Overview:

The MultiCell function is part of the extensive PDF generating library that was developed by Olivier Plathey. The library has become a major source of PDF generation within PHP-based applications. The use of the MultiCell function has generated occasional confusion and mis-understanding among new users to the FPDF library. It is the intent of this paper to explain in detail the basic structure of the function as well as providing application examples.

Intended Audience:

The contents of this paper are intended for individuals who have successfully installed the FPDF library on their web server. Additionally, this paper assumes that the reader has at least an intermediate level of experience in writing PHP applications.

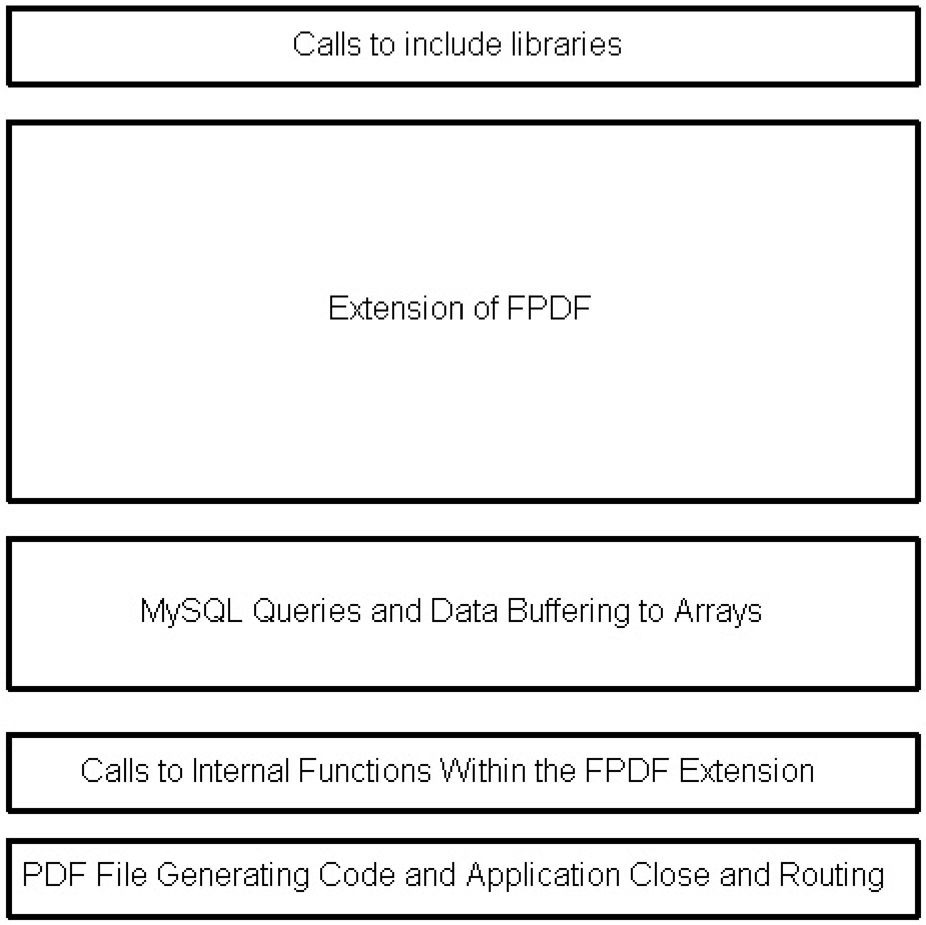
Conventions Used:

The following illustrated code examples will use these conventions:

1. Data is buffered into numerically indexed array(s).
2. The actual PDF generation is accomplished by calls to custom functions built within the program's FPDF call. As such, the examples will be based on a high level of object oriented coding.

The PHP FPDF applications illustrated in this paper will follow the program structure shown in Figure 1.

