Índice de contenido

Preface

What is InPAWS?

InPAWS is a tool written using C++, that using a text file with a defined syntax (explained along this document), creates a .tap file containing a database that may be loaded in any version of Professional Adventure Writing System by Gilsoft (PAWS) for Sinclair ZX Spectrum.

That is, InPAWS allows you to write Spectrum adventure games without using the built-in PAWS editor.

InPAWS Features

- Generation of PAWS databases in .tap file format using a source code in InPAWS language.
- Allows you to use the graphics and character set generated on PAWS editor.
- Allows you to define and use names for location, objects, messages and flags, so you don't have to remember number.
- Named constants to use on the condacts.
- Comments to improve code readability.
- Support for international characters and any other ASCII codes not supported by PAWS by default: accented characters and any other special character from your language (like spanish Ñ) are allowed to be written straightly on the message text. You just have to tell InPAWS which codes will replace them into the final game (and define the character set accordingly).
- Improved compression algorithm that allows you to create bigger games.
- Messages are defined in the condacts. That is , you can use MESSAGE “Is a pretty pendant” instead of MESSAGE 34 or MESSAGE MesCollar (though these options are also allowed).
- Improved structure of code: connections are defined together with location data, weight, initial location, related words of objects are defined together with the object, etc.
- It is possible to define part of the responses, process, vocabulary, messages as they are needed, not all of them should be defined at the same place.
- Allows the extraction of .Z80 or .SNA files containing a PAWS game to a InPAWS source

file.

What InPAWS cannot do... yet

- 128K mode not supported
- Macros to allow implementation of common tasks
- Many things the author didn't think about! Let him know.

Command line options

If we run InPAWS without parameters we will get the following message:

Syntax: inpaws <command> <input file> [options]

Commands:
c: Compile to tap file (default name: <input file>.tap)
cd: Compile to .SCE source file for PAW PC
cm: Compile to .SCE source file for PAW CPM (Amstrad)
e: Extract source from a 48k PAW .SNA or .Z80
cp: Only generate preprocessed file, without compiling (DEBUGING)
eg: Extract graphics code from a 48k PAW .SNA or .Z80
ec: Extract character info from a 48k PAW .SNA or .Z80

Options:
-o <output file>: Output file name
-s : Symbol detailed list after compilation

Compile [c]

Based in the given input file InPAWS will generate the .tap database, that may be loaded in any version of PAWS. Once we have generated the file, we have to load PAWS into our emulator and use the J option (Load Database). Our database will be loaded into PAWS and from that very moment we can do anything we could do before with PAWS. We can test the game or save it as adventure when we consider it is finished. The output file name will be the same than the input file but with .tap extension.

Compile to .SCE for PAW-PC [cd]

Based on an InPAWS source file, a source file for PAW for PC will be generated. This source file should be compiled with the PC compiler to obtain the adventure. If no output file is specified, the one generated will have the same name than the input one but with SCE extension.

Compilar a .SCE de PAW-CPM(Amstrad) [cm]

Based on an InPAWs source file, a source file for CP/M PAW version will be generated. This source file should be compiled with the CPM compiler (pawcomp) to get the final adventure. By default, on the /CTL section, target disk will be unit A. If no output file is specified, the one generated will have the same name than the input one but with SCE extension.

Extract [e]

This option allows us to extract the source code of any PAWS game from a 48k snapshot in .Z80 or .SNA format. The output file will be an InPAWS text file, that we could later modify and

recompile, and contains everything in the game including graphics and characters (sets, UDGs and shades). Sadly and obviously, it won't contain any added process that the author may have done using the EXTERN condact. That is, the EXTERN calls will be there but not the code called.

If no file name is specified the output is dumped to standard output.

Preprocess [cp]

This command allows you to generate the preprocess file that will be used to generate the .tap file, but without actually generating the .tap fill. This file contains an intermediate representation of code where no symbols exists and the structure is more compact and closer to what a PAWS database is.

