. The program will execute the summation for all tasks j with equal priority to the current task i.

- **sigma(all,** *expression***)** - summation over all tasks
  For example, write **sigma(all, R**[**i**] **/ T**[**j**]**)** to calculate:

  $$\sum_{\forall j \in alltasks} \frac{R_i}{T_j}$$

  The expression for summation is given between the parentheses. The program will execute the summation for all tasks, regardless of their priority.

- **ceiling(***expression***)** - the ceiling function
  For example, write **ceiling(R**[**i**] **/ T**[**j**]**)** to calculate:

  $$\left\lceil \frac{R_i}{T_j} \right\rceil$$

  This implements the ceiling operation, which returns the smallest integer greater than or equal to its operator.

- **floor(***expression***)** - the floor function
  For example, write **floor(R**[**i**] **/ T**[**j**]**)** to calculate:

  $$\left\lfloor \frac{R_i}{T_j} \right\rfloor$$

  This implements the floor operation, which returns the largest integer smaller than or equal to its operator.

- **min(***expression***)** - the min function
  For example, write **min(R**[**i**]**, T**[**i**]**)** to calculate:

  $$\min(R_i, T_i)$$

  This implements the min operation, which returns the smallest of its two operands.

- **max(***expression***)** - the max function
  For example, write **max(R**[**i**]**, T**[**j**]**)** to calculate:

  $$\max(R_i, T_i)$$

This implements the max operation, which returns the greatest of its two operands.

As you might have noticed, the formulas use two indexing parameters: **i** and **j**. These parameters are used by the program to index the various task-parameters. When the program starts, it will use **i** to address the current R_i being computed, while it uses **j** to address the various task-parameters during a summation.

However, there is one special way in which a specific element of an indexed variable can be addressed. For instance, if we want to address variable R_{task_name}, we should use the following notation in our formula:

R[task_name]

This method of indexing a variable is called *direct indexing.*

For each system, multiple formulas may be given the input file. When more than one formula is given, the program will sequentially iterate through all of them cyclically, until their results converge. This feature can be used to solve the *holistic scheduling problem* [Tin95].

2.1.5 Global Variables

When performing holistic scheduling analysis, we need to specify the interaction between the different components of a system. Since all variables within a system are completely isolated from each other, the concept of *global variables* has been defined. We can declare a certain variable *globally*, meaning that this variable is visible to all systems.

The declaration of global variables is done **before** the first system block. Global variables can only be declared of type indexed or scalar. Initialisation can be done within any system block. Note that a multiple initialisation will result in initialisation of the variable, according to the last initialisation given in the input file. Previous initialisation values will be overriden!

3 Running the Program

The syntax of the FPSCALC program is as follows:

```
fpscalc [-v] < input_file
```

The flag `-v` designates verbose mode. In this mode, the program will output the parsed initial values of all the variables after initialisation. It will also output the results of the calculations after each iteration. The program will read its input file from standard input.

After a correct input file has been set up, the program can be run.

3.1 An Example

Let us take the before mentioned task set with semaphores added to the input file and store it in a file named `definition_file.fps`. The input file will now look something like this:

```
! definition_file.fps - an example task set.

! Global variable declarations

scalar GlobalVar;


system testing {

   ! First and only system

   declarations {

      ! Variable declarations

      indexed Period, Deadline, CompTime, RespTime;
      blocking Blockvar;
      priority Priovar;

      ! Task declarations

      tasks A, B, C, D, E, F, G, H;

   }

   semaphores {
      semaphore(S2, A, 3);
      semaphore(S4, G, 3);
      semaphore(S1, C, 9);
      semaphore(S2, E, 13);
      semaphore(S3, E, 4);
      semaphore(S3, F, 4);
      semaphore(S4, B, 1);
      semaphore(S5, G, 7);
      semaphore(S5, H, 7);
   }

   initialise {
      Period[A] = 250;
      Period[B] = 500;
      Period[C] = 800;
      Period[D] = 800;
      Period[E] = 1000;
      Period[F] = 2000;
      Period[G] = 2000;
      Period[H] = 2000;

      Deadline[A] = 50;
      Deadline[B] = 200;
      Deadline[C] = 400;
      Deadline[D] = 800;
      Deadline[E] = 1000;
      Deadline[F] = 2000;
      Deadline[G] = 2000;
      Deadline[H] = 2000;

      Priovar[A] = 1;
      Priovar[B] = 2;
      Priovar[C] = 3;
      Priovar[D] = 4;
      Priovar[E] = 5;
      Priovar[F] = 6;
      Priovar[G] = 7;
      Priovar[H] = 8;

      CompTime[A] = 14;
      CompTime[B] = 50;
      CompTime[C] = 90;
      CompTime[D] = 20;
      CompTime[E] = 50;
      CompTime[F] = 10;
      CompTime[G] = 10;
      CompTime[H] = 30;
   }

! Formula(s) to be iterated

   formulas {
      RespTime[i] =
         CompTime[i] + Blockvar[i] +
         sigma(hp, ceiling(RespTime[i] / Period[j]) * CompTime[j]);
      GlobalVar = CompTime[A] + CompTime[B] * CompTime[C];
   }
} ! End of definition file
```