Figure 1 Application Block Structure


Calls to include libraries
Extension of FPDF
MySQL Queries and Data Buffering to Arrays
Calls to Internal Functions Within the FPDF Extension
PDF File Generating Code and Application Close and Routing

Difference Between a MultiCell() and a Cell():

Both the Cell() and MultiCell() functions support the display of data in table format. The MultiCell function allows text to be wrapped to new lines within the boundaries established for the width of the MultiCell. In the Cell function, text is only displayed within the originating line of the Cell call. It is not possible to accommodate wrapped text within the Cell function.

MultiCell Basics

The MultiCell consist of the following construction:

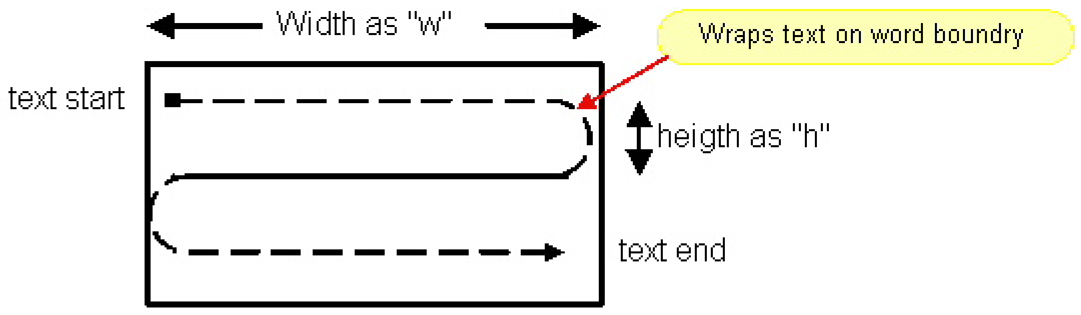
MultiCell(width of cell, height of each line, text content, border, alignment of the text, fill boolean).

An example would be :

```
$this ->MultiCell(25,6,”Here's some text for display”, 'LRT', 'L', 0);
```

In the above example, the MultiCell width is set at 25 units that were specified in the initial call to FPDF. Each line will have a height of 6. The text is self-explanatory. The border (LRT) will display a border at the left L, right R and top T. The text will be left justified in relation to the left boundary of the MultiCell. The MultiCell will not be filled solid as noted in the last variable in the MultiCell.

Figure 2 : MultiCell Geometries



Single Column Table:

The following section discusses the construction of a single column table. The table contents will consist of multiple rows within the table.

After the data has been buffered to an array, it is necessary to do the basic startup of the pdf generation. The following code shows an example of this startup:

```
$file_nm='test.pdf';
$pdf=new PDF();
$pdf->Open($file_nm);
$pdf->SetTopMargin(10);
$pdf->AddPage(L);
```

This will initiate the creation of a new pdf file called “test.pdf” . The top margin is being set at a value of 10 and the resulting PDF file will be in landscape orientation as shown in the AddPage(L) call.

After the basic setup has been achieved, it is time to run the code that will actually compile the pdf for output.

The next step will involve passing the text data that has been buffered to the specific function that will do the processing. For this example a single field of data is passed in array ar1. Each row (record) of data will comprise a single element in the array. The array ar1 is a one dimensional numerically indexed array.

```
$pdf->Section($ar1);
```

The PHP environment that is being used for coding this program has the global registration set to off. As a result, the data will be passed directly through the call to the Section() function rather than reference through a global variable.