This option is available just for debugging reasons, or in the case you would like to check how would the game look on your PAWS before loading it.

If no file name is specified the output is dumped to standard output.

Extract graphics [eg]

Like the *extract* option but extracts only the graphics from a 48k snapshot in .Z80 or .SNA format. Check the Graphics part of this manual to check how you ca include them into your InPAWS source.

Extract characters [ec]

Extracts only the character sets/UDG/shades from a .Z80 or .SNA 48 snapshot. Check the character set related part of this manual to know how to use them.

Options

Output file [-o]

Allows you to specify the output file for any of the previous commands.

Detailed symbol list [-s]

Once we have created the adventure and we are checking it, we will find there is a problem to know which location, object, flag, message has InPAWS assigned to those we didn't assign manually. This option generates a list of those symbols to help with that task.

Introduction to InPAWS syntax

Structure of an InPAWS adventure

InPAWS does not force a predefined order in the data included, you can start defining locations, messages, objects, etc. as you need them or you can maintain the structure you prefer. What InPAWS requires is basically what PAWS require:

- At least one location

- At least one object
- At least one user message
- Basic system messages to allow default responses

If any of these requisites are not matched, InPAWS will tell you, aborting the compilation.

Case sensibility

InPAWS is case insensitive, both for reserved words and symbols. That is, to define a location you can use the reserved word *LOCATION* but you can also use *Location, location or LocATiOn.* That rule is also applied to any symbol or vocabulary words defined in the adventure.

Strings

What we are going to say here is applied to location description, objects, messages and system messages.

Paws allows to add some control characters to texts (ESC codes) that will have some effect like changing text color, background color, change character set, etc.

InPAWS allows you to do that using control characters into curly bracers, and some other options:

{<code>}:

Allows you to insert any ASCII code. For instance {65} would match letter 'A'. The real utility of this ESC code is to add non visible codes as well, like these:

- ***{0-5}***: Selects the character set to use in the rest of message. Exmaple “I will use character set one {1}from here”
- ***{7}***: New line. There is an easier code for this though: '^'.
- ***{16}{<color>}***: Changes the color to the code specified just after: Example: “This text is {16}{4}GREEN”
- ***{17}{<color>}***: Changes the color of the bac to the code specified just after: Example: “This text background is {16}{6}YELLOW”
- ***{18}{<0-1>}***: Flash for the text after the code.
- ***{19}{<0-1>}***: Bright for the text after the code.
- ***{20}{<0-1>}***: Inverse video for the text after the code.

^ :

Used as a new line or carriage return character. Example: “Line 1^Line 2”

\” :

Represents a double quote inside a string. This is allowed to avoid confusing it with the end of string quote. Example: “The ogre says: \”Gimme somethin' \””

\^ :

Represents the '^' character in the code, when you really want to draw that character and not a carriage return. Example: “2\^3 equals 8”.

\{:

Represents the curly braces opening when you really want to draw it, to avoid confusion with a code. Example: “\{This is a text between curly braces}”.

Tips about adventure games generated with InPAWS

Apparently the PAW format is universal, and no differences has been found among the different PAW version relating to database structure. This will help in loading the game in any PAWS version you may have.

On the other hand, there are differences, some of them very important, in the way the different versions of PAWS interpret the same game. The bigger difference is the one in between the spanish and english versions (use of Ñ character, adjectives and pronouns), but there are also some minor differences you will have to take into account. For instance, from version 16 the treatment of conversations with other people (PARSE condact) has some modifications. Another important difference is the spanish version writes the object together with article when a special char _ appears on code.

As summary, you will have to take all that into account when you start creating the adventure. I recommend that you stick to one version from the beginning and never change it.

A side effect of all this is that if you extract database from a file and you load it into a PAWS whose version is different than the one used to create the game, things may not work perfectly. This is not an InPAWS error of course.