The program can now be run, using the following command:

```
$> fpscalc < definition_file.fps -v
```

When in the verbose mode, the program will now output the following:

```
Number of systems: 1
```

Furthermore, the program will output the found variables and their initial values as follows:

```
Variable ‘Period’
Period[A] = 250.000000
Period[B] = 500.000000
Period[C] = 800.000000
Period[D] = 800.000000
Period[E] = 1000.000000
Period[F] = 2000.000000
Period[G] = 2000.000000
Period[H] = 2000.000000
```

It will also output all the other variables it has found in exactly the same way as the above but these are omitted in this text.

After the initialisation has completed, the program will calculate the blocking factors and output the found values (when in verbose mode) as follows:

```
Variable ‘BlockVar’
Blockvar[A] = 13.000000
Blockvar[B] = 13.000000
Blockvar[C] = 13.000000
Blockvar[D] = 13.000000
Blockvar[E] = 4.000000
Blockvar[F] = 3.000000
Blockvar[G] = 7.000000
Blockvar[H] = 0.000000
```

It will also output the table of semaphores and their calculated ceilings as follows:

```
Semaphores:

Name    Locked by       Time held               ceiling
S2      A               3.000000                1.000000
S2      E               13.000000               1.000000
S4      B               1.000000                2.000000
S4      G               3.000000                2.000000
```

```
S1      C                9.000000                 3.000000
S3      E                4.000000                 5.000000
S3      F                4.000000                 5.000000
S5      G                7.000000                 7.000000
S5      H                7.000000                 7.000000
```

Now it has all the necessary data to start running the variables through the given formula(s), iterating as many times as is necessary to converge to the solutions. When in the verbose mode, it will output the value of the variables after each iteration. After a short while it will display the final values of the variables as follows:

```
System ‘testing’
-------------------

RespTime[A] = 27.000000
RespTime[B] = 77.000000
RespTime[C] = 167.000000
RespTime[D] = 187.000000
RespTime[E] = 228.000000
RespTime[F] = 237.000000
RespTime[G] = 265.000000
RespTime[H] = 288.000000

System ‘testing’
-------------------

GlobalVar = 4514.000000
```

If we examine the results, we can now conclude that the task set is schedulable: all deadlines are met!

Appendices

A Input File Grammar

TaskSet →
  GlobalDecs Systems

GlobalDecs →
  GlobalDecs GlobalDec
  | ϵ

GlobalDec →
  | **indexed** IdentifierList **;**
  | **scalar** IdentifierList **;**
  | **tasks** IdentifierList **;**

IdentifierList →
  IdentifierList , *id*
  | *id*

Systems →
  Systems SystemDef
  | SystemDef

SystemDef →
  system *id* **{** DeclareBlock SemBlock
  Initialisation FormulaBlock **}**

DeclareBlock →
  declarations **{** Declarations **}**

Declarations →
  Declaration Declarations
  | ϵ

Declaration →
  indexed IdentifierList ;
  | **scalar** IdentifierList ;
  | **blocking** *id* ;
  | **priority** *id* ;
  | **tasks** IdentifierList ;

SemBlock $\rightarrow$
    semaphores **{** Semaphores **}**
    | ϵ

Semaphores $\rightarrow$
    Semaphore Semaphores
    | ϵ

Semaphore $\rightarrow$
    semaphore **(** *id* **,** *id* **,** *number* **)** **;**

Initialisation
    initialise **{** Inits **}**
    | ϵ

Inits $\rightarrow$
    Init Inits
    | ϵ

Init $\rightarrow$
    id [*id*] = NumberExpression **;**
    | *id* [**i**[2]] = NumberExpression **;**
    | *id* = NumberExpression **;**

NumberExpression $\rightarrow$
    NumberExpression **+** NumberExpression
    | NumberExpression **-** NumberExpression
    | NumberExpression ***** NumberExpression
    | NumberExpression **/** NumberExpression
    | **-** NumberExpression
    | **(** NumberExpression **)**
    | *number*