At the top of our PDF class, it is recommended to include some basic functionality that has been developed by several contributors to the FPDF project. The basic initial class for this exercise is defined as:

```
class PDF extends FPDF
{

        #******************************
        function PDF($orientation='l', $unit='mm', $format='tabloid')
        {
                $this->FPDF($orientation,$unit,$format);
        }
```

The next recommended function to include in the basic class is the function for sensing where page breaks need to occur.

```
        function CheckPageBreak($h)
        {
                #If the height h would cause an overflow, add a new page immediately
                if($this->GetY()+$h>$this->PageBreakTrigger)
                {

                        $this->AddPage($this->CurOrientation);
                }
        }
```

Because the application contains variable length text, it will be necessary test for a page break before the row of data is generated. This is accomplished through the NbLines() function which is the next core function to be included in the class extension.

```
function NbLines($w,$txt)
        {
                //Computes the number of lines a MultiCell of width w will take
                $cw=&$this->CurrentFont['cw'];
                if($w==0)
                $w=$this->w-$this->rMargin-$this->x;
                $wmax=($w-2*$this->cMargin)*1000/$this->FontSize;
                $s=str_replace("\r",'',$txt);
                $nb=strlen($s);
                if($nb>0 and $s[$nb-1]=="\n")
                $nb--;
                $sep=-1;
                $i=0;
                $j=0;
                $l=0;
                $nl=1;
                while($i<$nb)
                {
                        $c=$s[$i];
                        if($c=="\n")
                        {
                                $i++;
                                $sep=-1;
                                $j=$i;
                                $l=0;
                                $nl++;
                                continue;
                        }
                        if($c==' ')
                        $sep=$i;
                        $l+=$cw[$c];
                        if($l>$wmax)
                        {
                                if($sep==-1)
                                {
                                        if($i==$j)
                                                $i++;
                                }
                                else
                                        $i=$sep+1;
                                $sep=-1;
                                $j=$i;
                                $l=0;
                                $nl++;
                        }
                        else
                                $i++;
                }
                return $nl;
        }
```

With the building of the support functions, the coding for the contents of the single column table can start.

```
        function Section($ar1)
        {
                #       add the page header
                $this->CustomHeader();

                #       set the font that will be used
                $this->SetFont('Arial', '', 9);

                for($i=0; $i<count($ar1); $i++)
                {
                        #       bring the array element back to a local variable f1
                        $f1 = $ar1[$i];

                        #       the following will return the number of lines for the
                        #       text field f1 based on the font selected above as well
                        #       as a MultiCell width of 25
                        $lines = $this->NbLines($f1, 25);

                        #       variable hx will yeild the amount of Y consummed by this
                        #       line of text

                        $hx = $lines * 6;
```

From the code listed, the data array has been brought into the function Section(). The first thing generated is the CustomHeader for the data page (this will be covered later).

After the header is issued, the default font used in the table contents is set.

From this point forward, the program is going to reiterate through the array containing the text data one row (record) at a time. The commented areas in the code should provide adequate explanation of what is going on. Note that the multiplier of 6 used in the $hx variable will probably need a little experimentation if the font used is anything but Arial size 9.

```
                        #       now going to check to see if the text in f1 will cause
                        #        a page break

                        if($this->GetY() + $hx > $this->PageBreakTrigger)
                        {
                                #       going to add a new page
                                $this->AddPage($this->CurOrientation);
                        #       now going to check to see if the text in f1 will cause
                        #        a page break
                        if($this->GetY() + $hx > $this->PageBreakTrigger)
                        {
                                #       going to add a new page
                                $this->AddPage($this->CurOrientation);
                                #       add the page header
                                $this->CustomHeader();
                                #       space the start of the text 5 mm below the end of the header
                                $this->Ln(5);
                                #       reset the font to what is needed for the text display (content)
                                $this->SetFont('Arial', '', 9);
                        }
```

The code illustrated above deals with text that may extend beyond the current page break boundary. If the incoming text exceed the limits base on the current Y position **plus** the amount of Y consumed by the text field $f1, then a page break is issued along with a new header. The text font is re-defined in this slice of code because a different size font may be used in the CustomHeader.