InPAWS Syntax

Locations

LOCATION [<location name>] | [<location number>]
{
“Location description”;
[CONNECTIONS { <direction> TO <destination> [<direction> TO <destination> ...] };
}

Location name

When you define a location, you can identify it by name, by number, or both:

- LOCATION House { ... } // Defines a house, InPAWS will assign number automatically
- LOCATION 13 { ... } // Defines location 13, unnamed
- LOCATION House 13 {...} // Define a house, that will have number 13 in PAWS

In the last case you can refer the location by its name, but we force InPAWS to assign that number to the location when creating the game. If we don't specify number InPAWS will assign one automatically.

Most important thing is that once we define a location we would only be able to refer to it by the symbol or value given. For instance, if the first location defined has no number assigned, even when most probably InPAWS will assign it the number 0, we cannot refer to it as 0.

Tips:

- Is recommended to assign the start room the number 0, whether we assign it a name or nor. That way it is sure the player starts the game in this room.
- It is recommended to force the room number in those locations who act as place for

container objects, and define same code for the container object. This is like this as PAWS uses the location with same number than a container object to keep the contents of it.

- It's recommended to define a location code for locations with graphics, as InPAWS support for graphics is very simple and does not allow to assign names to graphics.
- Location numbers needs to be in ascendant order in PAW. That means if you assign a high number to a location, when InPAWS compiles, it will generate the location numbers automatically for the other locations, and then the number of the forced location. If as result there is a gap between the last automatic number and the forced one, InPAWS will fail compilation.

Location description

This is the location description that PAWS shows when you enter the location, check the Strings part of this manual to check what are your options.

Connections

Connections from this location to others. The direction should be a word from vocabulary (verb/name whose number is under 14), and the destination, the name or number of another location(or the same location in case of loop). To add more connections just enqueue them separated by a semicolon.

The connections region can be missing if there are no obvious connections for this location.

Some location samples

```
LOCATION Start 0
{
"{16}{2}The CRAWLIN CHAOS{16}{7}^An adventure by Peter Plot^";
}

LOCATION GloomyFields
{
"You are in a gloomy field, glittering dawn sun rays shine all around.";
CONNECTIONS { N TO River S TO 2 ENTER TO House };
}

LOCATION 2
{
"You are beside a dark cave.";
CONNECTIONS { N TO GloomyFields};
}
```

Objects

```
OBJECT [<object name>] | [<object number>]
{
"Object Description";
```

[INITIALLYAT <location> | CARRIED | WORN | NOTCREATED;]

[WORDS <noun> <adjective>;]

[WEIGHT <weight>;]

[PROPERTY [CLOTHING] [,CONTAINER] ;]

}

Object name

The same rules applied for location descriptions are applied for object names. They may be defined by name, number or both. If you set a number, InPAWS will force that number to be used in the final game. On the rest of the code we will refe that object either by the name or the number.

Some tips about objects:

- Remember the recommendation about setting a forced number for container objects.
- It is recommended to define as object number 0, whether we name it or not, that one that's going to represent the light source, due to the way PAWS has to represent light and darkness.
- Again, object numbers must be continuous, they will start by 0 and they will be getting next number every time one more is defined. If you assign a high number to an object and the automatically assigned ones doesn't reach that number, there will be a gap between the last automatic number and the forced one, and the compiler will fail.

INITIALLYAT

Initial location of the object. You can set the initial location on the object either by name or number. Also, you can set it to some special places using the reserved words CARRIED (object is in player's inventory), WORN (object is worn by player) or NOTCREATED (the object is nowhere, it does not exist when the game starts). By default, if nothing is specified, NOTCREATED will be assumed.

WORDS

Vocabulary words we will use to refer the object when playing. The name and adjective will be set up here, or they can be replaced by a _ character if any of them are not applicable. By default, if no WORDS are specified, both name and adjective will be assumed to be _.

WEIGHT

Object weight. A valuer among 0 and 63. If absent the weight will be considered to be 1.

PROPERTY

Special properties of the objects (attributes). There are two possible options for this, that can be also used together: CLOTHING, when the object can be worn, and CONTAINER, when the object can contain other objects.