FormulaBlock $\rightarrow$
    formulas **{** Formulas **}**

Formulas $\rightarrow$
    Formula Formulas
    | ϵ

Formula $\rightarrow$
    id [*id*] = SimpleExpression **;**

[2] **i** denotes the main task index, described later on in this section.

| *id* [**i**] = Expression **;**
| *id* = SimpleExpression **;**

Expression $\rightarrow$
Expression **+** Expression
| Expression **-** Expression
| Expression ***** Expression
| Expression **/** Expression
| **-** Expression
| **floor** **(** Expression **)**
| **ceiling** **(** Expression **)**
| **sigma** **(hp,** SummationExpression **)**
| **sigma** **(lp,** SummationExpression **)**
| **sigma** **(ep,** SummationExpression **)**
| **sigma** **(all,** SummationExpression **)**
| **max** **(** Expression **,** Expression **)**
| **min** **(** Expression **,** Expression **)**
| **(** Expression **)**
| *number*
| *id*
| *id* [*id*]
| *id* [**i**]

SummationExpression $\rightarrow$
SummationExpression **+** SummationExpression
| SummationExpression **-** SummationExpression
| SummationExpression ***** SummationExpression
| SummationExpression **/** SummationExpression
| **-** SummationExpression
| **floor** **(** SummationExpression **)**
| **ceiling** **(** SummationExpression **)**
| **max** **(** SummationExpression **,** SummationExpression **)**
| **min** **(** SummationExpression **,** SummationExpression **)**
| **(** SummationExpression **)**
| *number*
| *id*
| *id* [*id*]
| *id* [**i**]
| *id* [**j**[3]]

SimpleExpression $\rightarrow$
SimpleExpression **+** SimpleExpression
| SimpleExpression **-** SimpleExpression
| SimpleExpression ***** SimpleExpression

[3]**j** denotes the secondary index, described later on in this section.

| SimpleExpression / SimpleExpression
| **-** SimpleExpression
| **floor (** SimpleExpression **)**
| **ceiling (** SimpleExpression **)**
| **max (** SimpleExpression **,** SimpleExpression **)**
| **min (** SimpleExpression **,** SimpleExpression **)**
| **(** SimpleExpression **)**
| *number*
| *id* [*id*]
| *id*

The *id* lexeme denotes any alphanumeric name, starting with a letter. The *number* lexeme denotes a floating point number in normal or scientific notation.

B Error Messages

There are a number of error messages that can occur during execution. Here is a list of them with a short explanation of possible cause:

- `Bad error: Task/variable not found`
  If any of these errors occur, an internal error in the program has occurred.

- `Conflicting variables`
  `Task index name not found in both system and global declaration`
  `Make sure the names of indexing tasks match`

  The program found a two indexed variables with different sets of indexing tasks in the same formula that were to be automatically indexed (with index i or j). This is only possible when the number of tasks and the names of the indexing tasks are equal.

- `Expression too negative`
  This is a lexical error that indicates that the user has written two minus signs placed directly behind each other.

- `Index used in formula with non-indexed result`
  The result variable of a formula and the variables used inside the formula do not match.

- `Missing blocking factor variable declaration`
  The user tried to declare semaphores, without first defining a blocking factors variable.

- `Missing priority variable declaration`
  The user tried to declare semaphores, without first defining a priority variable.

- `Nested summation`
  The user tried to declare a summation within a summation. This is not allowed.

- `System already defined`
  A system name was used that was already tied to a previously declared system.

- `Task already defined`
  A task was declared more than once in the input file.

- `Variable already defined`
  A variable was declared more than once in the input file.

- `Variable used as indexed, but declared scalar`
  Occurs when trying to put an indexed result into a scalar variable.

- `Variable used as scalar, but declared indexed`
  Occurs when trying to put a single value into an indexed variable.

- `Variables have different dimensions.`
  `Make sure the number of indexing tasks match`

  The program found two indexed variables with different dimensions, used in the same formula that were to be automatically indexed (with index i or j). This is only possible when the number of tasks and the names of the indexing tasks are equal.

C Internal Program Limitations

There are a couple of limitations that should be observed when operating the program. They are as follows:

- Maximum number of systems: 10
- Maximum number of variables/system: 50
- Maximum number of elements per variable: 50
- Maximum number of semaphores/system: 50
- Maximum number of formulas/system: 50
- Maximum formula length: 80 characters
- System stack size: 100 elements

References

[Tin95] Ken Tindell. *Real Time Systems and Fixed Priority Scheduling.* Department of Computer Systems Uppsala University, 1995.