The actual MultiCell is generated from the following code.

```
                        #       now going to generate the MultiCell
                        $this->MultiCell(25,6,$f1,1,'L',0);

                        #       if you want a space between the next row, insert a value
                        #       in the Ln() call, otherwise insert a 0
                        $this->Ln(0);
                }
        }
```

From the example above, the width is set at “25” and the row height is “6”. The text used to populate the MultiCell is “$f1”. The next variable “1” is for the border. In this example the border used will be a complete frame around the MultiCell. The text within the MultiCell will be left justified as defined with the value of “L”. Finally, this MultiCell will not be filled solid as reflected in the last variable value of “0”.

The Ln(0) call is issued to bring the active XY position back to the X origination. If the layout of the table calls for physical spacing between rows, a value other than 0 would be inserted within the Ln(0) call, other wise issue a 0 value as shown.

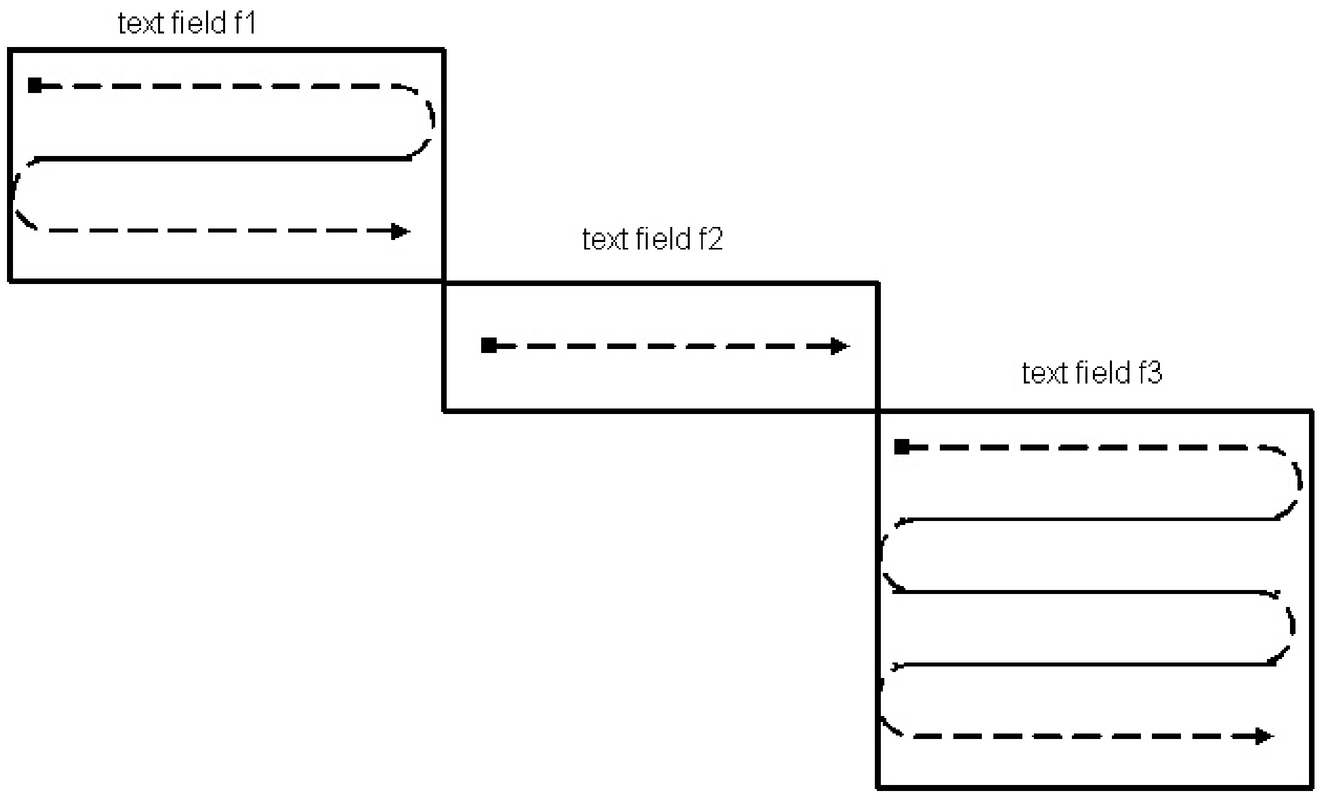
This completes the construction of a single column table using the MultiCell method.