Some object examples

OBJECT lantern 0

{

“Una lantern (lit)”;

INITIALLYAT CARRIED;

WORDS LANTE LIT;

```
}

OBJECT sword
{
"A challenging big sword";
INITIALLYAT Lake;
WORDS SWORD _;
WEIGHT 4;
}

OBJECT chest 1
{
"A chest";
INITIALLYAT 5;
WORDS CHEST  _;
WEIGHT 15;
PROPERTY CONTAINER;
}

OBJECT backpack 2
{
"A green backpack";
INITIALLYAT WORN;
WORDS BACKP GREEN;
WEIGHT 2;
PROPERTY CONTAINER, CLOTHING;
}

OBJECT 3
{
"A dumb object";
}
```

Vocabulary

```
VOCABULARY
{
<type> [<number>] : <synonim1> [, <synomin2>...];
...
<type> [<number>] : <synonim1> [, <synomin2>...];
}
```

Explanation of concepts

Type: a type of word chosen among the following : NOUN, VERB, ADJECTIVE,

PREPOSITION, ADVERB, CONJUNCTION, PRONOUN.

Number: the code that should be assigned to the word (optional). If the number is not written, InPAWS will assign a number automatically. The number will be assigned starting by 50 for names, 20 for verbs and 2 for the rest of types.

Synonyms: a quoted list of words, comma separated, which will represent the same word for the code and type defined.

Some tips about vocabulary:

- You can define one or more words on every vocabulary section.
- Some different vocabulary sections can be defined in different places along the code, for instance, as you need the words. You can define vocabulary just after an object definition, for the words that defined the object, or you can define words for the objects in a location, just after the location definition. You can also do it the traditional way and write all the vocabulary together.
- For the special PAW words, (movement, convert and proper name), you should assign the numbers manually. Please pay attention to manually assigned names, as if you define different words with same number, you will be making them synonyms. My recommendation for this kind of words is that you define them altogether on the same section, to avoid confusions.
- Vocabulary numbers don't need to be sorted, so you can assign any number without care (in difference to locations and objects).

Some vocabulary samples

```
VOCABULARY { NOUN: “backp”, “bag”; VERB: “take”, “get”; }
VOCABULARY
{
NOUN 11: “bow”, “BW”; // A name representing a direction
NOUN 12: “poop”, “PO”; // Another one
NOUN 17: “LIST”; // A name that may be used as verb
NOUN 40: “FRODO”; // proper name
PREPOSITION: “IN”, “INTO”;
}
```

User messages and system messages

```
MESSAGES
{
<message number> | <message name> : “Message text”;
...
[<message number> | <message name> : “Message text”;]
}

SYSMESSAGES
{
<message number> | <message name> : “Message text”;
...
[<message number> | <message name> : “Message text”; ]
```

}

General information about messages

InPAWS messages can be defined by number or name, but in this case you cannot define both name and number. You should decide if InPAWS will identify them by number or name, and then use them like that into code.

About the message text, just follow the directives explained above about strings.

You can define one or more messages on each MESSAGES or SYSMESSAGES section. You may be defining the messages as you need them as well (putting them all in the same section is not required).

PAW requisites include the requirement of at least one user message and 54 system messages that are used by the automation condacts or actions like AUTOG, AUTOD, or the location object list. System messages may be modified as you like, but you won't be able to compile a game with less than 54 system messages.

Message definition on the condacts

One powerful feature of InPAWS is allowing the programmer to define the messages exactly when it's needed: into the condacts code. You can just write the text you want to show after the MES, SYSMESS or MESSAGE condacts. In that case, InPAWS will create a new message and assign internally a number.

Example (on response table):

EXAMI SWORD: PRESENT Sword MESSAGE “Sharpened steel, awesome.” DONE;

Of course, this way of adding messages is under all the general restriction for messages (like having no more of 255).