Multiple Column Table

The multiple column table presents a few additional considerations when the MultiCell is used. The following code example assumes that there are 3 fields of variable text data in the data array $ar1 (f1, f2, and f3). Each of these fields is of variable length from record to record. It is also assumed that a uniform border will be drawn around each cell in the table.

Following the example used in the single column table, if nothing was done with the 3 fields other than to initiate a MultiCell call to each text field, the resulting table row might look like the following illustration.

Figure 3 :



This is not a desirable table layout from a formatting standpoint. Each subsequent row of the table would vary depending upon the variable length of each field.

The recommended way to resolve this situation is to do the pre-calculation of the maximum row height **before** the multiple MultiCell calls are made. The engine for this calculation comes from the function NbLines() that was previously built into the top of the class extension.

```
function Section($ar1)
{
        #       add the page header
        $this->CustomHeader();

        #       set the font that will be used
        $this->SetFont('Arial', '', 9);

        for($i=0; $i<count($ar1); $i++)
        {
                #       bring the array element back to a local variable f1, f2, f3
                $f1 = $ar1[$i][0];
                $f2 = $ar1[$i][1];
                $f3 = $ar1[$i][2];

                #       the following will return the number of lines for the
                #       text field f1, f2, and f3
                $nb1 = $this->NbLines($f1,25);
                $nb2 = $this->NbLines($f2,35);
                $nb3 = $this->NbLines($f3,20);

                #       variable hx will yeild the amount of Y consummed by this
                #       line of text

                $hx = max($nb1, $nb2, $nb3) * 6;
```

In the example, a separate variable is assigned to the NbLines() calculation for each text field. Also note, that for this example different width cells are used for each data field as identified in the second variable in the respective NbLines calls.

The final part of this preparation work calculates then maximum value of the Y dimension that will be consumed with this row of data (variable $hx).

At this point in the application it is necessary to test whether a page break is necessary before generating the next row of the table.

```
#       now going to check to see if the text in f1 will cause
#        a page break

if($this->GetY() + $hx > $this->PageBreakTrigger)
{
        #       going to add a new page
        $this->AddPage($this->CurOrientation);
        #       add the page header
        $this->CustomHeader();
        #       space the start of the text 5 mm below the end of the header
        $this->Ln(5);
        #       reset the font to what is needed for the text display (content)
        $this->SetFont('Arial', '', 9);
}
```

There is some additional preparation that needs to be done at this point before the actual MultiCell generation starts. Since a new row of the table is starting, the starting point X and Y values need to be assigned to distinct variables.

```
$startx = $this->GetX();
$starty = $this->GetY();
$rowmaxy = $starty + $hx;
```

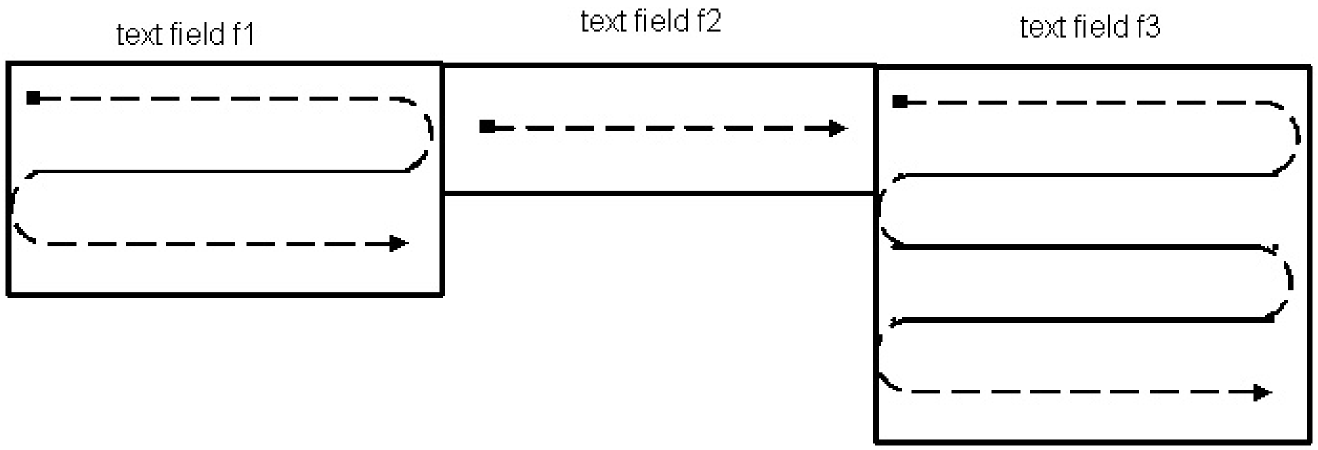
The variable $rowmaxy reflects the maximum Y position for the new row of data.

```
#       start of the MultiCell for field f1
#       set our current position to the starting point
$this ->SetXY($startx, $starty);
#       actual MultiCell for f1
$this->MultiCell(25,6,$f1,'LRT','L',0);
```

The above code will generate the actual first MultiCell. Note that the starting coordinate is set with the SetXY($startx, $starty) call.

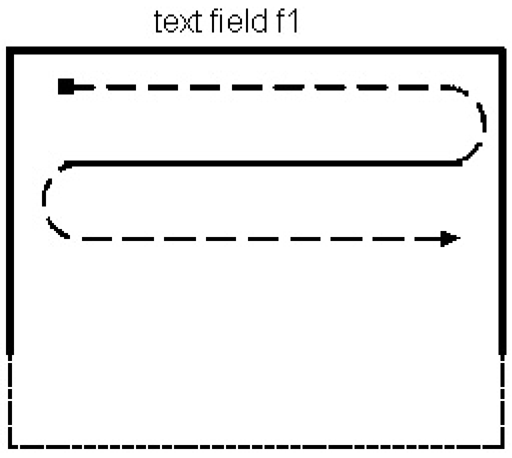
The next part gets a little bit tricky inasmuch the framing on each cell (MultiCell) is going to be balanced out for the row. If this step is not performed, the resulting table row might look like Figure 4 :

Figure 4:



That's better than the previous example but still not good formatting for the row of tabled data. The challenge is to finish off the bottom portion of each cell so that all cells within the row line up with respect to the bottom maximum Y dimension for the row. Figure 5 illustrates the situation for completing the cell.

Figure 5 :



The bold lines show the border that was drawn using the 'LRT' call within the MultiCell. The dotted border reflects the area that needs to be drawn to finish off the MultiCell.

The following code will finish off the border for the variable length text field “f1”.

```
#       will set the present Y location with a new variable $tempy
$tempy = $this->GetY();
#       now set the present location to the starting point on
#       the X axis and our current Y position
$this->SetXY($startx, $tempy);
#       now test to see if this Y location is less than our row
#       maximum Y as calculated earlier as variable $rowmaxy
if($temp < $rowmaxy)
{
        #       compute the Y distance from the current temporary Y
        #       to the row maximum Y ($diffy)
        $diffy = $rowmaxy - $tempy;
        #       now going to draw a pseudo multi-cell with no text to
        #       finish the cell outlilne
        $this->MultiCell(25,$diffy, '' , 'LRB', 'C', 0);
}
else
{
        #       we are at the row's maximum Y  Only need to draw the
        #       bottom border of the cell
        $this->MultiCell(25,0,'','B','C',0);
}
```