In the case InPAWS detects two or more messages defined like this whose text is the same, it will be clever enough to create just one and use it in every case. Obviously this is only useful if the messages are EXACTLY the same:

LISTE FORES: AT forest MESSAGE “You listen but can't hear anything unexpected” DONE;

LISTE _: MESSAGE “You listen but can't hear anything unexpected” DONE;

This sample above will only generate a message.

Some message samples

MESSAGE

{

0: “This is a fixed number message.”;

MyMessage: “You will have to refer this message by name.”;

}

SYSMESSAGES

{

MsgChest: “Into the chest there are: “;

57: “This is message 57, on PAWS and on InPAWS.”;

}

Colors and default character set

DEFAULTS

{

[CHARSET: <0-5>;]

[INK: <0-9>;]

[PAPER: <0-9>;]

[FLASH: <0-1>;]

[BRIGHT: <0-1>;]

[INVERSE: <0-1>;]

[OVER: <0-1>;]

[BORDER: <0-7>;]

}

This option matches the “B-Background Colors” from PAW, and allows you to stablish the default values for ink, paper, bright, flash, inverse video and border, together with the character set to use.

This values will be used by PAW for every location except if the programmer modifies it with some of the condacts related to this, or by control (ESC) codes into the messages.

All of the values are optional, even the section is optional.

If they are not defined the default values will be like this:

DEFAULTS

{

CHARSET: 0;

INK: 7;

PAPER: 0;

FLASH: 0;

BRIGHT: 0;

INVERSE: 0;

OVER: 0;

BORDER: 0;

}

Flags and constants

FLAG <flag name> [<flag number>];

CONSTANT <constant name> <constant value>;

PAW provides 256 flags and allows to assign numeric constants to almost all condacts (LET, EQ, ISAT). To make this task easier InPAWS allow you to assign names to flags and constants, so you can easily remember them.

We have two options to define a flag: first one is to just saying a name, and in that case InPAWS will assign a number automatically from those available for the user (from 11 to 18, or 60 and above), the other option is manually setting the number. In that case, InPAWS allows you to point to one of the reserved flags, assuming the programmer knows what he/she is doing. This trick allows that you can, for instance, refer to CurrentLocation instead of flag 38.

Important: when we define a flag we are just assigning a name, as the flag are all defined by default by PAW. That is, we can use a flag in condacts wheter we have defined them or not.

In the case of constants assigning a value is mandatory. Constants may be used in the condacts to replace a numeric value, for instance:

```
FLAG moneyAvailable;
CONSTANT maxMoney 100;
_ _ : GT moneyAvailable maxMoney MESSAGE “I'm rich!”
etc
```

Some samples with flags and constants

```
FLAG PlayerLoc 38;
FLAG CurrentObj 51;
FLAG MyNewFlag;
FLAG AnotherFlag 100;
CONSTANT MyValue 150;
```

Process and Response tables

```
RESPONSE
{
 <entry verb> <entry name> : <list of condacts> ;
 ...
 [ <entry verb> <entry name> : <list of condacts> ;]
}

PROCESS <process name> | <process number>; // Process declaration
PROCESS <process name> | <process number>
{
 <entry verb> <entry name> : <list of condacts> ;
```

...

[<entry verb> <entry name> : <list of condacts> ;]

}

Process syntax

The process may be identified in InPAWS by a number, or by a name. Assigning by both ways is not an option. On the rest of the code we will have to refer that process by the name or number specified. Of course, if a name is assigned, InPAWS will assign a number internally.

Entry verb: the verb is matched if the player entered it. You can replace it with the special values _ and * the same way it works in PAWS.

Entry name: same thing, for the name matching the current LS.

Condact list: a list of condacts to run when the entry verb+name match the player input.

Process declaration

As an exception into the InPAWS syntax, is required that processes created by user (3 and above) are defined before being called/used. If a process is used before the declaration the compiler will fail with an error message.

Multiple options, synonims and "OR" operator use

Inpaws allow to specify multiple options (also called sinonyms) on the processes entries, and also at the condact list of those entries that are conditions. To use that we will use the *pipe* character "|" between the entries or condacts we would like to be evaluated as options.

- Using synonims on the entrie headers of the process/response tables: every pair verb-noun should be detailed in the source code, separating every pair with the pipe character.
  - Example: USE KEY| OPEN DOOR: <list of condacts>;
- Using alternative conditions into the entrt body (on condacts): you can separate very alternative option whose condact is a condition with the character "|". All the combinations possible will be generated on the output code in PAWs.
  - Example: OPEN DOOR: CARRIED key |ISAT key pocket SET open ...;

Both capabilities may be combined, it's not a problem:

USE KEY |OPEN DOOR: CARRIED key | ISAT key pocket SET open ...;

Also, we can specify any number of alternative options into the condacts as we want, you just need to know that InPAWS will generate a entry into output code for PAWS for each possible combination. So, for instance if you write one alternative, you will get two entries in output code, but if you add two, you will get four entries, etc. Also, every entry who has more than one pair of verb-noun will be repeated, so an entry with two pairs verb-noun and one alternative condition condact in the code will produce also four entries.

Some other things to take into account

- InPAWS has processes 1 and 2 defined by default. You cannot change the name so you will have to refer them by number.
- For the rest of processes existing, InPAWS will try to make an ordered list in the way it does

for locations, objects and messages, if that is not possible, the compiler will fail.

- Defining all the content of a process or response table in the same section is not required, yo can add new entries as you need in different sections. For instance, you can define the responses that may happen at a given location just after defining it. That is, you can open several PROCESS sections with same number/name in different positions of code, and the compiler will join them before creating the output file.

Entry order in the process/response tables

When PAWS runs a response/process table, it is done from top to bottom of code, trying to find a matching entry until one of them runs a DONE condact. So, the order of the entries is important and the programmer should know it.

PAWS sorts entries in a process by verb code first, and by name code afterwards: Reserved words * and _ represents code 1 and 255. Important: is ordered by word number, not alphabetically.

For same verb+name PAWS keeps the order used when entering the entries. That is, if we have several entries EXAMI SWORD, first one to appear on source code will be first one to appear in PAWS response table.

InPAWS behavior relating to this follows the PAW style line, but it's good to take some things into account to understand the differences between the text source file and what PAWS editor shows:

- The text file for InPAWS will have the entries sorted as you have written them, not caring about the word numbers, but you will have to take into account once it is loaded into PAWS sorted by word number, and so the order may be changed.
- For entries whose verb and name are the same, they will be sorted in PAWS as they are sorted in your InPAWS source file.
- When you define vocabulary words without assigning them a name, you won't be able to know which one has a lower code (remember the order is not alphabetical). So, if you want to have full control or the order you will have to assign a number to those vocabulary words you want to control the order.

Condacts parameter

Parameters used in condacts should match the required data type. Please check PAW technical guide to know all them (you can find the documentation at WOS).

Depending on the parameter type, and based in Gilsoft naming conventions, you should provide the following:

LOCNO / LOCNO+: number or name of a location, as we have defined it. In the case of LOCNO+ we can use the special locations as well: NOTCREATED (252), CARRIED(254) and WORN(253), also we can use 255, which represents the current player location.

MESNO / SYSNO: number or name of a message, as we have defined then into the

MESSAGES o SYSMESSAGES sections, or just a quoted string with the text we want to show.

FLAGNO: number of name of a flag. Remember you don't have to define the flags to use them.

PROCNO: number or name of a process.

WORD: a word defined into vocabulary, or the reserver words _ and *.

VALUE: PAWS documents this, pointing to different value ranges like 0-255, 0-7, etc. If we use this values we can just write the numerical value, but we can also use vocabulary words, location names, object names or constants.

Graphics

InPAWS is a tool to generate PAWS databases from a source file, and as it is, it cannot generate graphics. The solution provided is creating them into PAWS editor, to be later imported into your source file. Once imported into your source file InPAWS can generate them.