The only thing that remains for this cell is to now increment the X axis start point to the end of the cell we just created.

```
$addx = $this->GetX();
$startx+= $addx;
```

Repeat this same process for each of the remaining text fields ($f2 and $f3). The following code finishes off the generation of the MultiCell table for the data contained in $ar1.

```
                        #       insert duplicate type of code for f2 and f3 here.  Adjust
                        #       the variables for the cell width where applicable

                        #       at the completion of the row of data :
                        #
                        $this->Ln(0);
                        #       this will set the current location at the maximum Y for the row and
                        #       the X location will be at the left margin
                }
```

If the table contains a fair number of cells, it is recommended to move the functionality of the border finishing calculations and generation to a callable function with the associated variables passed to the function. This will save a lot of repetitive code generation.

Finishing off the PDF Generation :

It is now time to return to the bottom portion of the program to finalize the actual PDF generation as shown in the bold text below.

```
$file_nm='test.pdf';
$pdf=new PDF();
$pdf->Open($file_nm);
$pdf->SetTopMargin(10);
$pdf->AddPage(L);
$pdf->Section($ar1);
$pdf->Output($_SERVER['DOCUMENT_ROOT'].'/PDF/'.$file_nm);
$pdf->Close();

include $_SERVER['DOCUMENT_ROOT'].'/reports/menu1.php';

exit;
?>
```

Custom Header :

As mentioned previously, for this example a custom header was utilized. The custom header is embedded within the class extension of the FPDF as a separate callable function. If your project gets complex, you can create as many different custom headers as required. Call the headers within your equivalent to the Section() part of the code. As an illustration, in an application that I created several years ago, the PDF produced had 14 different forms embedded as part of the PDF output file. Each form had it's own unique header as called by the respective section generating code. To help you get started, here's a snippet of a simple custom header.

```
function CustomHeader($ar2)
{
        $project_in     =       $ar2[0];
        $fdesc_in               =       $ar2[1];
        $family_in              =       $ar2[2];
        $fchild_in              =       $ar2[3];

        $this->SetFillColor(217,217,217);
        $this->SetFont('Arial','B',18);
        #       main title - fixed length so we used the cell vs mulitcell
        $this->Cell(0,10,'A Sample Custom Header',0,1,'C',0);
        $this->Ln(0);

        #row of captions
        $this->SetFont('Arial','',9);
        $this->Cell(30,4,'Project',0,0,'C',0);
        $this->Cell(105,4,'Description',0,0,'C',0);
        $this->Cell(15,4,'Family',0,0,'C',0);
        $this->Cell(105,4,'Child',0,1,'C',0);
        $this->Ln(0);

        #row of values
        $this->SetFont('Arial','',12);
        $this->Cell(30,6,$fproject_in,1,0,'C',0);
        $this->Cell(105,6,$fdesc_in,1,0,'C',0);
        $this->Cell(15,6,$family_in,1,0,'C',0);
        $this->Cell(105,6,$fchild_in,1,1,'C',0);
        $this->Ln(0);
        $this->Cell(255,2,str_repeat('_',130),0,1,'C',0);
        $this->Ln(2);
}
```

Conclusion :

The MultiCell function facilitates the generation of complex table structures within the FPDF PDF generating library. When used properly, the application can handle variable length data within each cell of the table. Simplistically stated, the MultiCell function is all about geometries. Within the application, the developer needs to keep track of the current X and Y locations in relation to the coordinates that the new row of data started with.

Olivier Plathey and the other developers of the FPDF library are to be commended for producing such a high quality, stable product for open source usage. In the five + years that I have used the library, I have found it's application to be consistently stable and meeting the requirements for successful PDF generation. The library does come with a rather significant learning curve so the developer should be prepared to devote adequate time to learn the effective use of the library.

Links:

The core library and supporting documentation can be found at:

http://www.fpdf.org/