We have approached the solution into two different ways: create all the game and add the graphics at the end (way 1), and add graphics as they are needed (way 2)

Way 1

1) Write your adventure, test it, and when you are ready to add graphics go to next step.
2) Compile your InPAWS game, using the -s option to get a list of codes assigned to your locations.
3) Load your database into PAWS, and add the graphics needed using PAWS editor.
4) Once everything is correct, save the adventure (A-Save Adventure)into a tap file, reset the emulated Spectrum in 48k mode and load it as you would do it if you were about to play it. Once it is running, save a snapshot in Z80 format: graphics.z80
5) Run InPAWS with the following syntax: : inpaws eg graphics.z80 -o graphics.txt
6) Open graphics.txt file, that contains a GRAPHICS section that you will have to add (or replace if it exists) on the source file you have.
7) That's all, every time you compile your game again graphics will be included. You can add new objects, change code, change message, etc. You just cannot change the location structure, amount or order, as it may generate a change in the matching of locations an associated graphics.

Way 2 (more flexible)

This is the most recommended approach, as it doesn't force you to create the whole game before adding graphics. These are the steps to follow:

1) Write your game in the usual way, but assign a number to all locations (you can add a name too, but force the number manually, don't let InPAWS find one automatically). You just have to control which codes you have already assigned to assign the next one. Once you have assigned a code to a location, never change it.
2) When you have defined a location that may have a graphic (or subroutine), or just when you decide to add some graphics, generate a tap file and load it into PAWS.

3) Create your graphics, save game, and make a snapshot in the way described above. Extract the graphcis using InPAWS and add the GRAPHICS section to your source code.
4) Continue editing your source code (that will now have the graphics) and when you want to add more graphics repeat this process from step 2.
5) That's all, as you see is more reliable but it forces you to assign numbers to locations.

Character sets

As it happens with graphics, UDG definition, shades and characters sets should be done into PAWS, and the same way should be included into your source code obtaining a CHARACTERS section. In this case you can make the process without fearing to generate a mismatch, as characters sets are not matches with locations as graphics are. Steps to include the character sets, UDGs and shades are:

1) Open PAWS on your emulator (no need to load an adventure) and define all character sets, shades and UDGs you may need.
2) Save the game with PAWS option (A-Save Adventure). Reset the emulator and load the game is you were going to play it.
3) Save a snapshot into file charac.z80.
4) Run InPAWS this way: inpaws ec charac.z80 -o charac.txt
5) Open your source file and replace all the CHARACTERS section of your source with the one appearing into the charac.txt file.
6) That's all, if you need to modify anything, load you game into PAWS, change it, and go back to step 2.

Preprocessor options

For those new to programming, a directive is a command that does not belog to the programming language itself, that makes the compiler do something during compilation time. Inpaws provides a basic support for two targets:

- Allowing the source code to be divided into serveral files to make editing easier
- Decide, depending on hardware we are doing the adventure for, if some parts of the code should be taken into account or not.

All InPAWS directives starts by character #

Including files

To help managing large source files, it is possible to split those files into several ones and join them at compile time. To do that we may use the #INCLUDE directive.

#INCLUDE “<file to include name>”

Once the compiler process that directive, it will continue reading the new file until its end, and then continue the current file just after the INCLUDE directive.

There is no problem in adding #INCLUDE directive into a file already included by others, but you will have to be careful not including files making an endless loop (A includes B and B includes

A, or A includes B, B includes C, C includes A, etc.). You will get a warning from compiler if you do that though.

All the files included should be on the same folder the “parent” file is, so no paths are allowed into the include directive, just file names.

Conditional compiling

The aim of conditional compiling is force or ignore some parts of the source code depending on the existance of some declared *labels* or preprocessor variables. In the case of InPAWS, support for conditional compiling has been built to help creating adventures for several systems (Spectrum, Amstrad or PC). The programmer should consider every system capabilities and use conditionals depending on that. Anyway, of course conditional compiling is not restricted to that, and if the programmer finds another utility, is free to use them as he/she pleases.

#DEFINE <label>

The label named will be defined, and from this point it will “exist” for the rest of compilation.

Labels do not have a value, they just exists or no, to be able to be checked by #IFDEF and #IFNDEF directives .

Inpaws has some labels predefined depending of the compilation option used. For instance, if you used the “c” option (compile for Spectrum) the label PAWSPECTRUM will exis automatically. If you select “cd” option (for PC) the label PAWPC will exist, and finally if you compiled for Amstrad, the predefined label will be PAWCPM.

#IFDEF <label>

... Code ...

#ENDIF

The code among the IFDEF AND ENDIF directives will only be taken into account if the label specified “exist”, otherwise it will be ignored.

#IFNDEF <variable>

... Código ...

#ENDIF

The code among the IFNDEF AND ENDIF directives will only be taken into account if the label specified “does not exist”, otherwise it will be ignored.

Nesting #IFDEF and #IFNDEF into the same file is not allowed. Those familiar with comditional compiling should know InPAWS does not support the usual #ELSE directive.

International Characters

SUBCHAR “<character>” “<replacing string>”;

International characters problems

[Note of translator: *I've been thinking if removing this section from manual as apparently it doesn't make any sense to talk about non english characters on english documentation. Finally, I have decided to add it though, as this documentation may be used for non english people to make their games in their own languages. On the other hand, the examples has not been translated, as obviously I cannot find any english sentence using international characters, so I have keep the spanish examples hoping those who need to use international characters may understand the way it is done, and use it for their own languages.]*

Character set managed by Spectrum is the standard ASCII up to code 127 (actually, something more for the UDG). Everything above that in the current PCs is not manageable by the Spectrum. PAW uses those codes over 127 to generate the compression system.

That means that if you write a message into your editor like “Estás en un camión añil” the emulator will show something like “Est STOP s en un cami NOTn a\il”, or worse.

To solve this problem, what is usually done in paws is redefine some UDG or characters from character set that are not used often (like %, $ or backslash) that are shown by Spectrum to actually print those codes. Once defined you would have to use them into your messages this way:

“Est{144}s en un cami{158}n a\nil”.

This allows us to use our language characters into our game, but it makes harder the creation of the messages, as we have to take into account all those special characters.

To make that easy InPAWS allows you to write the messages in a normal way (using your characters) and specify which code will replace those chars when the final tap file is generated. For instance, if you have defined accented “a” letter (á) using the first UDG (code 144), just write this at the beginning of your source file:

SUBCHAR “á” “{144}”;

This makes InPAWs replace all apparitions of “á” into messages, locations, etc. by {144} before dumping it to tap file. That is, if you typed “Estás en un sitio muy bonito”, InPAWS will actually generate “Est{144}s en un sitio muy bonito”.

The replaced character can only be one character, not a string. The replacing string may have any length while you keep the rules to make any string as explained above on the Strings section of this manual.

Some char replacing samples

SUBCHAR "¿" "{146}";

SUBCHAR "á" "{144}";

SUBCHAR "é" "{148}";

SUBCHAR "í" "{152}";

SUBCHAR "ó" "{158}";

SUBCHAR "ú" "{159}";

SUBCHAR "¡" "{147}";

SUBCHAR "ñ" "{149}";

SUBCHAR "Ñ" "{150}";

SUBCHAR "ü" "{151}";

Acknowledgments

To Graeme Yeandle and Tim Gilberts for creating PAW.

To all authors of UNPAWS tool: José Luis Cebrian (original version), Carlos Sánchez (Pascal version and Z80 loader) and Alexander Katz (128k and graphics support), for the archaeological work done on the PAWS database format; without their work this tool would not have been possible.

Contact the author

Currently (February,2009) you can send your suggestions, complaints, bug reports, or cash to the following address:

lane.mastodon [at] gmail.com

Last update: February 16th, 2009

Documentation Translation by: Carlos Sánchez

© 2009 Francisco Javier López